CSE 431/531: Algorithm Analysis and Design (Spring 2021)
Graph Algorithms

Lecturer: Shi Li

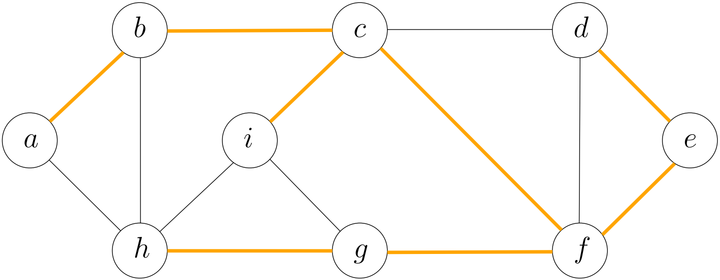
Department of Computer Science and Engineering
University at Buffalo

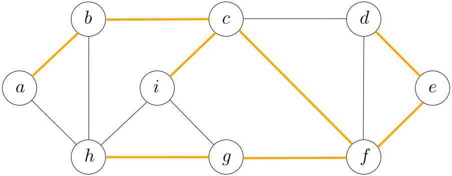
Outline

1. Minimum Spanning Tree
   - Kruskal's Algorithm
   - Reverse-Kruskal's Algorithm
   - Prim's Algorithm
2. Single Source Shortest Paths
   - Dijkstra's Algorithm
3. Shortest Paths in Graphs with Negative Weights
   - Bellman-Ford Algorithm
4. All-Pair Shortest Paths and Floyd-Warshall

Spanning Tree

Def. Given a connected graph $G = (V, E)$, a spanning tree $T = (V, F)$ of G is a sub-graph of G that is a tree including all vertices V.





Lemma Let $T = (V, F)$ be a subgraph of $G = (V, E)$. The following statements are equivalent:

- T is a spanning tree of G;
- T is acyclic and connected;
- T is connected and has $n - 1$ edges;
- T is acyclic and has $n - 1$ edges;
- T is minimally connected: removal of any edge disconnects it;
- T is maximally acyclic: addition of any edge creates a cycle;
- T has a unique simple path between every pair of nodes.

Minimum Spanning Tree (MST) Problem

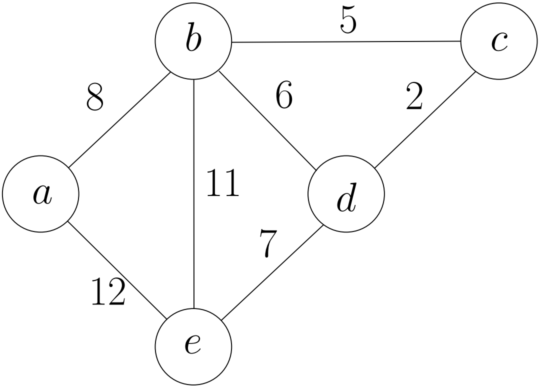
Input: Graph $G = (V, E)$ and edge weights $w : E \to \mathbb{R}$

Output: the spanning tree T of G with the minimum total weight

Minimum Spanning Tree (MST) Problem

Input: Graph $G = (V, E)$ and edge weights $w : E \to \mathbb{R}$

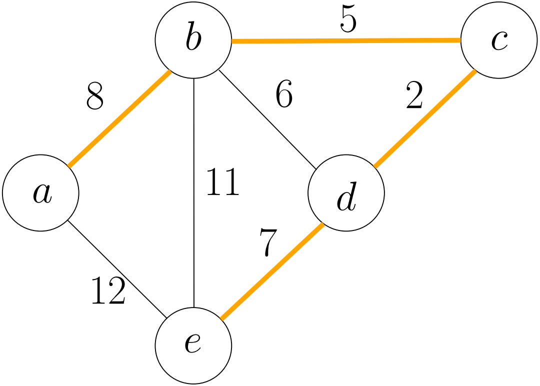
Output: the spanning tree T of G with the minimum total weight



Minimum Spanning Tree (MST) Problem

Input: Graph $G = (V, E)$ and edge weights $w : E \to \mathbb{R}$

Output: the spanning tree T of G with the minimum total weight



Recall: Steps of Designing A Greedy Algorithm

- Design a "reasonable" strategy
- Prove that the reasonable strategy is "safe" (key, usually done by "exchanging argument")
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually trivial)

Def. A choice is "safe" if there is an optimum solution that is "consistent" with the choice

Recall: Steps of Designing A Greedy Algorithm

- Design a "reasonable" strategy
- Prove that the reasonable strategy is "safe" (key, usually done by "exchanging argument")
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually trivial)

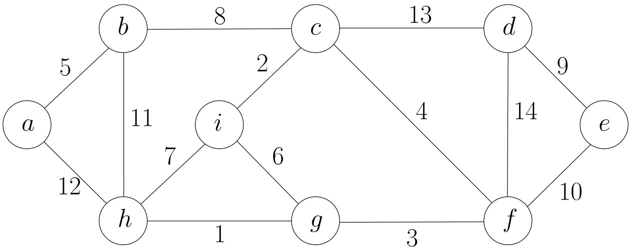
Def. A choice is "safe" if there is an optimum solution that is "consistent" with the choice

Two Classic Greedy Algorithms for MST

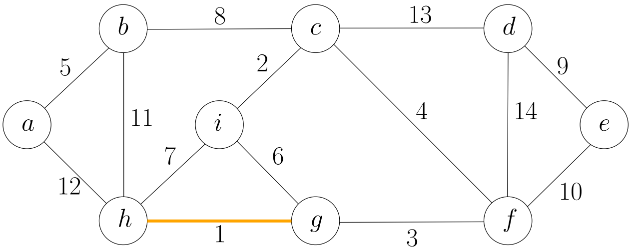
- Kruskal's Algorithm
- Prim's Algorithm

Outline

1. Minimum Spanning Tree
   - Kruskal's Algorithm
   - Reverse-Kruskal's Algorithm
   - Prim's Algorithm
2. Single Source Shortest Paths
   - Dijkstra's Algorithm
3. Shortest Paths in Graphs with Negative Weights
   - Bellman-Ford Algorithm
4. All-Pair Shortest Paths and Floyd-Warshall



Q: Which edge can be safely included in the MST?



Q: Which edge can be safely included in the MST?

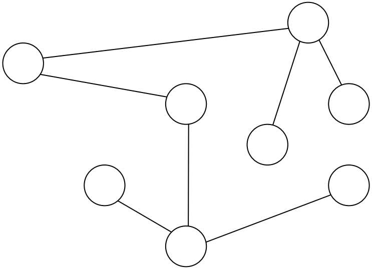
A: The edge with the smallest weight (lightest edge).

Lemma It is safe to include the lightest edge: there is a minimum spanning tree, that contains the lightest edge.

Lemma It is safe to include the lightest edge: there is a minimum spanning tree, that contains the lightest edge.

Proof.

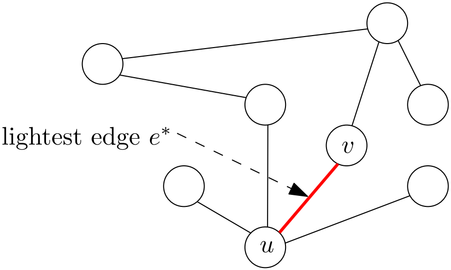
- Take a minimum spanning tree T



Lemma It is safe to include the lightest edge: there is a minimum spanning tree, that contains the lightest edge.

Proof.

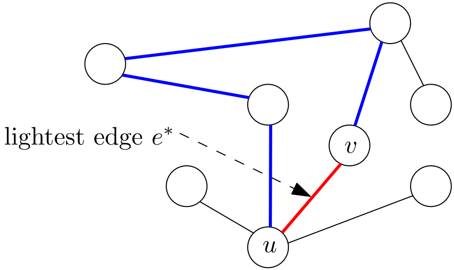
- Take a minimum spanning tree T
- Assume the lightest edge e^* is not in T



Lemma It is safe to include the lightest edge: there is a minimum spanning tree, that contains the lightest edge.

Proof.

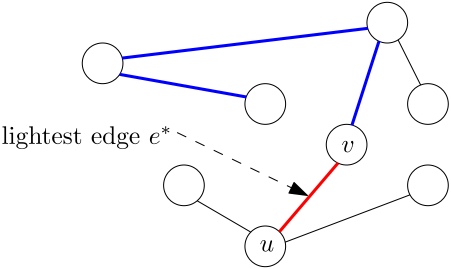
- Take a minimum spanning tree T
- Assume the lightest edge e^* is not in T
- There is a unique path in T connecting u and v



Lemma It is safe to include the lightest edge: there is a minimum spanning tree, that contains the lightest edge.

Proof.

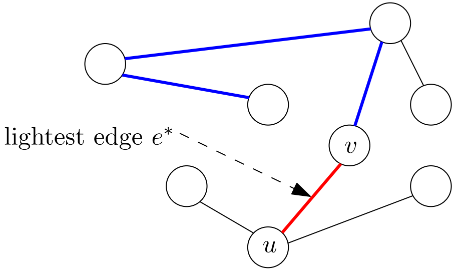
- Take a minimum spanning tree T
- Assume the lightest edge e^* is not in T
- There is a unique path in T connecting u and v
- Remove any edge e in the path to obtain tree T'



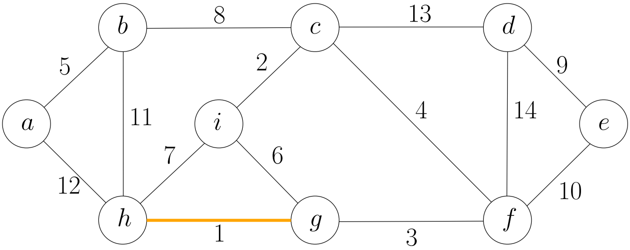
Lemma It is safe to include the lightest edge: there is a minimum spanning tree, that contains the lightest edge.

Proof.

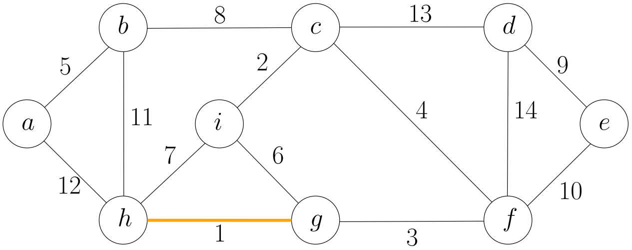
- Take a minimum spanning tree T
- Assume the lightest edge e^* is not in T
- There is a unique path in T connecting u and v
- Remove any edge e in the path to obtain tree T'
- $w(e^*) \leq w(e) \implies w(T') \leq w(T)$: T' is also a MST □



Is the Residual Problem Still a MST Problem?

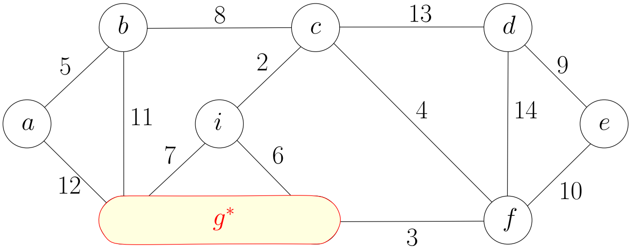


Is the Residual Problem Still a MST Problem?



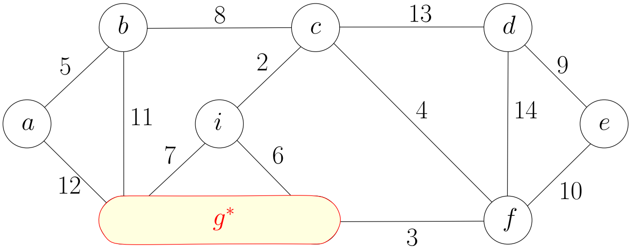
- Residual problem: find the minimum spanning tree that contains edge (g, h)

Is the Residual Problem Still a MST Problem?



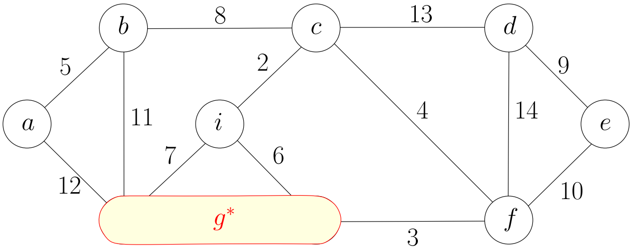
- Residual problem: find the minimum spanning tree that contains edge (g, h)
- Contract the edge (g, h)

Is the Residual Problem Still a MST Problem?

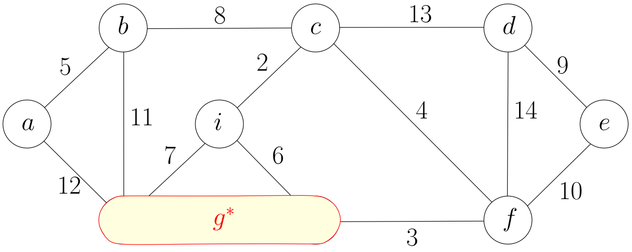


- Residual problem: find the minimum spanning tree that contains edge (g, h)
- Contract the edge (g, h)
- Residual problem: find the minimum spanning tree in the contracted graph

Contraction of an Edge (u, v)

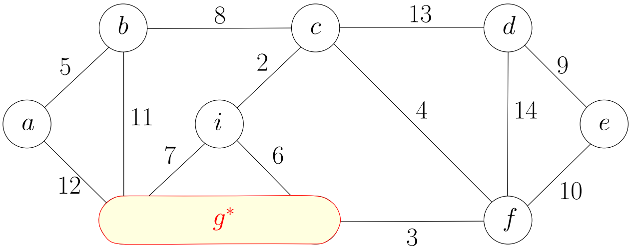


Contraction of an Edge (u, v)



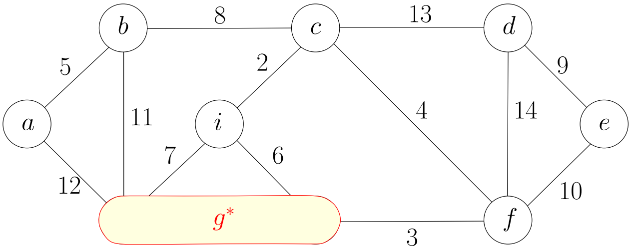
- Remove u and v from the graph, and add a new vertex u^*

Contraction of an Edge (u, v)



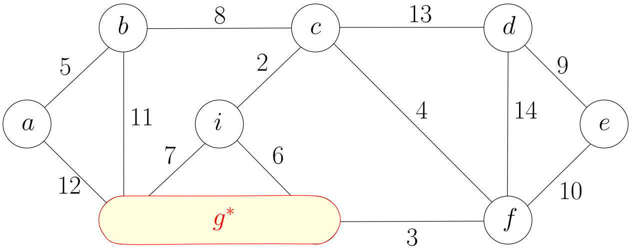
- Remove u and v from the graph, and add a new vertex u^*
- Remove all edges (u, v) from E

Contraction of an Edge (u, v)



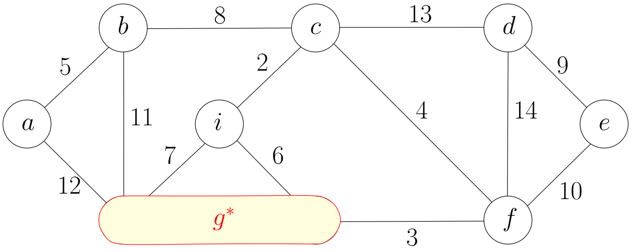
- Remove u and v from the graph, and add a new vertex u^*
- Remove all edges (u, v) from E
- For every edge $(u, w) \in E, w \neq v$, change it to (u^*, w)

Contraction of an Edge (u, v)



- Remove u and v from the graph, and add a new vertex u^*
- Remove all edges (u, v) from E
- For every edge $(u, w) \in E, w \neq v$, change it to (u^*, w)
- For every edge $(v, w) \in E, w \neq u$, change it to (u^*, w)

Contraction of an Edge (u, v)



- Remove u and v from the graph, and add a new vertex u^*
- Remove all edges (u, v) from E
- For every edge $(u, w) \in E, w \neq v$, change it to (u^*, w)
- For every edge $(v, w) \in E, w \neq u$, change it to (u^*, w)
- May create parallel edges! E.g. : two edges (i, g^*)

Greedy Algorithm

Repeat the following step until G contains only one vertex:

1. Choose the lightest edge e^*, add e^* to the spanning tree
2. Contract e^* and update G be the contracted graph

Greedy Algorithm

Repeat the following step until G contains only one vertex:

1. Choose the lightest edge e^*, add e^* to the spanning tree
2. Contract e^* and update G be the contracted graph

Q: What edges are removed due to contractions?

Greedy Algorithm

Repeat the following step until G contains only one vertex:

1. Choose the lightest edge e^*, add e^* to the spanning tree
2. Contract e^* and update G be the contracted graph

Q: What edges are removed due to contractions?

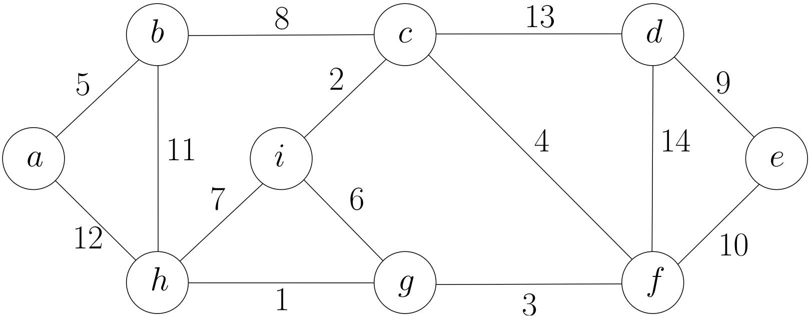
A: Edge (u, v) is removed if and only if there is a path connecting u and v formed by edges we selected

Greedy Algorithm

MST-Greedy(G, w)

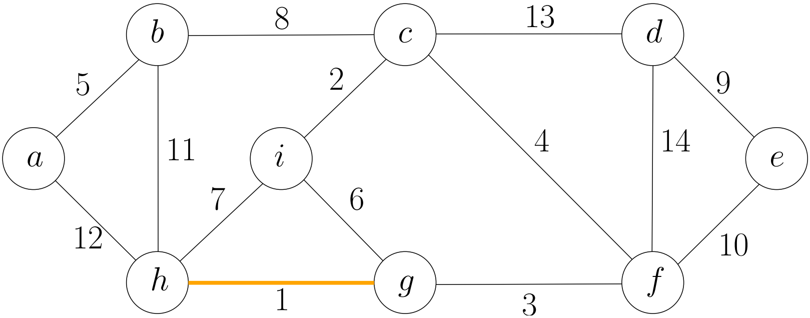
$F \leftarrow \emptyset$
sort edges in E in non-decreasing order of weights w
for each edge (u, v) in the order **do**
    if u and v are not connected by a path of edges in F **then**
        $F \leftarrow F \cup \{(u, v)\}$
return (V, F)

Kruskal's Algorithm: Example



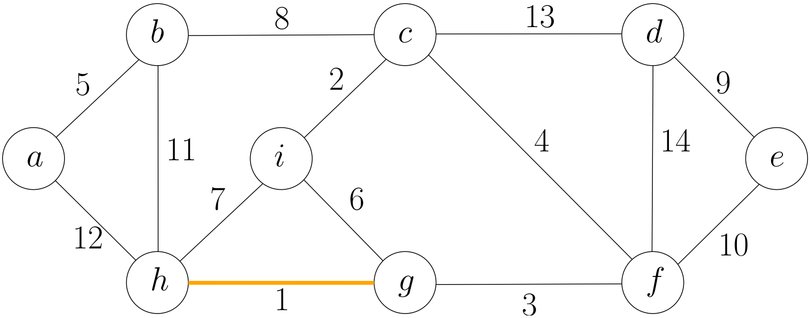
Sets: $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}, \{h\}, \{i\}$

Kruskal's Algorithm: Example



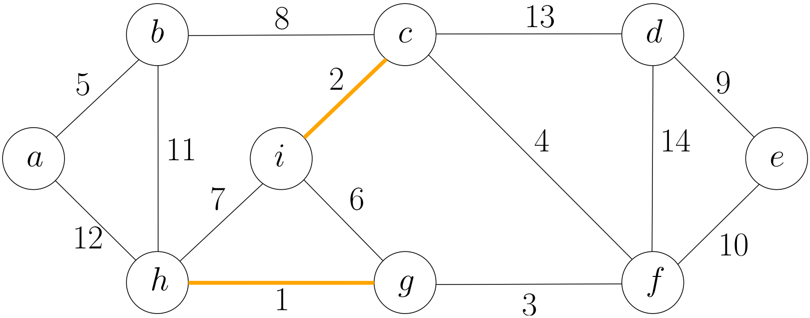
Sets: $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}, \{h\}, \{i\}$

Kruskal's Algorithm: Example



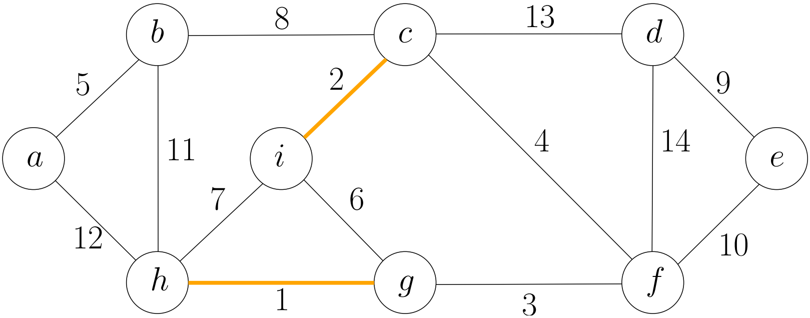
Sets: $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g, h\}, \{i\}$

Kruskal's Algorithm: Example



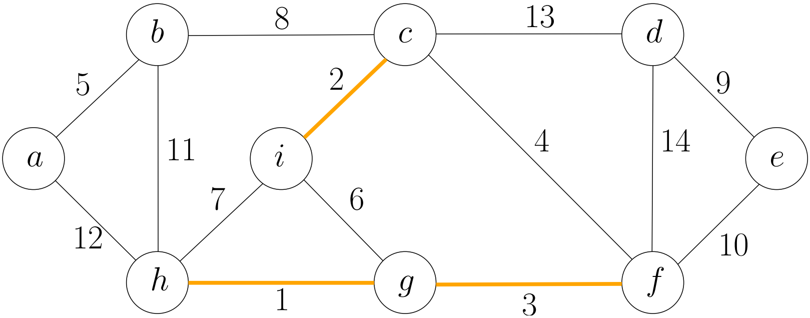
Sets: $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g, h\}, \{i\}$

Kruskal's Algorithm: Example



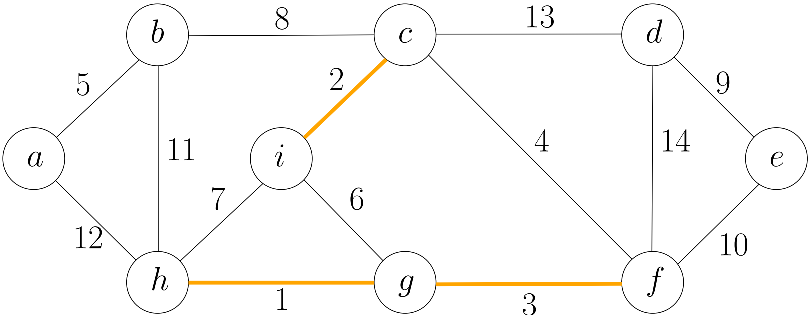
Sets: $\{a\}, \{b\}, \{c, i\}, \{d\}, \{e\}, \{f\}, \{g, h\}$

Kruskal's Algorithm: Example



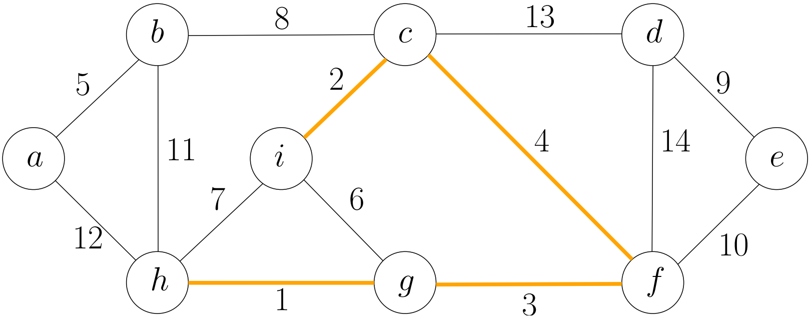
Sets: $\{a\}, \{b\}, \{c, i\}, \{d\}, \{e\}, \{f\}, \{g, h\}$

Kruskal's Algorithm: Example



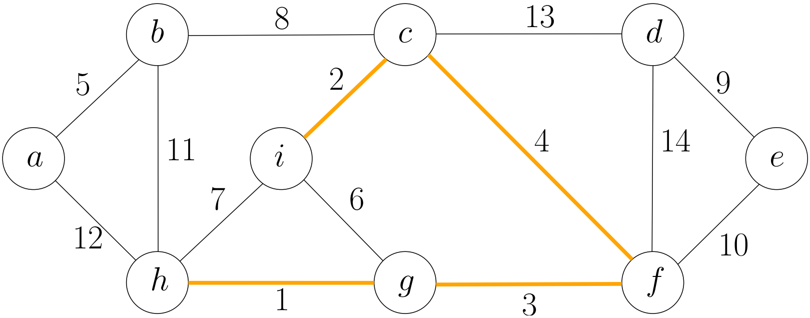
Sets: $\{a\}, \{b\}, \{c, i\}, \{d\}, \{e\}, \{f, g, h\}$

Kruskal's Algorithm: Example



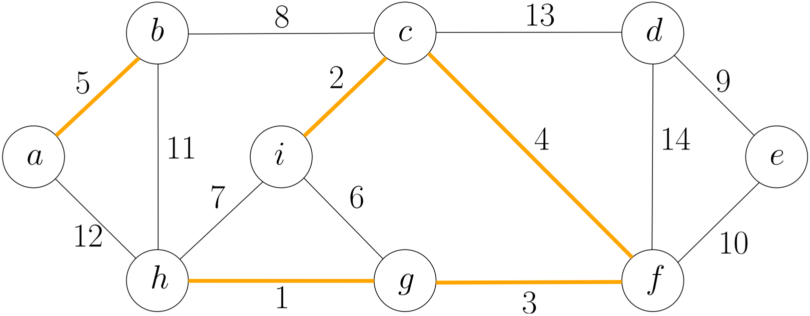
Sets: $\{a\}, \{b\}, \{c, i\}, \{d\}, \{e\}, \{f, g, h\}$

Kruskal's Algorithm: Example



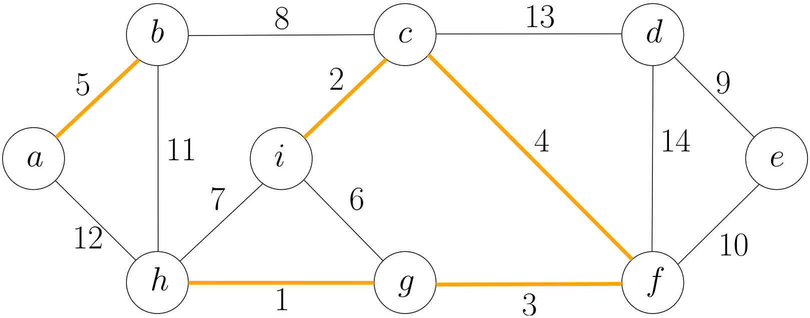
Sets: $\{a\}, \{b\}, \{c, i, f, g, h\}, \{d\}, \{e\}$

Kruskal's Algorithm: Example



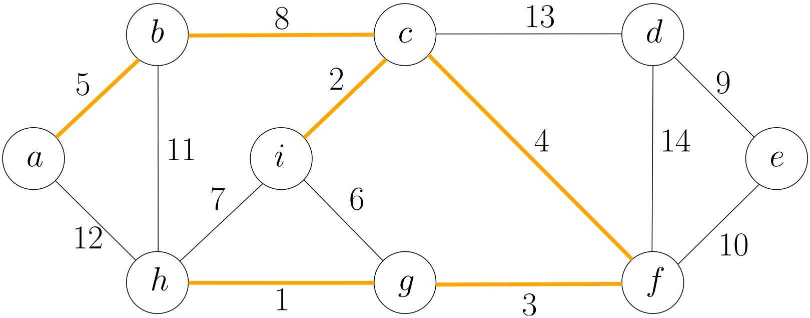
Sets: $\{a\}, \{b\}, \{c, i, f, g, h\}, \{d\}, \{e\}$

Kruskal's Algorithm: Example



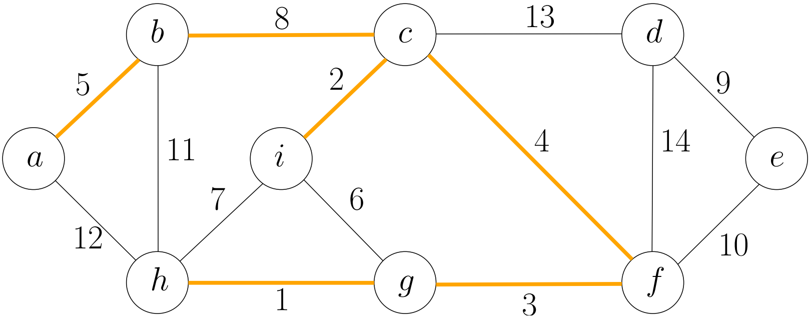
Sets: $\{a, b\}, \{c, i, f, g, h\}, \{d\}, \{e\}$

Kruskal's Algorithm: Example



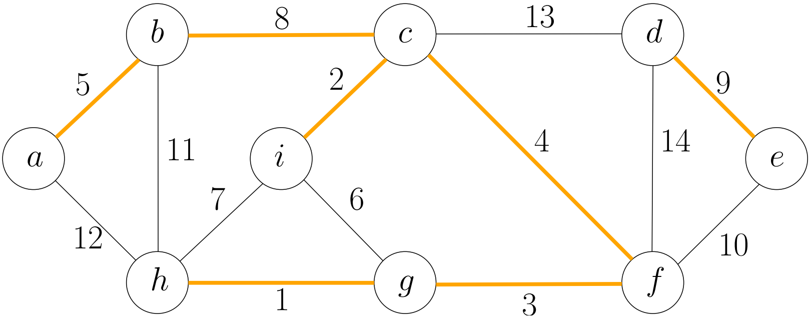
Sets: $\{a, b\}, \{c, i, f, g, h\}, \{d\}, \{e\}$

Kruskal's Algorithm: Example



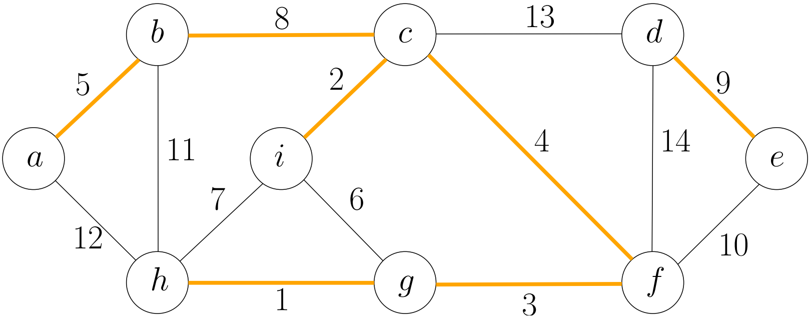
Sets: $\{a, b, c, i, f, g, h\}, \{d\}, \{e\}$

Kruskal's Algorithm: Example



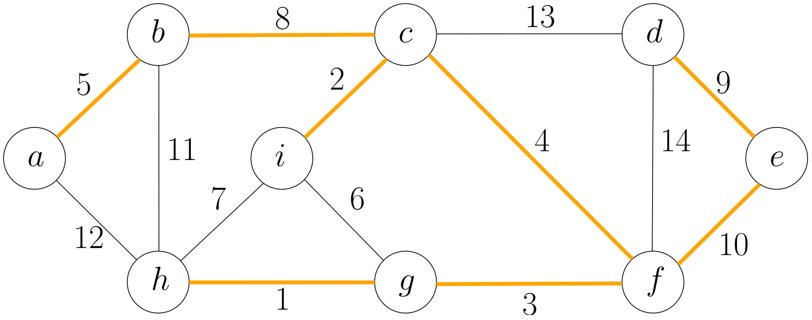
Sets: $\{a, b, c, i, f, g, h\}, \{d\}, \{e\}$

Kruskal's Algorithm: Example



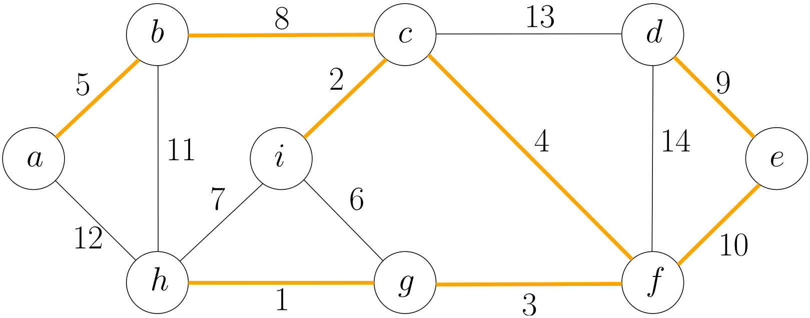
Sets: $\{a, b, c, i, f, g, h\}, \{d, e\}$

Kruskal's Algorithm: Example



Sets: $\{a, b, c, i, f, g, h\}, \{d, e\}$

Kruskal's Algorithm: Example



Sets: $\{a, b, c, i, f, g, h, d, e\}$

Kruskal's Algorithm: Efficient Implementation of Greedy Algorithm

MST-Kruskal(G, w)

```
F ← ∅
𝒮 ← {{v} : v ∈ V}
sort the edges of E in non-decreasing order of weights w
for each edge (u, v) ∈ E in the order do
    S_u ← the set in 𝒮 containing u
    S_v ← the set in 𝒮 containing v
    if S_u ≠ S_v then
        F ← F ∪ {(u, v)}
        𝒮 ← 𝒮 \ {S_u} \ {S_v} ∪ {S_u ∪ S_v}
return (V, F)
```

Running Time of Kruskal's Algorithm

MST-Kruskal(G, w)

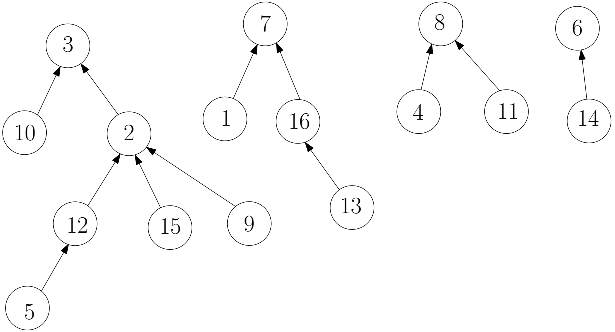
1: $F \leftarrow \emptyset$
2: $\mathcal{S} \leftarrow \{\{v\} : v \in V\}$
3: sort the edges of E in non-decreasing order of weights w
4: **for** each edge $(u, v) \in E$ in the order **do**
5: $\quad S_u \leftarrow$ the set in $\mathcal{S}$ containing u
6: $\quad S_v \leftarrow$ the set in $\mathcal{S}$ containing v
7: $\quad$ **if** $S_u \neq S_v$ **then**
8: $\quad\quad F \leftarrow F \cup \{(u, v)\}$
9: $\quad\quad \mathcal{S} \leftarrow \mathcal{S} \setminus \{S_u\} \setminus \{S_v\} \cup \{S_u \cup S_v\}$
10: **return** (V, F)

Use union-find data structure to support ❷, ❺, ❻, ❼, ❾.

Union-Find Data Structure

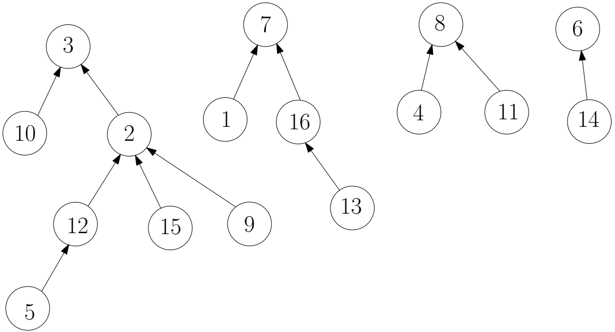
- V: ground set
- We need to maintain a partition of V and support following operations:
  - Check if u and v are in the same set of the partition
  - Merge two sets in partition

- $V = \{1, 2, 3, \cdots, 16\}$
- Partition: $\{2, 3, 5, 9, 10, 12, 15\}, \{1, 7, 13, 16\}, \{4, 8, 11\}, \{6, 14\}$

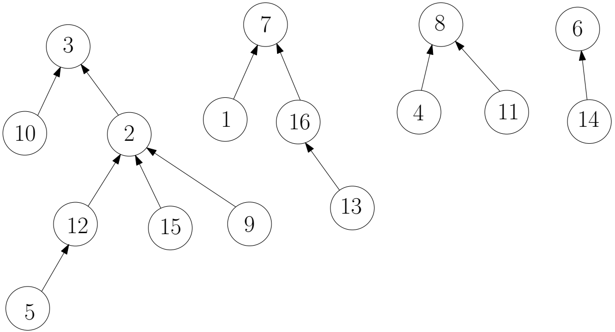


- $par[i]$: parent of i, ($par[i] = \perp$ if i is a root).

Union-Find Data Structure

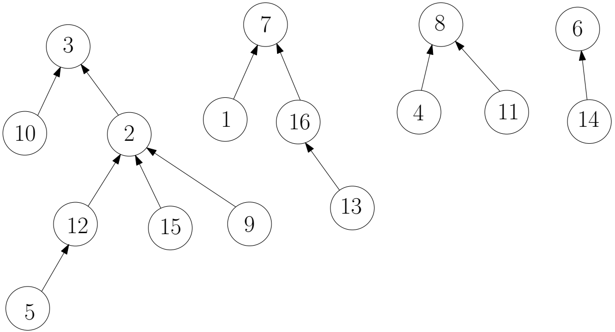


Union-Find Data Structure



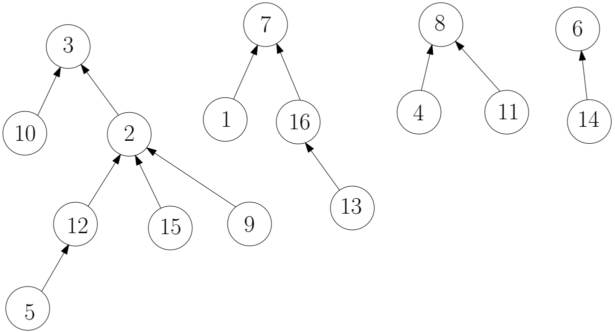
- Q: how can we check if u and v are in the same set?

Union-Find Data Structure



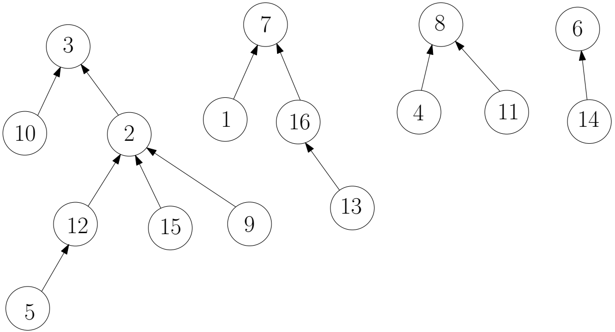
- Q: how can we check if u and v are in the same set?
- A: Check if root$(u) =$ root(v).

Union-Find Data Structure



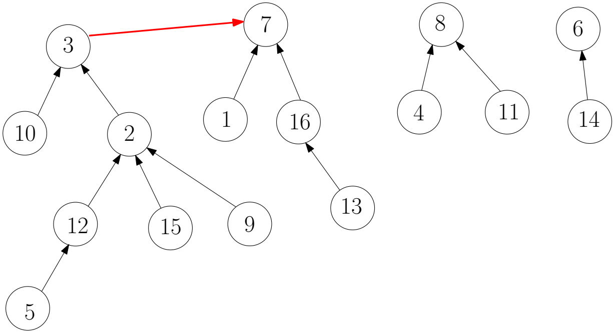
- Q: how can we check if u and v are in the same set?
- A: Check if $\text{root}(u) = \text{root}(v)$.
- $\text{root}(u)$: the root of the tree containing u

Union-Find Data Structure



- Q: how can we check if u and v are in the same set?
- A: Check if root(u) = root(v).
- root(u): the root of the tree containing u
- Merge the trees with root r and r': $par[r] \leftarrow r'$.

Union-Find Data Structure



- Q: how can we check if u and v are in the same set?
- A: Check if root(u) = root(v).
- root(u): the root of the tree containing u
- Merge the trees with root r and r': $par[r] \leftarrow r'$.

Union-Find Data Structure

root(v)

1: **if** $par[v] = \perp$ **then**
2:     **return** v
3: **else**
4:     **return** root($par[v]$)

Union-Find Data Structure

root(v)

1: **if** $par[v] = \perp$ **then**
2:     **return** v
3: **else**
4:     **return** root($par[v]$)

- Problem: the tree might too deep; running time might be large

Union-Find Data Structure

root(v)

if $par[v] = \perp$ **then**
  return v
else
  return root($par[v]$)

- Problem: the tree might too deep; running time might be large
- Improvement: all vertices in the path directly point to the root, saving time in the future.

Union-Find Data Structure

root(v)

```
if par[v] = ⊥ then
    return v
else
    return root(par[v])
```

root(v)

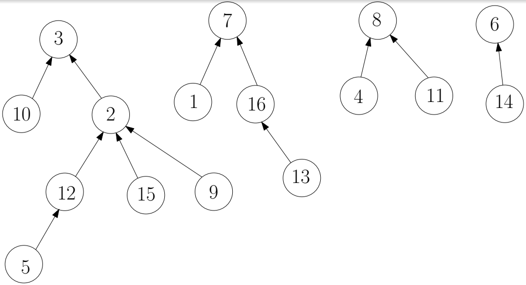
```
if par[v] = ⊥ then
    return v
else
    par[v] ← root(par[v])
return par[v]
```

- Problem: the tree might too deep; running time might be large
- Improvement: all vertices in the path directly point to the root, saving time in the future.

Union-Find Data Structure

root(v)

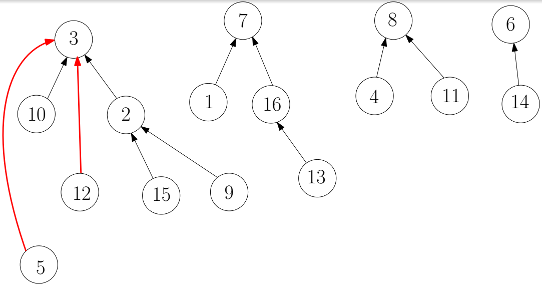
if $par[v] = \perp$ **then**
    return v
else
    $par[v] \leftarrow$ root($par[v]$)
    return $par[v]$



Union-Find Data Structure

root(v)

if $par[v] = \perp$ **then**
    return v
else
    $par[v] \leftarrow \text{root}(par[v])$
    return $par[v]$



MST-Kruskal(G, w)

$F \leftarrow \emptyset$
$\mathcal{S} \leftarrow \{\{v\} : v \in V\}$
sort the edges of E in non-decreasing order of weights w
for each edge $(u, v) \in E$ in the order **do**
    $S_u \leftarrow$ the set in $\mathcal{S}$ containing u
    $S_v \leftarrow$ the set in $\mathcal{S}$ containing v
    if $S_u \neq S_v$ **then**
        $F \leftarrow F \cup \{(u, v)\}$
        $\mathcal{S} \leftarrow \mathcal{S} \setminus \{S_u\} \setminus \{S_v\} \cup \{S_u \cup S_v\}$
return (V, F)

MST-Kruskal(G, w)

$F \leftarrow \emptyset$
for every $v \in V$ **do**: $par[v] \leftarrow \perp$
sort the edges of E in non-decreasing order of weights w
for each edge $(u, v) \in E$ in the order **do**
    $u' \leftarrow \text{root}(u)$
    $v' \leftarrow \text{root}(v)$
    if $u' \neq v'$ **then**
        $F \leftarrow F \cup \{(u, v)\}$
        $par[u'] \leftarrow v'$
return (V, F)

MST-Kruskal(G, w)

1: $F \leftarrow \emptyset$
2: **for** every $v \in V$ **do**: $par[v] \leftarrow \perp$
3: sort the edges of E in non-decreasing order of weights w
4: **for** each edge $(u, v) \in E$ in the order **do**
5: $\quad u' \leftarrow \text{root}(u)$
6: $\quad v' \leftarrow \text{root}(v)$
7: $\quad$ **if** $u' \neq v'$ **then**
8: $\quad\quad F \leftarrow F \cup \{(u, v)\}$
9: $\quad\quad par[u'] \leftarrow v'$
10: **return** (V, F)

- 2, 5, 6, 7, 9 takes time $O(m\alpha(n))$
- $\alpha(n)$ is very slow-growing: $\alpha(n) \leq 4$ for $n \leq 10^{80}$.

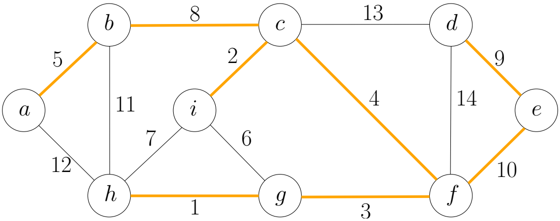
MST-Kruskal(G, w)

```
1: F ← ∅
2: for every v ∈ V do: par[v] ← ⊥
3: sort the edges of E in non-decreasing order of weights w
4: for each edge (u, v) ∈ E in the order do
5:     u' ← root(u)
6:     v' ← root(v)
7:     if u' ≠ v' then
8:         F ← F ∪ {(u, v)}
9:         par[u'] ← v'
10: return (V, F)
```

- 2,5,6,7,9 takes time $O(m\alpha(n))$
- $\alpha(n)$ is very slow-growing: $\alpha(n) \leq 4$ for $n \leq 10^{80}$.
- Running time $=$ time for 3 $= O(m \lg n)$.

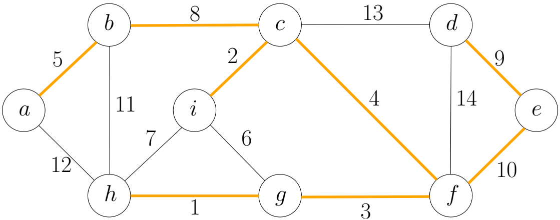
Assumption Assume all edge weights are different.

Lemma An edge $e \in E$ is not in the MST, if and only if there is cycle C in G in which e is the heaviest edge.



Assumption Assume all edge weights are different.

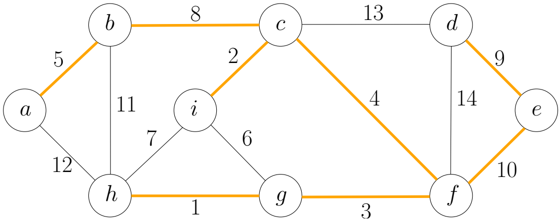
Lemma An edge $e \in E$ is not in the MST, if and only if there is cycle C in G in which e is the heaviest edge.



- (i, g) is not in the MST because of cycle (i, c, f, g)

Assumption Assume all edge weights are different.

Lemma An edge $e \in E$ is not in the MST, if and only if there is cycle C in G in which e is the heaviest edge.



- (i, g) is not in the MST because of cycle (i, c, f, g)
- (e, f) is in the MST because no such cycle exists

Outline

Two Methods to Build a MST

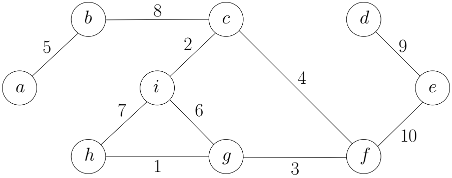
1. Start from $F \leftarrow \emptyset$, and add edges to F one by one until we obtain a spanning tree

Two Methods to Build a MST

1. Start from $F \leftarrow \emptyset$, and add edges to F one by one until we obtain a spanning tree
2. Start from $F \leftarrow E$, and remove edges from F one by one until we obtain a spanning tree

Two Methods to Build a MST

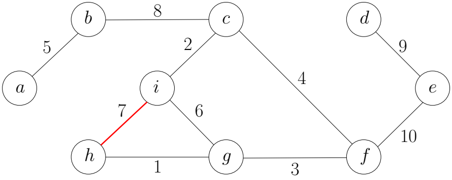
1. Start from $F \leftarrow \emptyset$, and add edges to F one by one until we obtain a spanning tree
2. Start from $F \leftarrow E$, and remove edges from F one by one until we obtain a spanning tree



Q: Which edge can be safely excluded from the MST?

Two Methods to Build a MST

1. Start from $F \leftarrow \emptyset$, and add edges to F one by one until we obtain a spanning tree
2. Start from $F \leftarrow E$, and remove edges from F one by one until we obtain a spanning tree

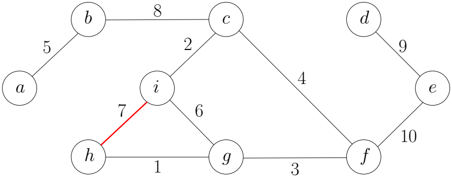


Q: Which edge can be safely excluded from the MST?

A: The heaviest non-bridge edge.

Two Methods to Build a MST

1. Start from $F \leftarrow \emptyset$, and add edges to F one by one until we obtain a spanning tree
2. Start from $F \leftarrow E$, and remove edges from F one by one until we obtain a spanning tree



Q: Which edge can be safely excluded from the MST?

A: The heaviest non-bridge edge.

Def. A bridge is an edge whose removal disconnects the graph.

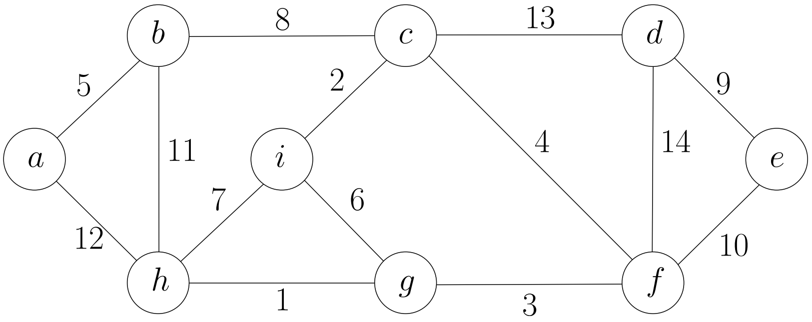
Lemma It is safe to exclude the heaviest non-bridge edge: there is a MST that does not contain the heaviest non-bridge edge.

Reverse Kruskal's Algorithm

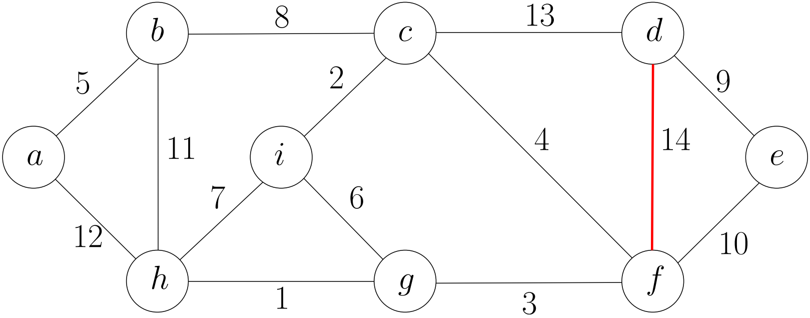
MST-Greedy(G, w)

```
F ← E
sort E in non-increasing order of weights
for every e in this order do
    if (V, F \ {e}) is connected then
        F ← F \ {e}
return (V, F)
```

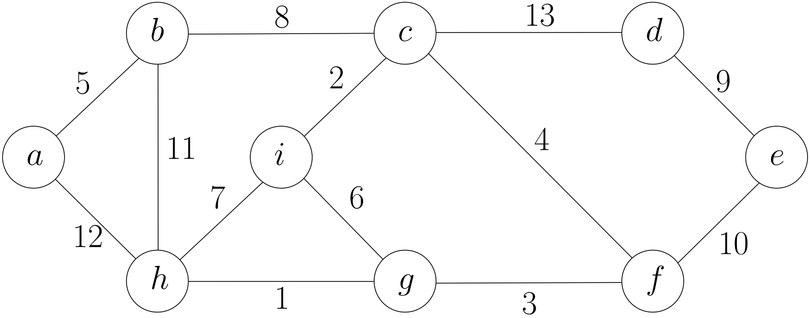
Reverse Kruskal's Algorithm: Example



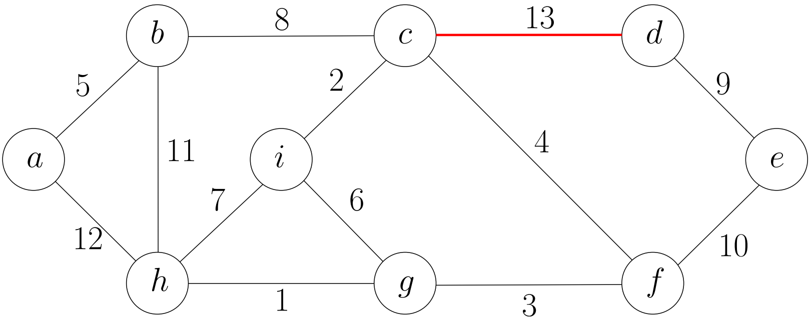
Reverse Kruskal's Algorithm: Example



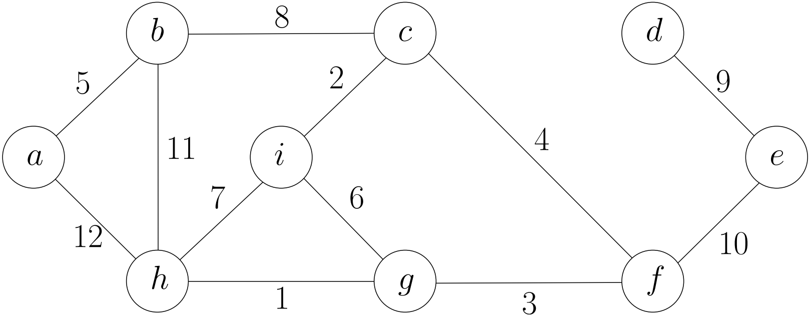
Reverse Kruskal's Algorithm: Example



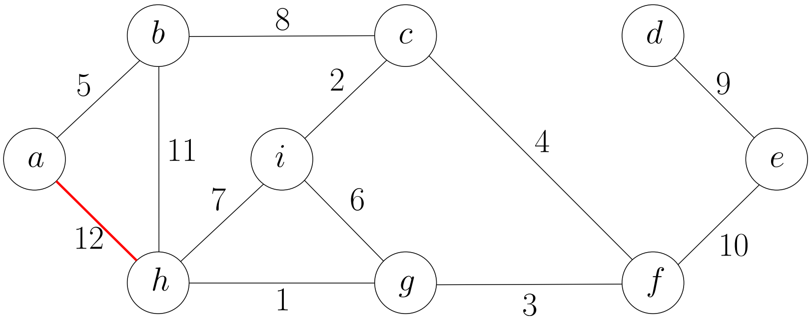
Reverse Kruskal's Algorithm: Example



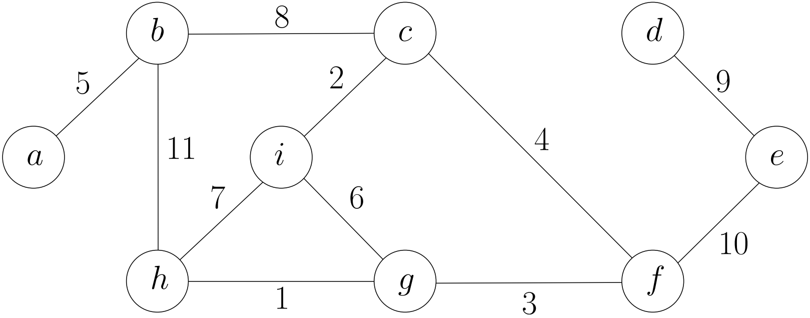
Reverse Kruskal's Algorithm: Example



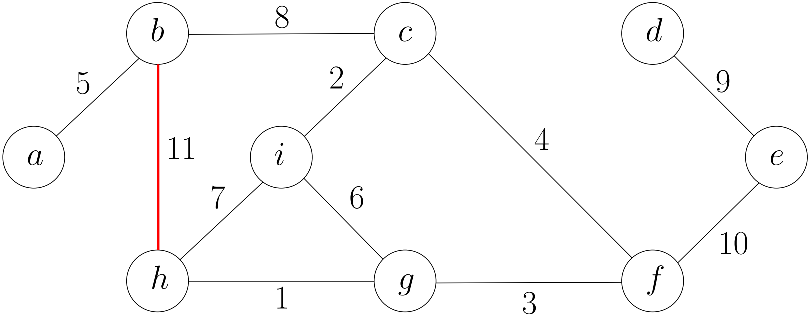
Reverse Kruskal's Algorithm: Example



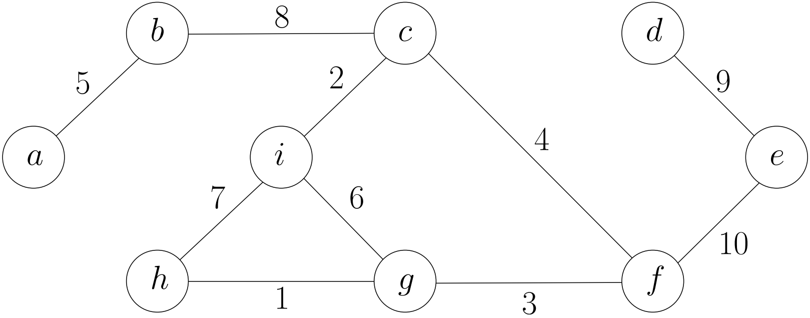
Reverse Kruskal's Algorithm: Example



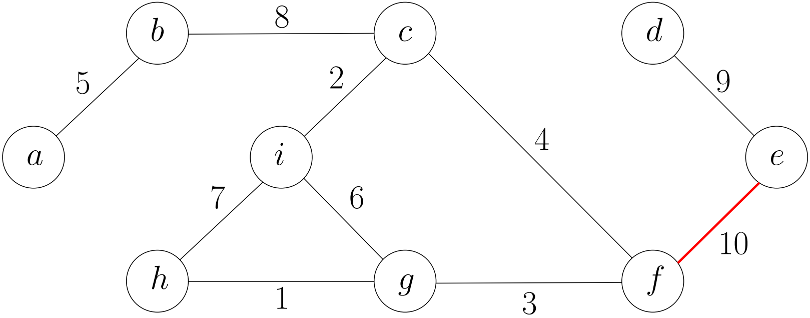
Reverse Kruskal's Algorithm: Example



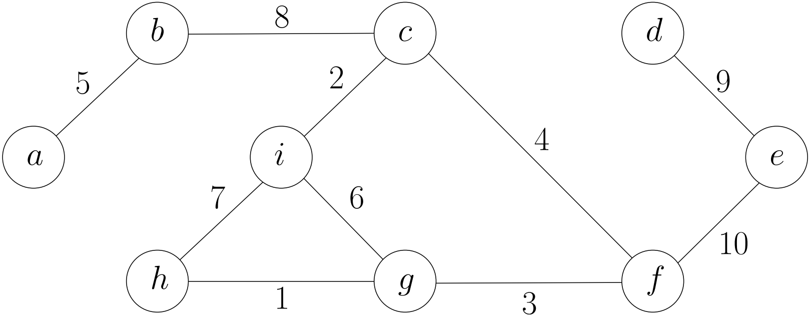
Reverse Kruskal's Algorithm: Example



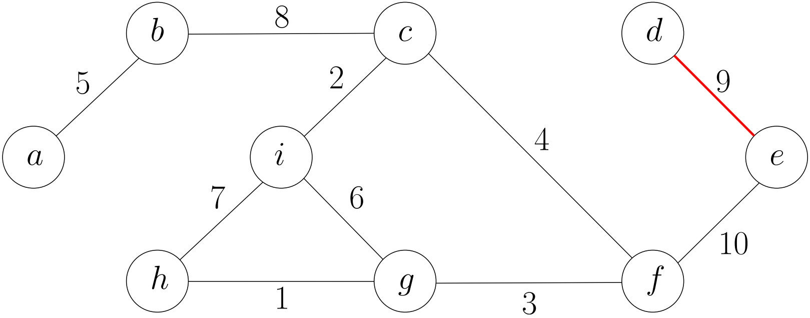
Reverse Kruskal's Algorithm: Example



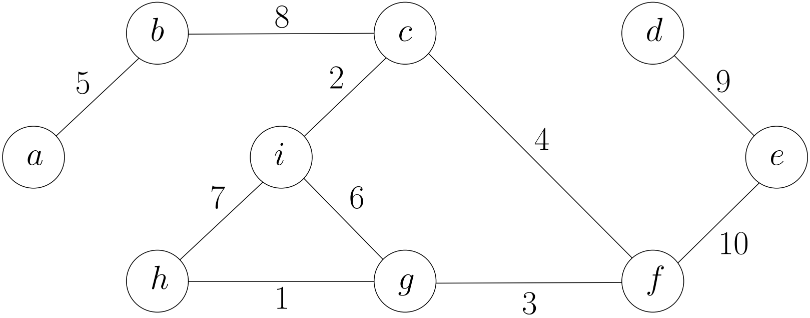
Reverse Kruskal's Algorithm: Example



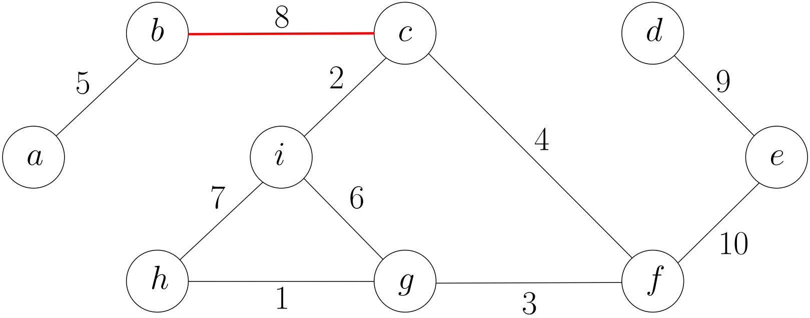
Reverse Kruskal's Algorithm: Example



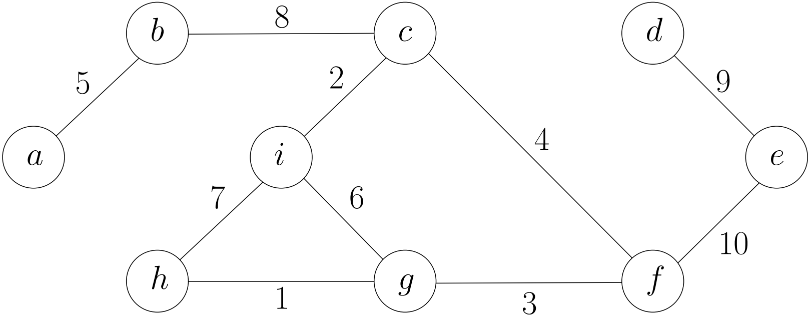
Reverse Kruskal's Algorithm: Example



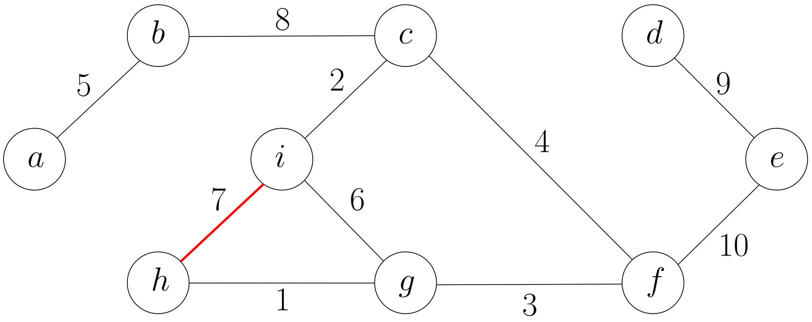
Reverse Kruskal's Algorithm: Example



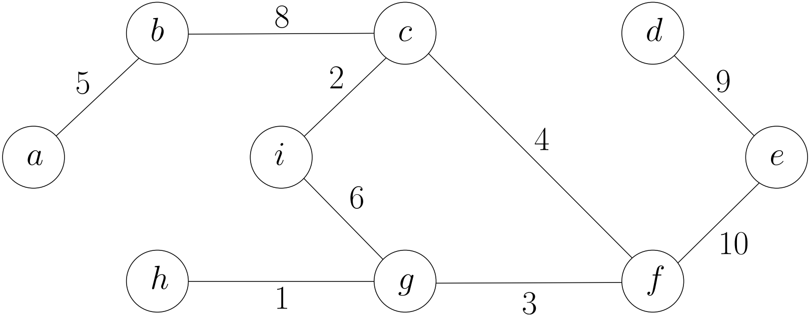
Reverse Kruskal's Algorithm: Example



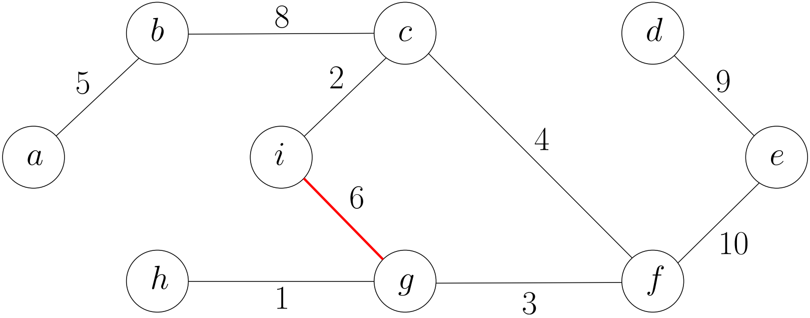
Reverse Kruskal's Algorithm: Example



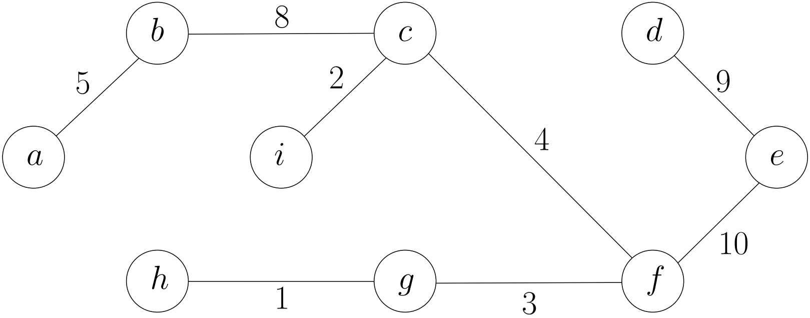
Reverse Kruskal's Algorithm: Example



Reverse Kruskal's Algorithm: Example



Reverse Kruskal's Algorithm: Example

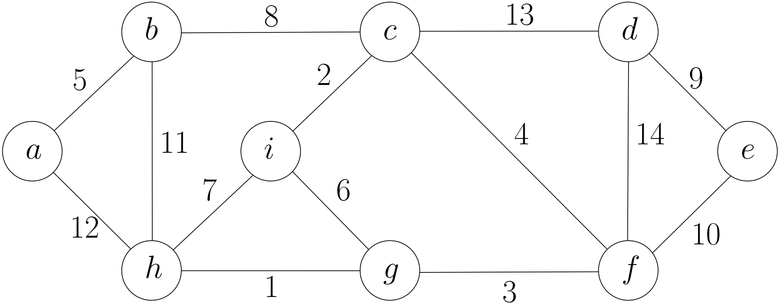


Outline

1. Minimum Spanning Tree
   - Kruskal's Algorithm
   - Reverse-Kruskal's Algorithm
   - Prim's Algorithm
2. Single Source Shortest Paths
   - Dijkstra's Algorithm
3. Shortest Paths in Graphs with Negative Weights
   - Bellman-Ford Algorithm
4. All-Pair Shortest Paths and Floyd-Warshall

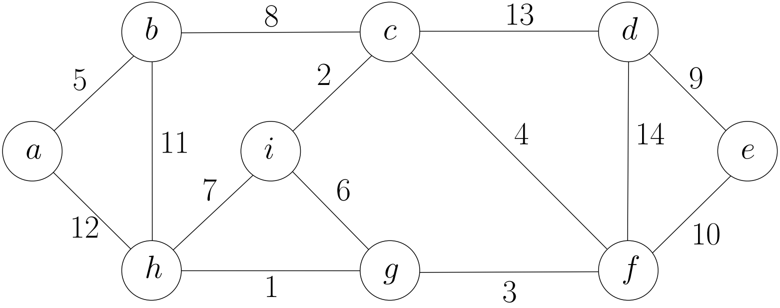
Design Greedy Strategy for MST

- Recall the greedy strategy for Kruskal's algorithm: choose the edge with the smallest weight.



Design Greedy Strategy for MST

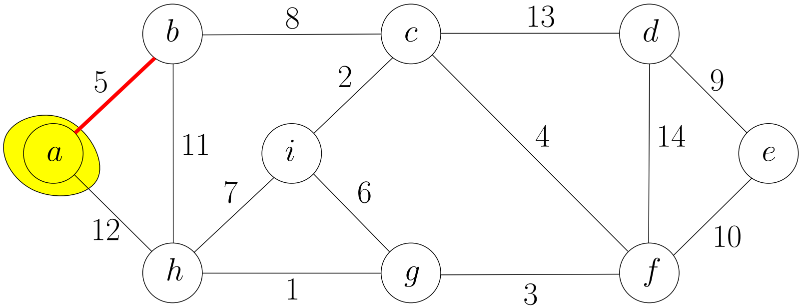
- Recall the greedy strategy for Kruskal's algorithm: choose the edge with the smallest weight.



- Greedy strategy for Prim's algorithm: choose the lightest edge incident to a.

Design Greedy Strategy for MST

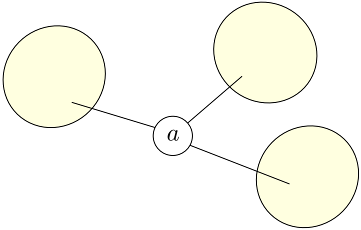
- Recall the greedy strategy for Kruskal's algorithm: choose the edge with the smallest weight.



- Greedy strategy for Prim's algorithm: choose the lightest edge incident to a.

Lemma It is safe to include the lightest edge incident to a.

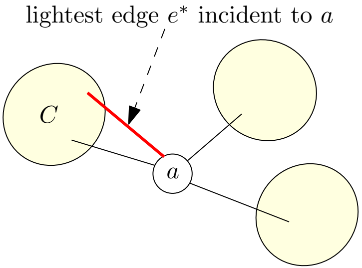
Lemma It is safe to include the lightest edge incident to a.



Proof.

- Let T be a MST
- Consider all components obtained by removing a from T

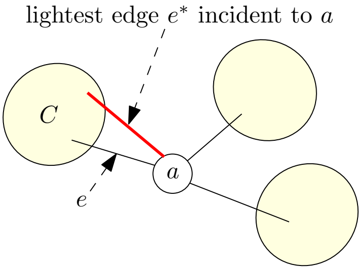
Lemma It is safe to include the lightest edge incident to a.



Proof.

- Let T be a MST
- Consider all components obtained by removing a from T
- Let e^* be the lightest edge incident to a and e^* connects a to component C

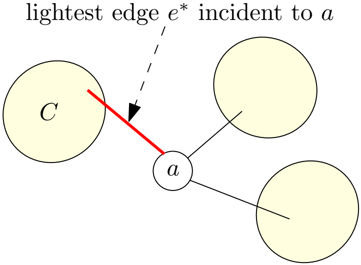
Lemma It is safe to include the lightest edge incident to a.



Proof.

- Let T be a MST
- Consider all components obtained by removing a from T
- Let e^* be the lightest edge incident to a and e^* connects a to component C
- Let e be the edge in T connecting a to C

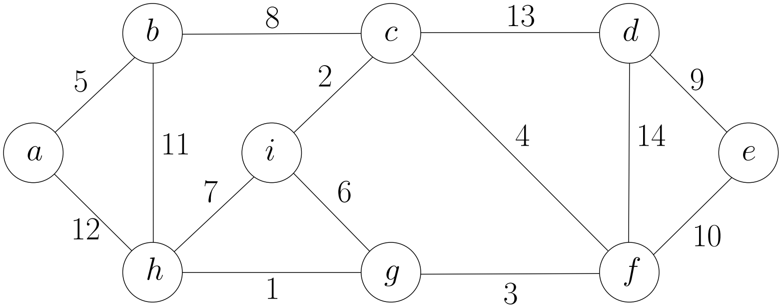
Lemma It is safe to include the lightest edge incident to a.



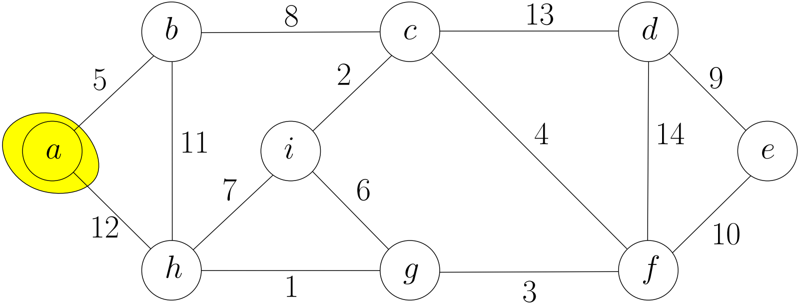
Proof.

- Let T be a MST
- Consider all components obtained by removing a from T
- Let e^* be the lightest edge incident to a and e^* connects a to component C
- Let e be the edge in T connecting a to C
- $T' = T \setminus \{e\} \cup \{e^*\}$ is a spanning tree with $w(T') \leq w(T)$ $\square$

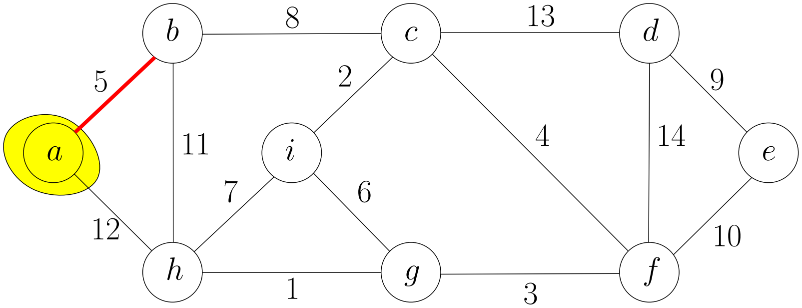
Prim's Algorithm: Example



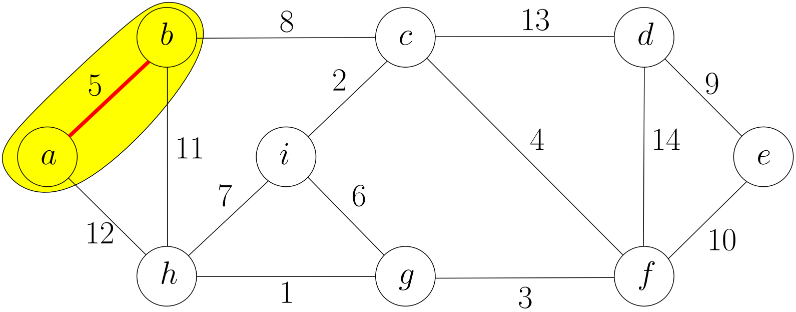
Prim's Algorithm: Example



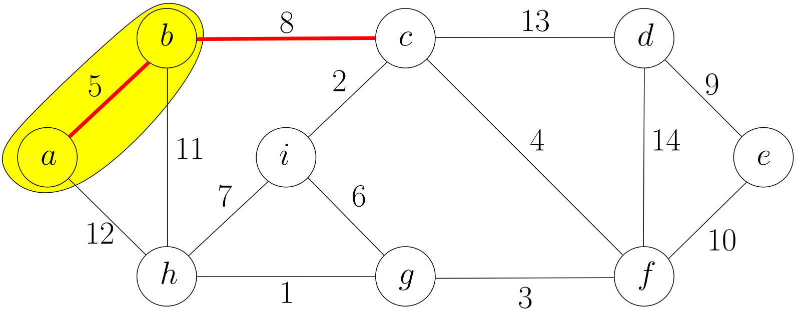
Prim's Algorithm: Example



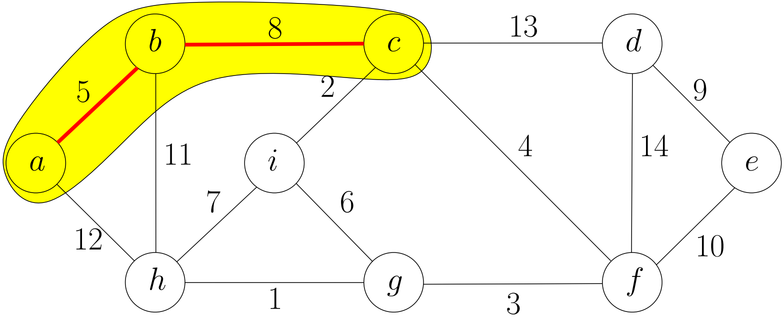
Prim's Algorithm: Example



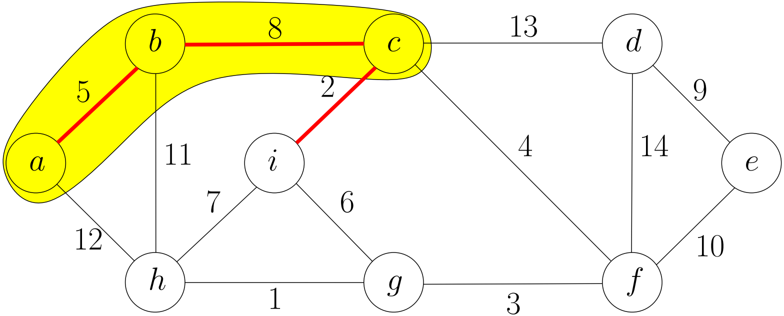
Prim's Algorithm: Example



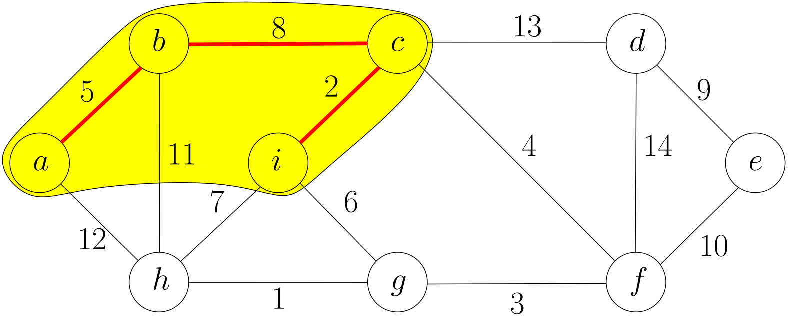
Prim's Algorithm: Example



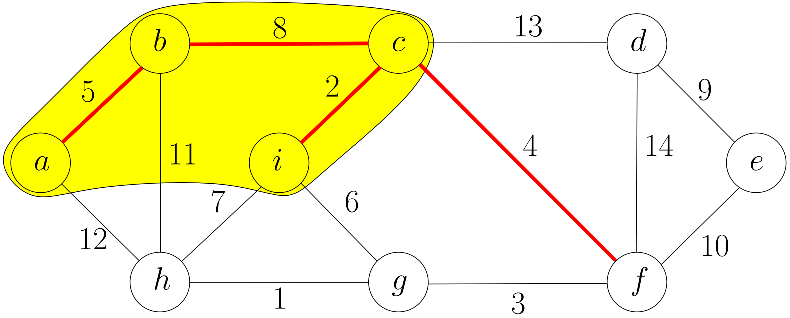
Prim's Algorithm: Example



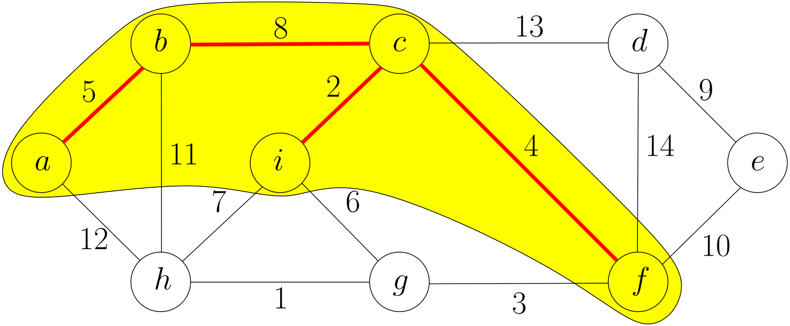
Prim's Algorithm: Example



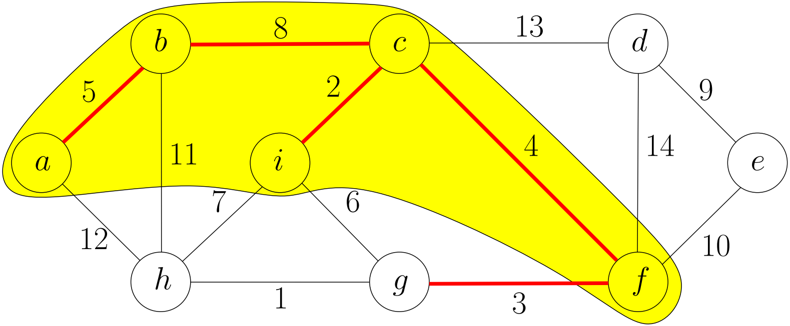
Prim's Algorithm: Example



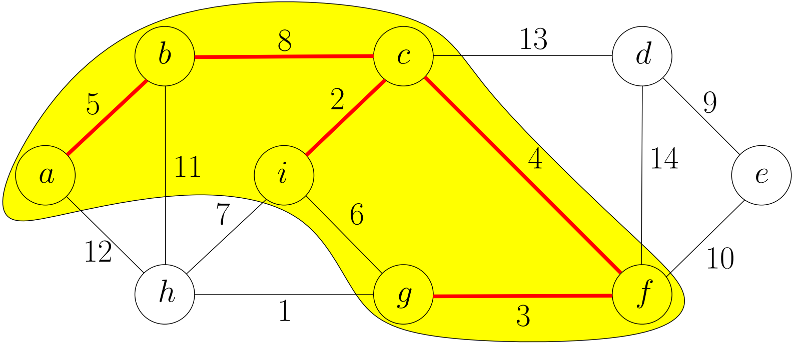
Prim's Algorithm: Example



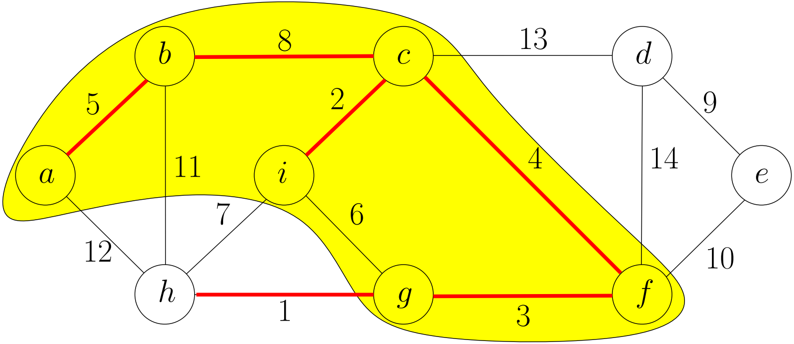
Prim's Algorithm: Example



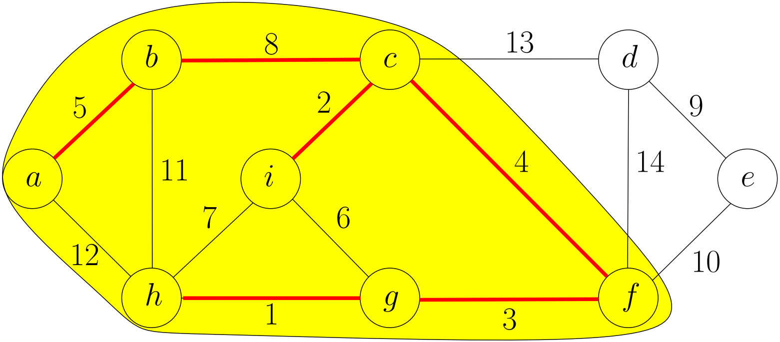
Prim's Algorithm: Example



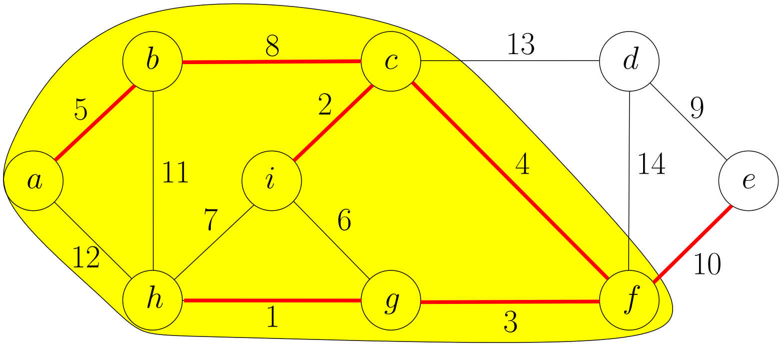
Prim's Algorithm: Example



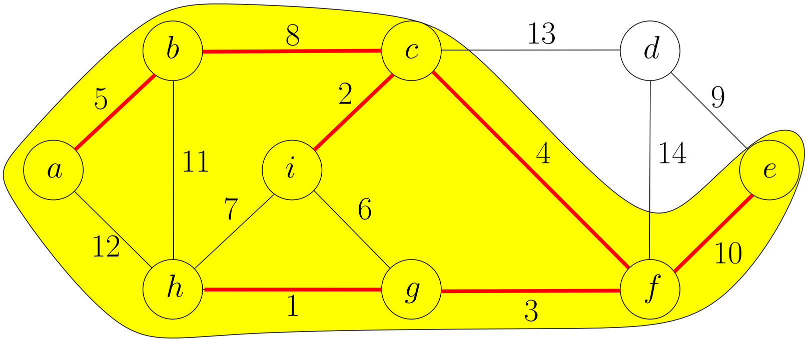
Prim's Algorithm: Example



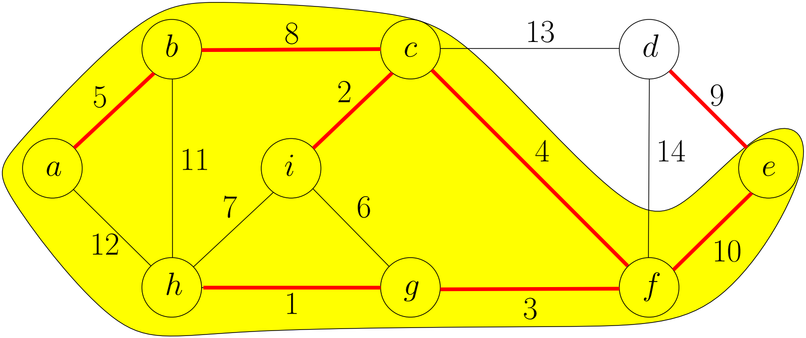
Prim's Algorithm: Example



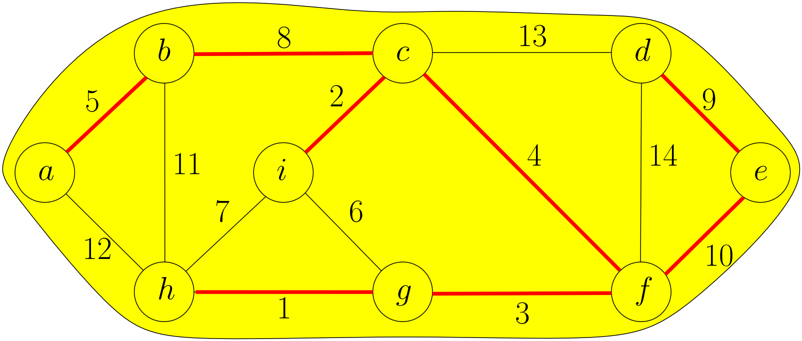
Prim's Algorithm: Example



Prim's Algorithm: Example



Prim's Algorithm: Example



Greedy Algorithm

MST-Greedy1(G, w)

```
S ← {s}, where s is arbitrary vertex in V
F ← ∅
while S ≠ V do
    (u, v) ← lightest edge between S and V \ S,
                          where u ∈ S and v ∈ V \ S
    S ← S ∪ {v}
    F ← F ∪ {(u, v)}
return (V, F)
```

Greedy Algorithm

MST-Greedy1(G, w)

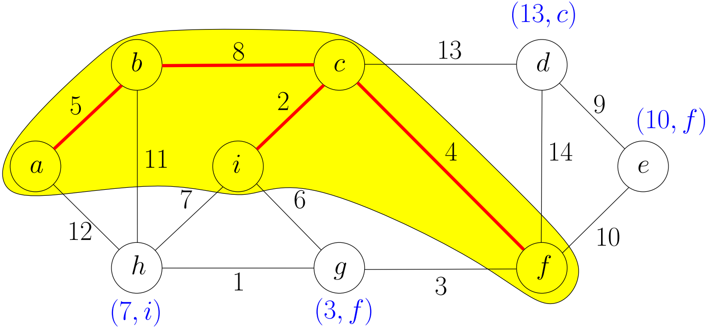
$S \leftarrow \{s\}$, where s is arbitrary vertex in V
$F \leftarrow \emptyset$
while $S \neq V$ **do**
    $(u, v) \leftarrow$ lightest edge between S and $V \setminus S$,
        where $u \in S$ and $v \in V \setminus S$
    $S \leftarrow S \cup \{v\}$
    $F \leftarrow F \cup \{(u, v)\}$
return (V, F)

- Running time of naive implementation: $O(nm)$

Prim's Algorithm: Efficient Implementation of Greedy Algorithm

For every $v \in V \setminus S$ maintain

- $d(v) = \min_{u \in S:(u,v) \in E} w(u,v)$:
  the weight of the lightest edge between v and S
- $\pi(v) = \arg\min_{u \in S:(u,v) \in E} w(u,v)$:
  $(\pi(v), v)$ is the lightest edge between v and S



Prim's Algorithm: Efficient Implementation of Greedy Algorithm

For every $v \in V \setminus S$ maintain

- $d(v) = \min_{u \in S:(u,v) \in E} w(u,v)$:
  the weight of the lightest edge between v and S
- $\pi(v) = \arg\min_{u \in S:(u,v) \in E} w(u,v)$:
  $(\pi(v), v)$ is the lightest edge between v and S

In every iteration

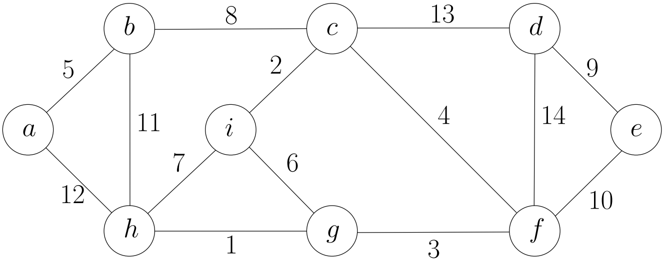
- Pick $u \in V \setminus S$ with the smallest $d(u)$ value
- Add $(\pi(u), u)$ to F
- Add u to S, update d and π values.

Prim's Algorithm

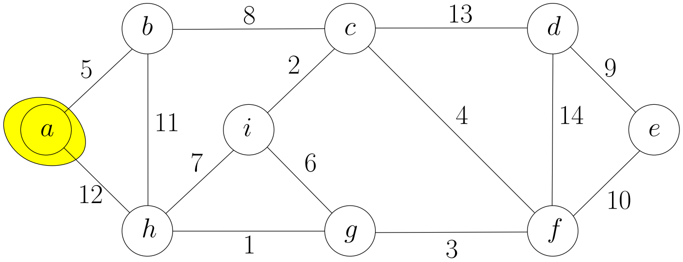
MST-Prim(G, w)

$s \leftarrow$ arbitrary vertex in G
$S \leftarrow \emptyset, d(s) \leftarrow 0$ and $d(v) \leftarrow \infty$ for every $v \in V \setminus \{s\}$
while $S \neq V$ **do**
    $u \leftarrow$ vertex in $V \setminus S$ with the minimum $d(u)$
    $S \leftarrow S \cup \{u\}$
    for each $v \in V \setminus S$ such that $(u, v) \in E$ **do**
        if $w(u, v) < d(v)$ **then**
            $d(v) \leftarrow w(u, v)$
            $\pi(v) \leftarrow u$
return $\{(u, \pi(u)) | u \in V \setminus \{s\}\}$

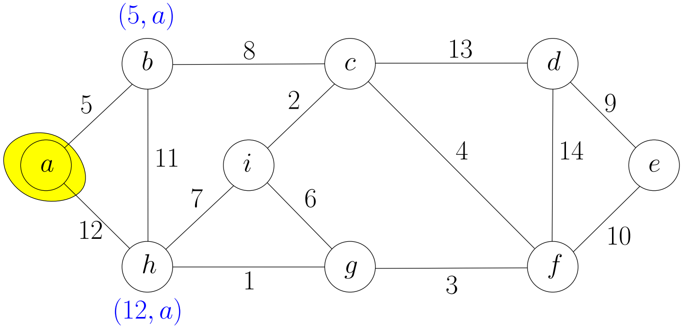
Example



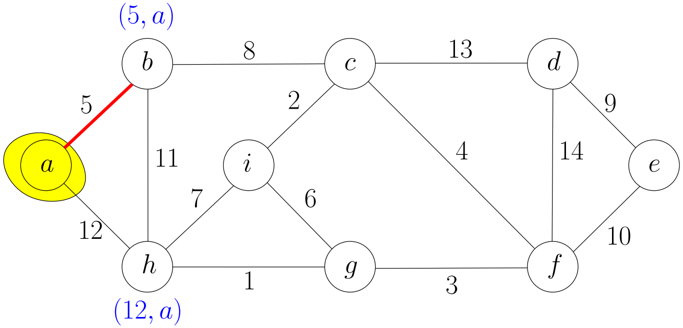
Example



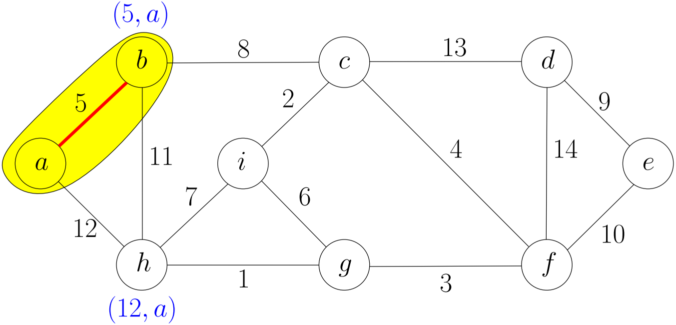
Example



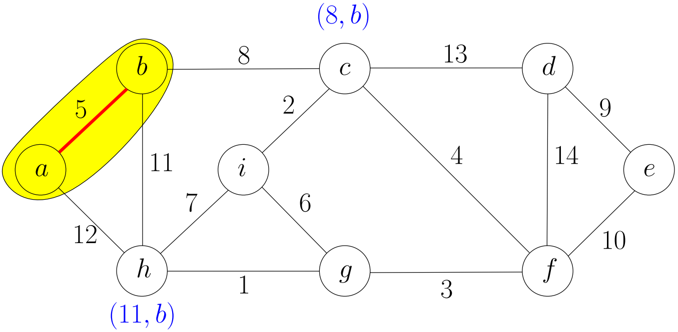
Example



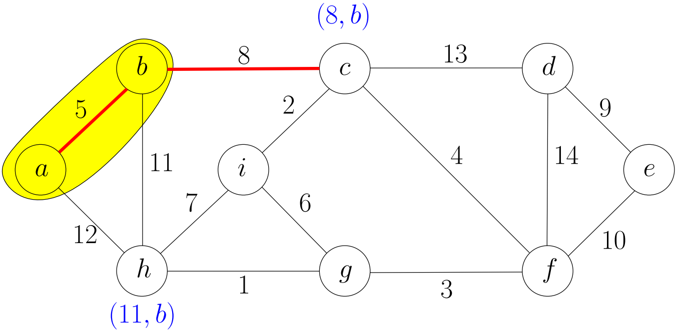
Example



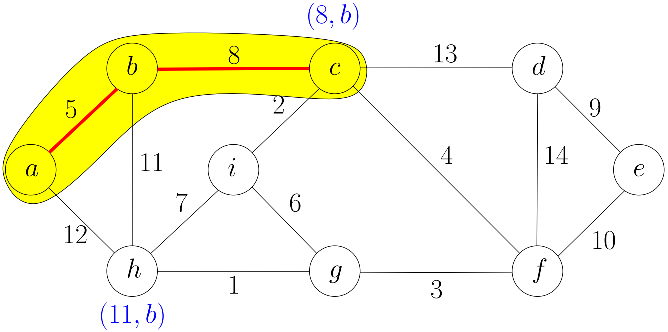
Example



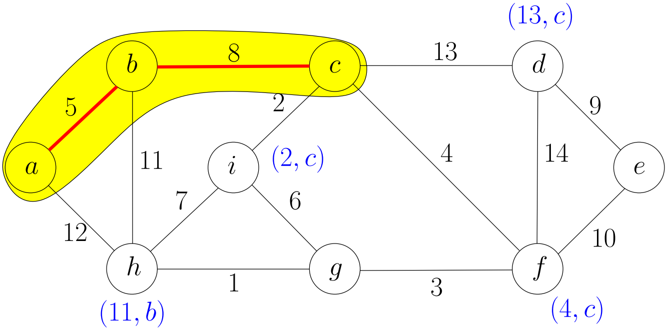
Example



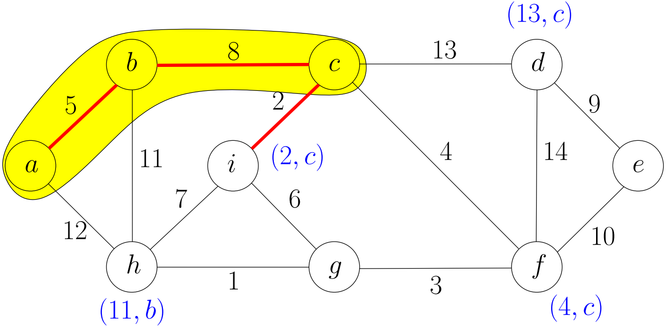
Example



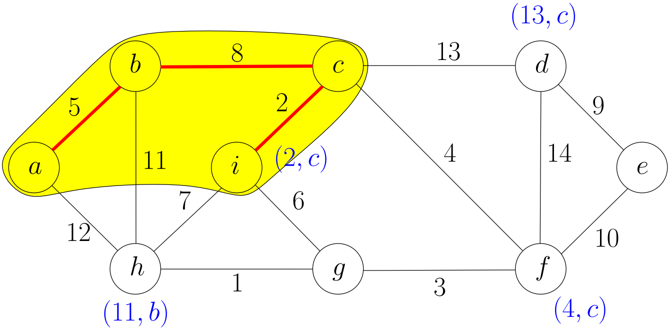
Example



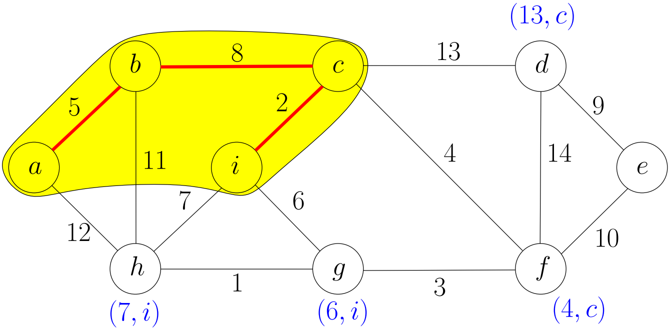
Example



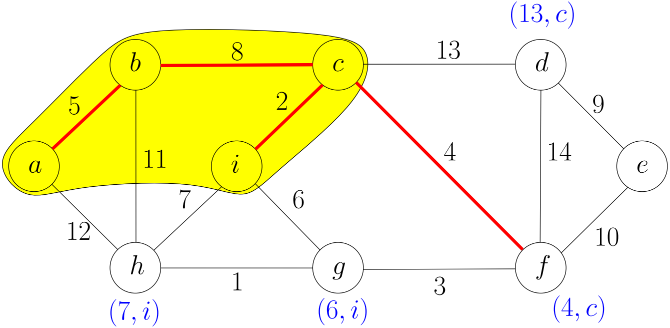
Example



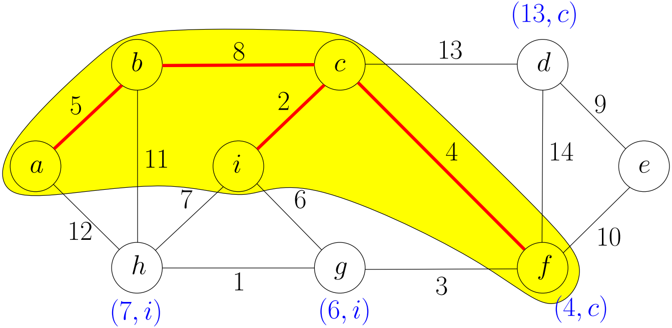
Example



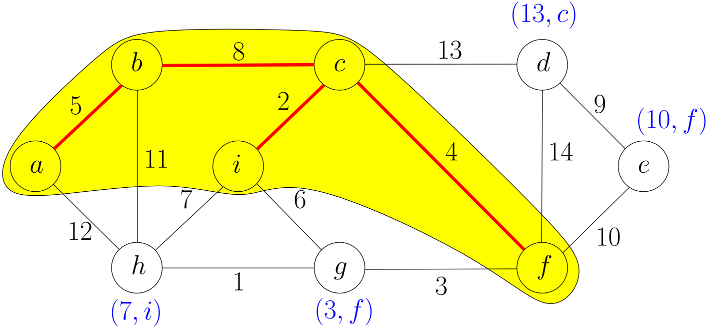
Example



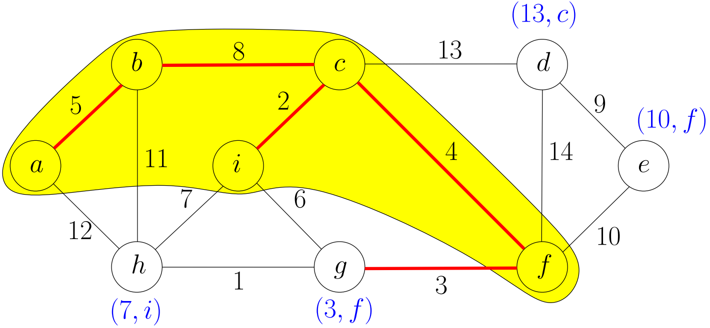
Example



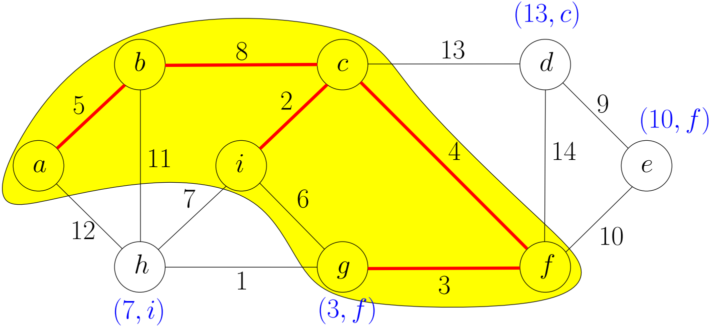
Example



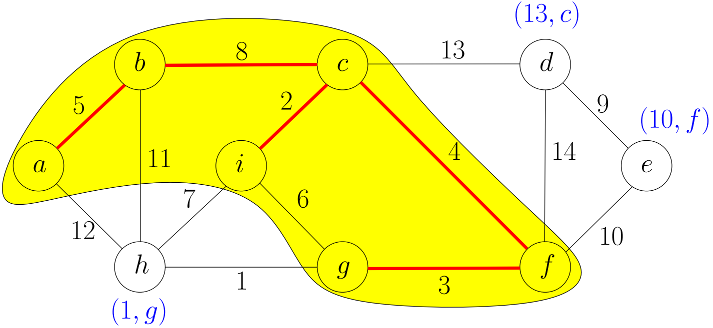
Example



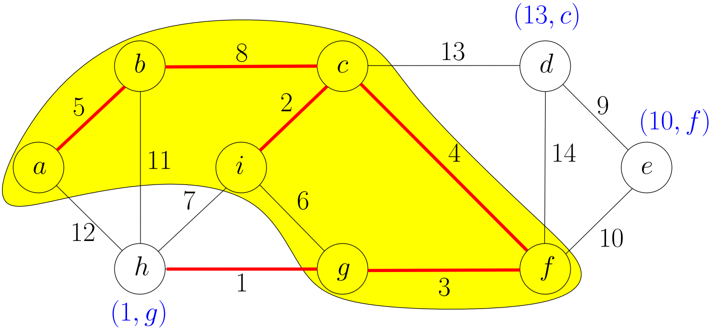
Example



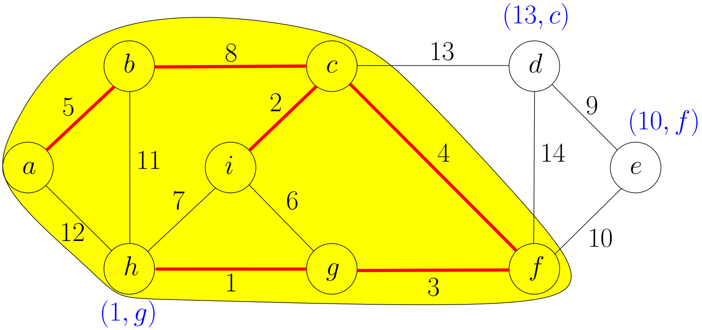
Example



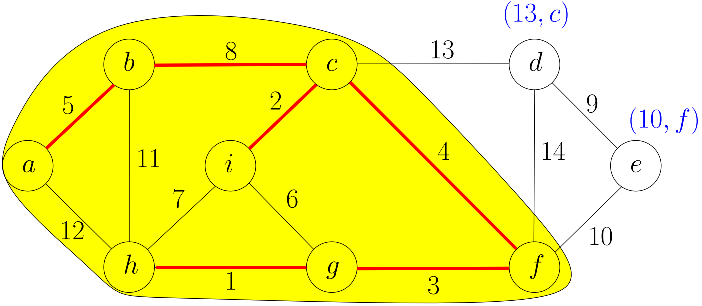
Example



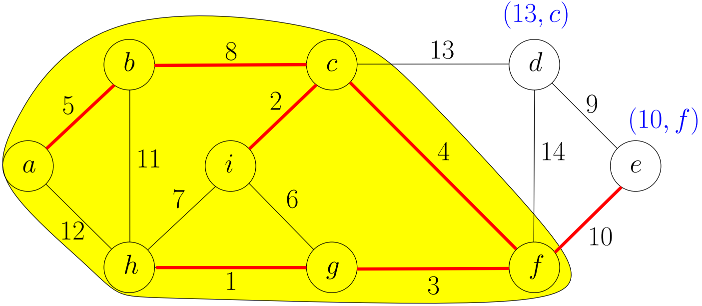
Example



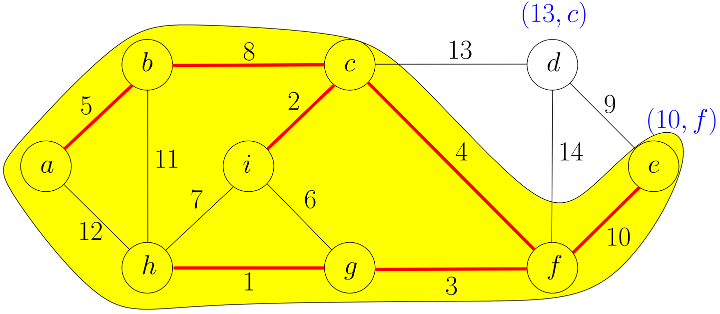
Example



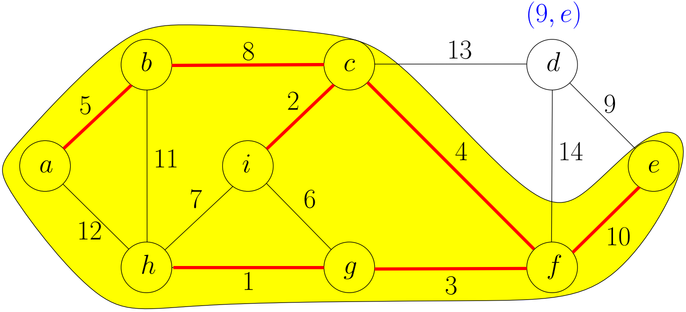
Example



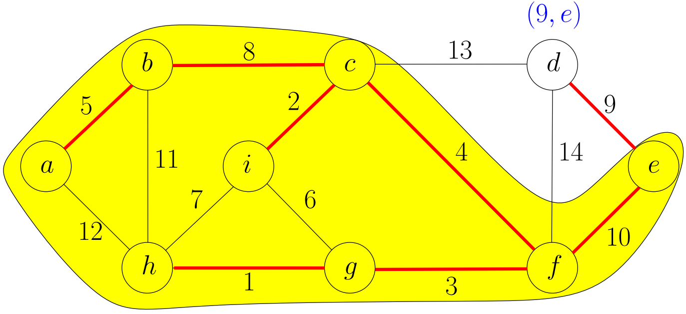
Example



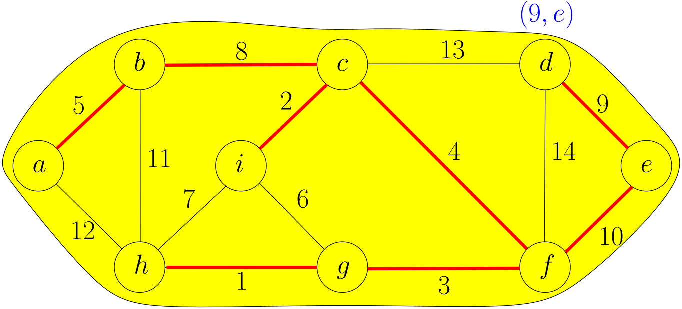
Example



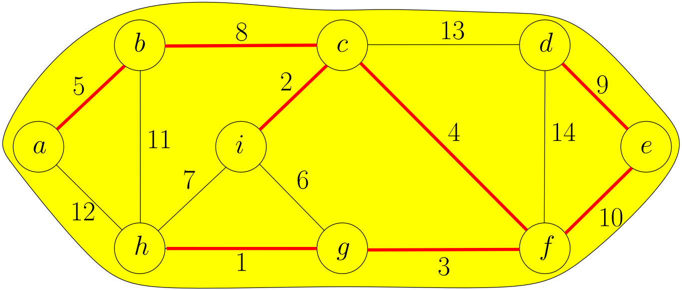
Example



Example



Example



Prim's Algorithm

For every $v \in V \setminus S$ maintain

- $d(v) = \min_{u \in S:(u,v) \in E} w(u,v)$:
  the weight of the lightest edge between v and S
- $\pi(v) = \arg\min_{u \in S:(u,v) \in E} w(u,v)$:
  $(\pi(v), v)$ is the lightest edge between v and S

In every iteration

- Pick $u \in V \setminus S$ with the smallest $d(u)$ value
- Add $(\pi(u), u)$ to F
- Add u to S, update d and π values.

Prim's Algorithm

For every $v \in V \setminus S$ maintain

- $d(v) = \min_{u \in S:(u,v) \in E} w(u,v)$:
  the weight of the lightest edge between v and S
- $\pi(v) = \arg\min_{u \in S:(u,v) \in E} w(u,v)$:
  $(\pi(v), v)$ is the lightest edge between v and S

In every iteration

- Pick $u \in V \setminus S$ with the smallest $d(u)$ value   extract_min
- Add $(\pi(u), u)$ to F
- Add u to S, update d and π values.   decrease_key

Use a priority queue to support the operations

Def. A priority queue is an abstract data structure that maintains a set U of elements, each with an associated key value, and supports the following operations:

- insert(v, key_value): insert an element v, whose associated key value is key_value.
- decrease_key(v, new_key_value): decrease the key value of an element v in queue to new_key_value
- extract_min$()$: return and remove the element in queue with the smallest key value
- $\cdots$

Prim's Algorithm

MST-Prim(G, w)

$s \leftarrow$ arbitrary vertex in G
$S \leftarrow \emptyset, d(s) \leftarrow 0$ and $d(v) \leftarrow \infty$ for every $v \in V \setminus \{s\}$

while $S \neq V$ **do**
    $u \leftarrow$ vertex in $V \setminus S$ with the minimum $d(u)$
    $S \leftarrow S \cup \{u\}$
    for each $v \in V \setminus S$ such that $(u, v) \in E$ **do**
        if $w(u, v) < d(v)$ **then**
            $d(v) \leftarrow w(u, v)$
            $\pi(v) \leftarrow u$
return $\big\{(u, \pi(u)) | u \in V \setminus \{s\}\big\}$

Prim's Algorithm Using Priority Queue

MST-Prim(G, w)

$s \leftarrow$ arbitrary vertex in G
$S \leftarrow \emptyset, d(s) \leftarrow 0$ and $d(v) \leftarrow \infty$ for every $v \in V \setminus \{s\}$
$Q \leftarrow$ empty queue, for each $v \in V$: Q.insert$(v, d(v))$
while $S \neq V$ **do**
    $u \leftarrow Q$.extract_min()
    $S \leftarrow S \cup \{u\}$
    for each $v \in V \setminus S$ such that $(u, v) \in E$ **do**
        if $w(u, v) < d(v)$ **then**
            $d(v) \leftarrow w(u, v)$, Q.decrease_key$(v, d(v))$
            $\pi(v) \leftarrow u$
return $\big\{(u, \pi(u)) | u \in V \setminus \{s\}\big\}$

Running Time of Prim's Algorithm Using Priority Queue

$O(n)\times$ (time for extract_min) $+ O(m)\times$ (time for decrease_key)

Running Time of Prim's Algorithm Using Priority Queue

$O(n)\times$ (time for extract_min) $+$ $O(m)\times$ (time for decrease_key)

concrete DS	extract_min	decrease_key	overall time
heap	$O(\log n)$	$O(\log n)$	$O(m \log n)$
Fibonacci heap	$O(\log n)$	$O(1)$	$O(n \log n + m)$

Running Time of Prim's Algorithm Using Priority Queue

$O(n)\times$ (time for extract_min) $+ O(m)\times$ (time for decrease_key)

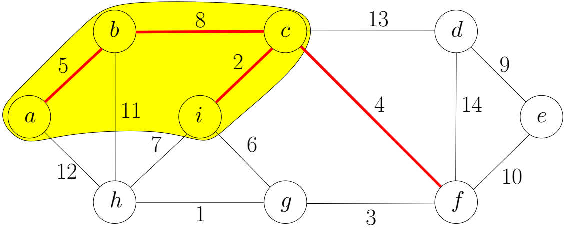
concrete DS	extract_min	decrease_key	overall time
heap	$O(\log n)$	$O(\log n)$	$O(m \log n)$
Fibonacci heap	$O(\log n)$	$O(1)$	$O(n \log n + m)$

Assumption Assume all edge weights are different.

Lemma (u, v) is in MST, if and only if there exists a cut $(U, V \setminus U)$, such that (u, v) is the lightest edge between U and $V \setminus U$.

Assumption Assume all edge weights are different.

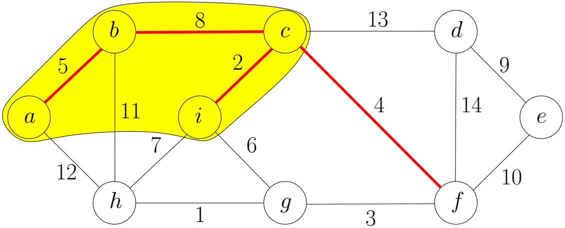
Lemma (u, v) is in MST, if and only if there exists a cut $(U, V \setminus U)$, such that (u, v) is the lightest edge between U and $V \setminus U$.



- (c, f) is in MST because of cut $\big(\{a, b, c, i\}, V \setminus \{a, b, c, i\}\big)$

Assumption Assume all edge weights are different.

Lemma (u, v) is in MST, if and only if there exists a cut $(U, V \setminus U)$, such that (u, v) is the lightest edge between U and $V \setminus U$.



- (c, f) is in MST because of cut $\big(\{a, b, c, i\}, V \setminus \{a, b, c, i\}\big)$
- (i, g) is not in MST because no such cut exists

"Evidence" for $e \in$ MST or $e \notin$ MST

Assumption Assume all edge weights are different.

- $e \in$ MST $\leftrightarrow$ there is a cut in which e is the lightest edge
- $e \notin$ MST $\leftrightarrow$ there is a cycle in which e is the heaviest edge

"Evidence" for $e \in$ MST or $e \notin$ MST

Assumption Assume all edge weights are different.

- $e \in$ MST $\leftrightarrow$ there is a cut in which e is the lightest edge
- $e \notin$ MST $\leftrightarrow$ there is a cycle in which e is the heaviest edge

Exactly one of the following is true:

- There is a cut in which e is the lightest edge
- There is a cycle in which e is the heaviest edge

"Evidence" for $e \in$ MST or $e \notin$ MST

Assumption Assume all edge weights are different.

- $e \in$ MST $\leftrightarrow$ there is a cut in which e is the lightest edge
- $e \notin$ MST $\leftrightarrow$ there is a cycle in which e is the heaviest edge

Exactly one of the following is true:

- There is a cut in which e is the lightest edge
- There is a cycle in which e is the heaviest edge

Thus, the minimum spanning tree is unique with assumption.

Outline

s-t Shortest Paths

Input: (directed or undirected) graph $G = (V, E)$, $s, t \in V$

$w : E \to \mathbb{R}_{\geq 0}$

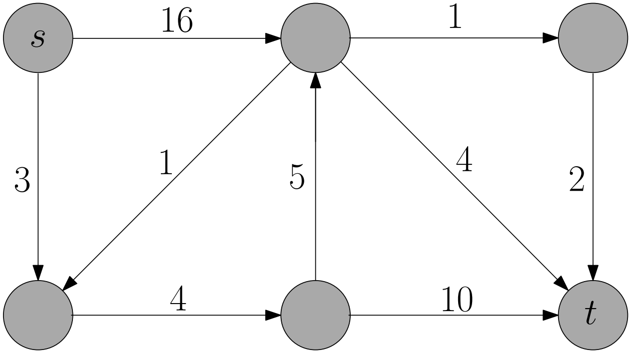
Output: shortest path from s to t

s-t Shortest Paths

Input: (directed or undirected) graph $G = (V, E)$, $s, t \in V$

$w : E \to \mathbb{R}_{\geq 0}$

Output: shortest path from s to t

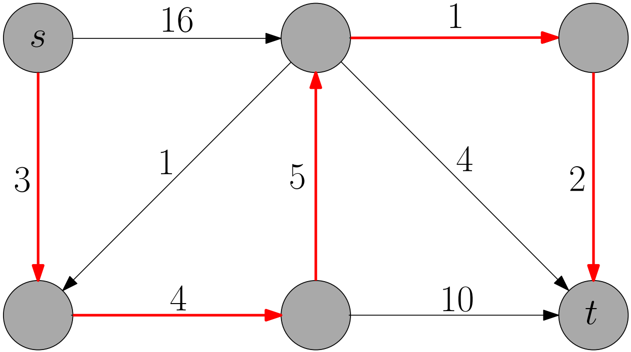


s-t Shortest Paths

Input: (directed or undirected) graph $G = (V, E)$, $s, t \in V$

$w : E \to \mathbb{R}_{\geq 0}$

Output: shortest path from s to t



Single Source Shortest Paths

Input: (directed or undirected) graph $G = (V, E)$, $s \in V$

$w : E \to \mathbb{R}_{\geq 0}$

Output: shortest paths from s to all other vertices $v \in V$

Single Source Shortest Paths

Input: (directed or undirected) graph $G = (V, E)$, $s \in V$

$w : E \to \mathbb{R}_{\geq 0}$

Output: shortest paths from s to all other vertices $v \in V$

Reason for Considering Single Source Shortest Paths Problem

- We do not know how to solve s-t shortest path problem more efficiently than solving single source shortest path problem

Single Source Shortest Paths

Input: (directed or undirected) graph $G = (V, E)$, $s \in V$

$w : E \to \mathbb{R}_{\geq 0}$

Output: shortest paths from s to all other vertices $v \in V$

Reason for Considering Single Source Shortest Paths Problem

- We do not know how to solve s-t shortest path problem more efficiently than solving single source shortest path problem

- Shortest paths in directed graphs is more general than in undirected graphs: we can replace every undirected edge with two anti-parallel edges of the same weight

Single Source Shortest Paths

Input: (directed or undirected) graph $G = (V, E)$, $s \in V$

$w : E \to \mathbb{R}_{\geq 0}$

Output: shortest paths from s to all other vertices $v \in V$

Reason for Considering Single Source Shortest Paths Problem

- We do not know how to solve s-t shortest path problem more efficiently than solving single source shortest path problem

- Shortest paths in directed graphs is more general than in undirected graphs: we can replace every undirected edge with two anti-parallel edges of the same weight

Single Source Shortest Paths

Input: directed graph $G = (V, E)$, $s \in V$

$w : E \to \mathbb{R}_{\geq 0}$

Output: shortest paths from s to all other vertices $v \in V$

Reason for Considering Single Source Shortest Paths Problem

- We do not know how to solve s-t shortest path problem more efficiently than solving single source shortest path problem

- Shortest paths in directed graphs is more general than in undirected graphs: we can replace every undirected edge with two anti-parallel edges of the same weight

- Shortest path from s to v may contain $\Omega(n)$ edges

- Shortest path from s to v may contain $\Omega(n)$ edges
- There are $\Omega(n)$ different vertices v

- Shortest path from s to v may contain $\Omega(n)$ edges
- There are $\Omega(n)$ different vertices v
- Thus, printing out all shortest paths may take time $\Omega(n^2)$

- Shortest path from s to v may contain $\Omega(n)$ edges
- There are $\Omega(n)$ different vertices v
- Thus, printing out all shortest paths may take time $\Omega(n^2)$
- Not acceptable if graph is sparse

Shortest Path Tree

Shortest Path Tree

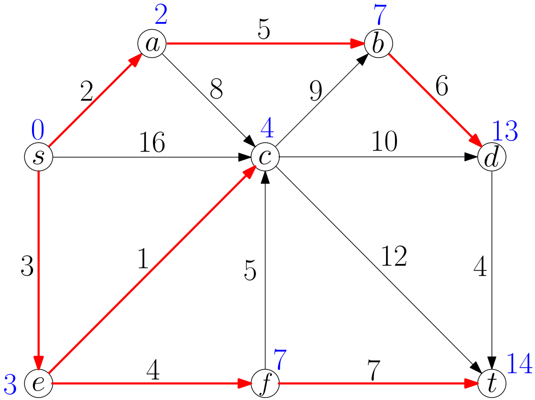
- $O(n)$-size data structure to represent all shortest paths

Shortest Path Tree

- $O(n)$-size data structure to represent all shortest paths
- For every vertex v, we only need to remember the parent of v: second-to-last vertex in the shortest path from s to v (why?)

Shortest Path Tree

- $O(n)$-size data structure to represent all shortest paths
- For every vertex v, we only need to remember the parent of v: second-to-last vertex in the shortest path from s to v (why?)



Single Source Shortest Paths

Input: directed graph $G = (V, E)$, $s \in V$

$w : E \to \mathbb{R}_{\geq 0}$

Output: $\pi(v), v \in V \setminus s$: the parent of v

$d(v), v \in V \setminus s$: the length of shortest path from s to v

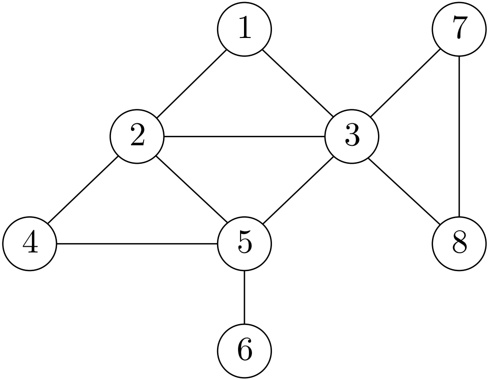
Q: How to compute shortest paths from s when all edges have weight 1?

Q: How to compute shortest paths from s when all edges have weight 1?

A: Breadth first search (BFS) from source s

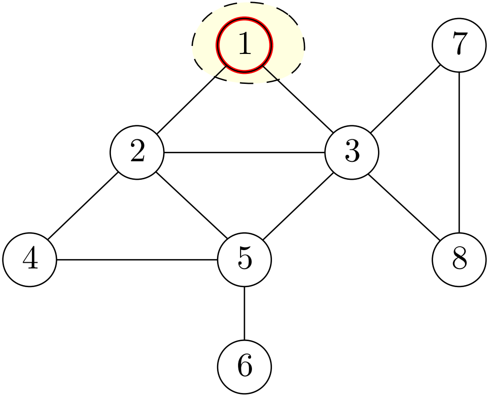
Q: How to compute shortest paths from s when all edges have weight 1?

A: Breadth first search (BFS) from source s



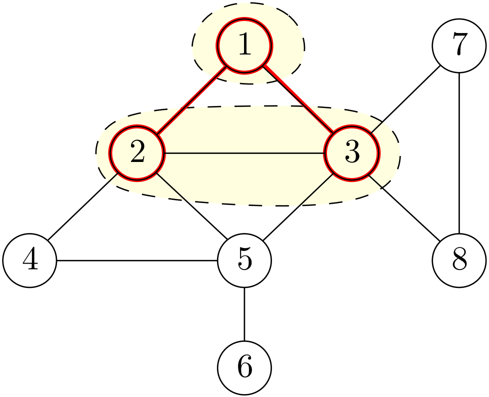
Q: How to compute shortest paths from s when all edges have weight 1?

A: Breadth first search (BFS) from source s



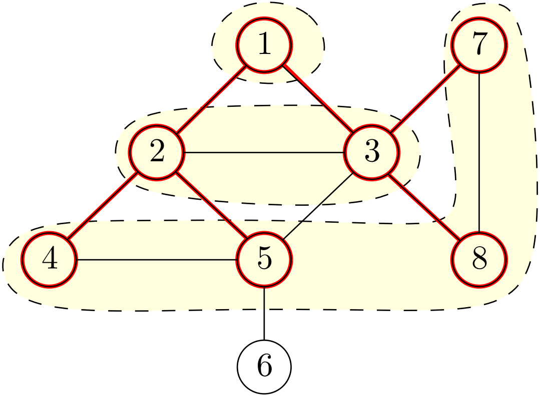
Q: How to compute shortest paths from s when all edges have weight 1?

A: Breadth first search (BFS) from source s



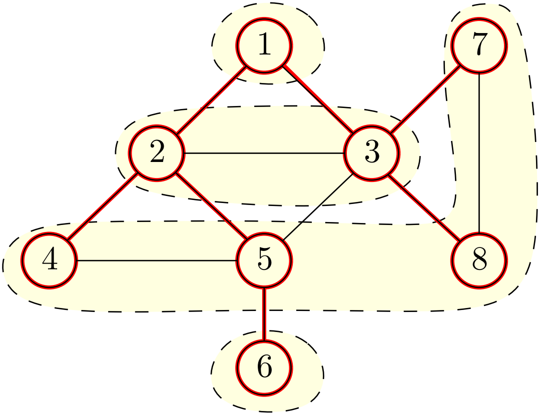
Q: How to compute shortest paths from s when all edges have weight 1?

A: Breadth first search (BFS) from source s



Q: How to compute shortest paths from s when all edges have weight 1?

A: Breadth first search (BFS) from source s



Assumption Weights $w(u, v)$ are integers (w.l.o.g).

Assumption Weights $w(u,v)$ are integers (w.l.o.g).

- An edge of weight $w(u,v)$ is equivalent to a pah of $w(u,v)$ unit-weight edges

Assumption Weights $w(u, v)$ are integers (w.l.o.g).

- An edge of weight $w(u, v)$ is equivalent to a pah of $w(u, v)$ unit-weight edges

Shortest Path Algorithm by Running BFS

1: replace (u, v) of length $w(u, v)$ with a path of $w(u, v)$ unit-weight edges, for every $(u, v) \in E$
2: run BFS
3: $\pi(v) \leftarrow$ vertex from which v is visited
4: $d(v) \leftarrow$ index of the level containing v

Assumption Weights $w(u,v)$ are integers (w.l.o.g).

- An edge of weight $w(u,v)$ is equivalent to a pah of $w(u,v)$ unit-weight edges

Shortest Path Algorithm by Running BFS

replace (u,v) of length $w(u,v)$ with a path of $w(u,v)$ unit-weight edges, for every $(u,v) \in E$
run BFS
$\pi(v) \leftarrow$ vertex from which v is visited
$d(v) \leftarrow$ index of the level containing v

- Problem: $w(u,v)$ may be too large!

Assumption Weights $w(u, v)$ are integers (w.l.o.g).

- An edge of weight $w(u, v)$ is equivalent to a pah of $w(u, v)$ unit-weight edges

Shortest Path Algorithm by Running BFS

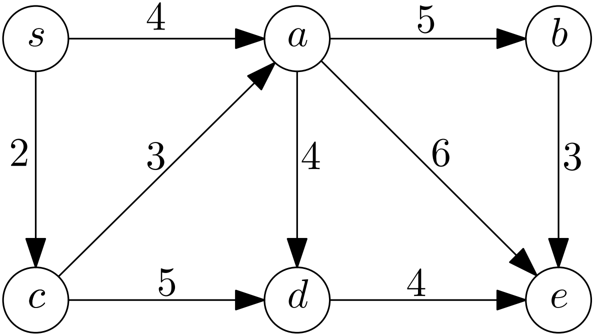
replace (u, v) of length $w(u, v)$ with a path of $w(u, v)$ unit-weight edges, for every $(u, v) \in E$
run BFS virtually
$\pi(v) \leftarrow$ vertex from which v is visited
$d(v) \leftarrow$ index of the level containing v

- Problem: $w(u, v)$ may be too large!

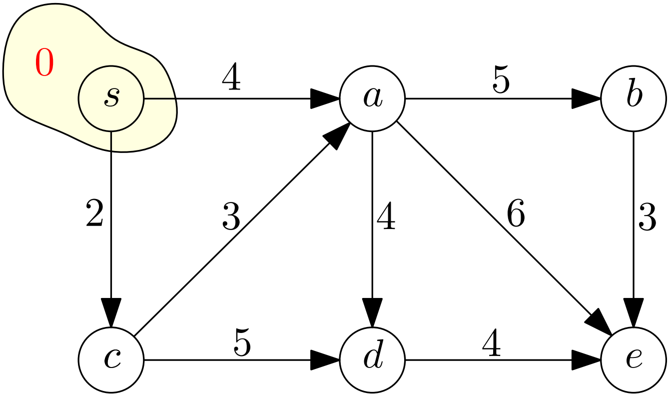
Shortest Path Algorithm by Running BFS Virtually

$S \leftarrow \{s\}, d(s) \leftarrow 0$
while $|S| \leq n$ **do**
    find a $v \notin S$ that minimizes $\displaystyle\min_{u \in S:(u,v) \in E} \{d(u) + w(u,v)\}$
    $S \leftarrow S \cup \{v\}$
    $d(v) \leftarrow \min_{u \in S:(u,v) \in E} \{d(u) + w(u,v)\}$

Virtual BFS: Example

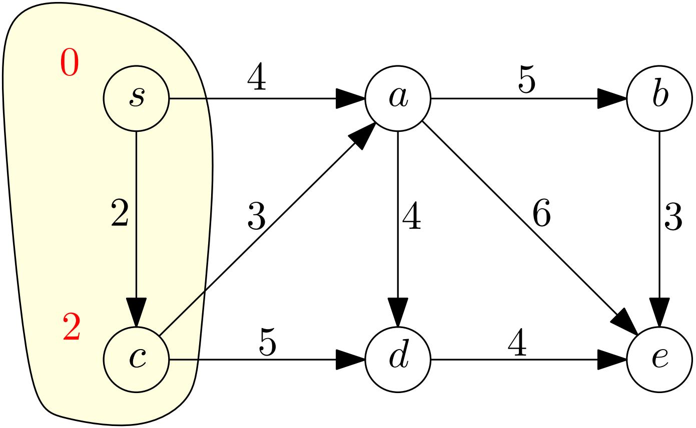


Virtual BFS: Example



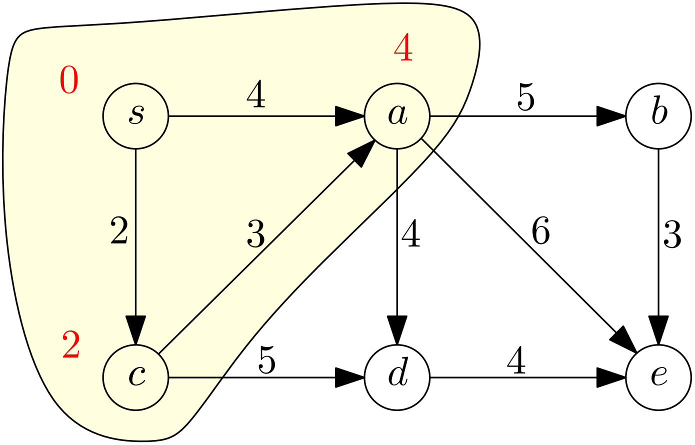
Time 0

Virtual BFS: Example



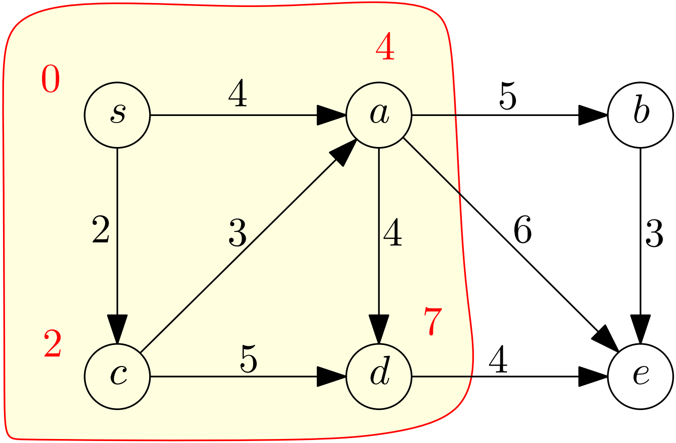
Time 2

Virtual BFS: Example



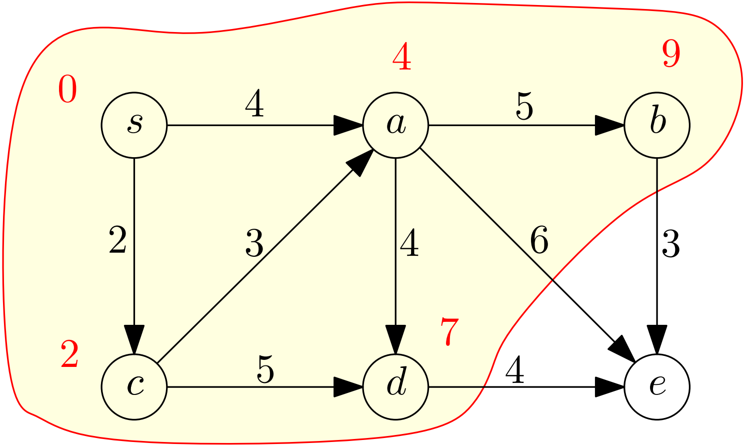
Time 4

Virtual BFS: Example



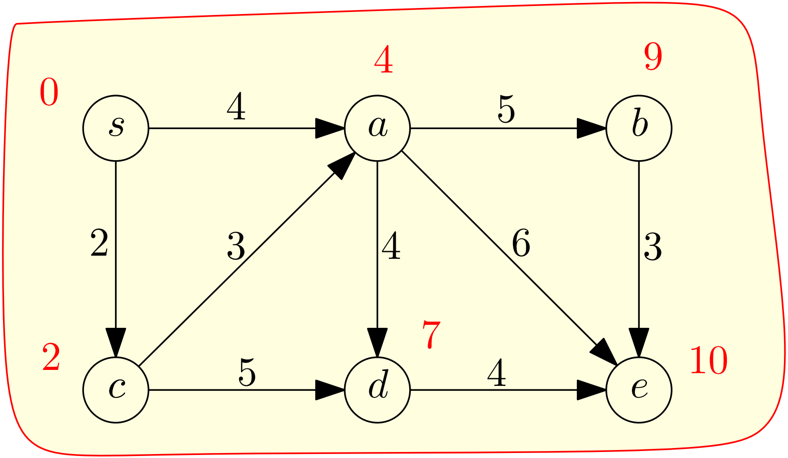
Time 7

Virtual BFS: Example



Time 9

Virtual BFS: Example



Time 10

Outline

Dijkstra's Algorithm

Dijkstra(G, w, s)

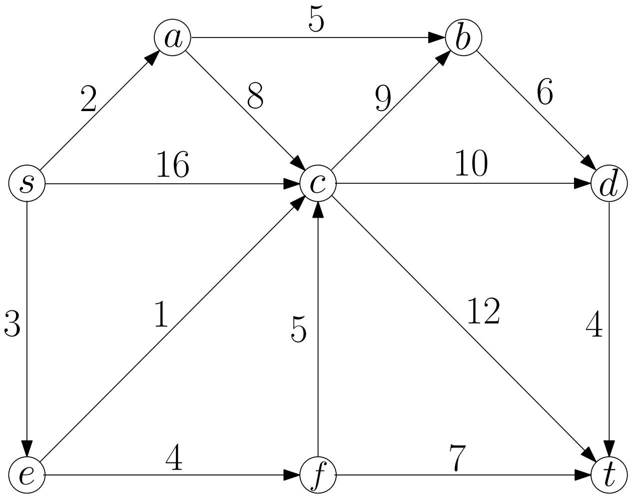
$S \leftarrow \emptyset, d(s) \leftarrow 0$ and $d(v) \leftarrow \infty$ for every $v \in V \setminus \{s\}$
while $S \neq V$ **do**
    $u \leftarrow$ vertex in $V \setminus S$ with the minimum $d(u)$
    add u to S
    for each $v \in V \setminus S$ such that $(u, v) \in E$ **do**
        if $d(u) + w(u, v) < d(v)$ **then**
            $d(v) \leftarrow d(u) + w(u, v)$
            $\pi(v) \leftarrow u$
return (d, π)

Dijkstra's Algorithm

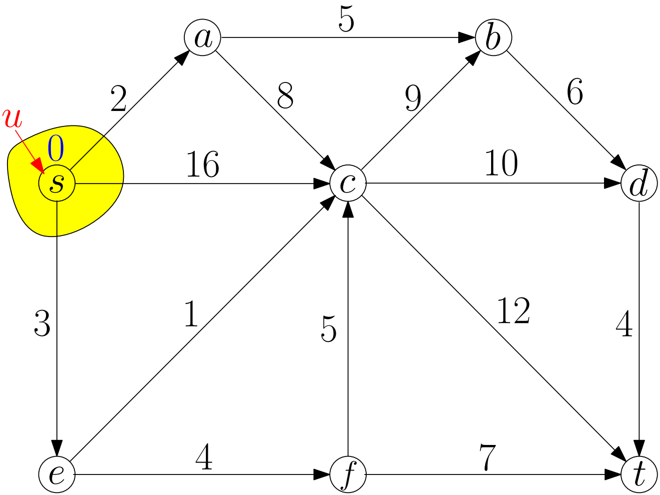
Dijkstra(G, w, s)

$S \leftarrow \emptyset, d(s) \leftarrow 0$ and $d(v) \leftarrow \infty$ for every $v \in V \setminus \{s\}$
while $S \neq V$ **do**
    $u \leftarrow$ vertex in $V \setminus S$ with the minimum $d(u)$
    add u to S
    for each $v \in V \setminus S$ such that $(u, v) \in E$ **do**
        if $d(u) + w(u, v) < d(v)$ **then**
            $d(v) \leftarrow d(u) + w(u, v)$
            $\pi(v) \leftarrow u$
return (d, π)

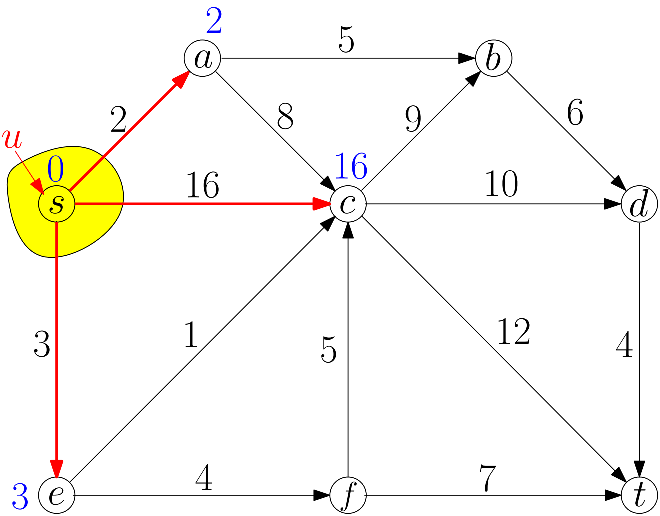
- Running time $= O(n^2)$

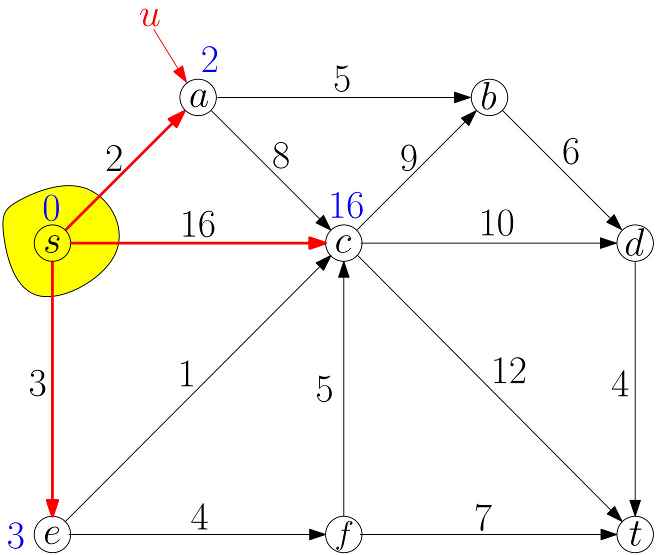


5
a
b
2
8
9
6
16
10
s
c
d
3
1
5
12
4
4
7
e
f
t

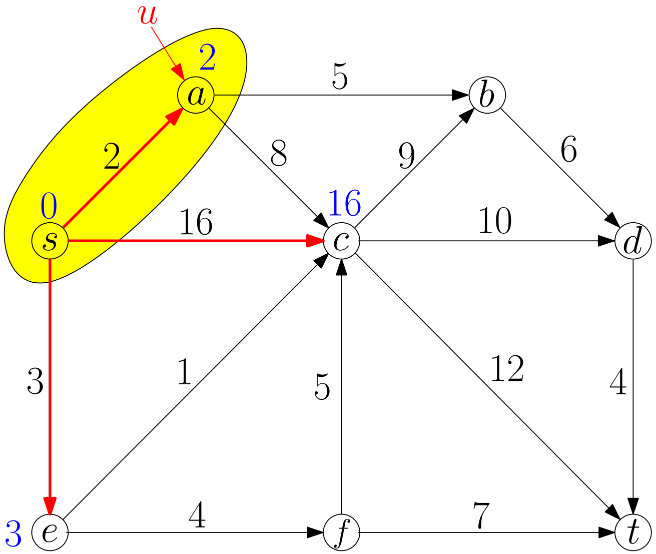


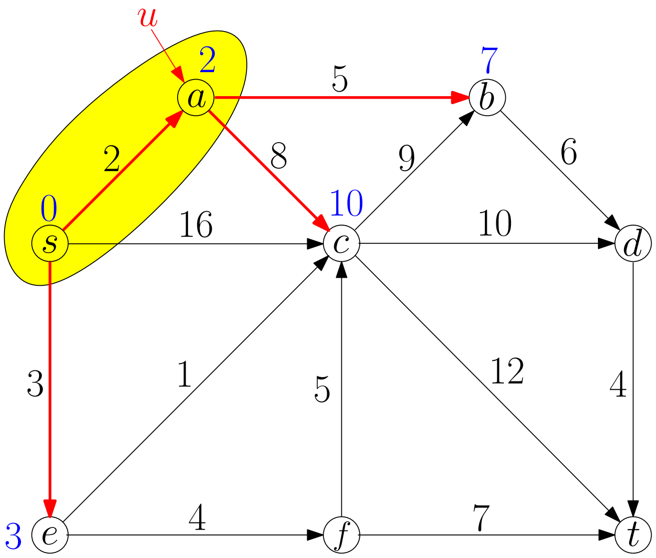
5
2
8
9
6
u
0
16
10
3
1
5
12
4
4
7
a
b
s
c
d
e
f
t

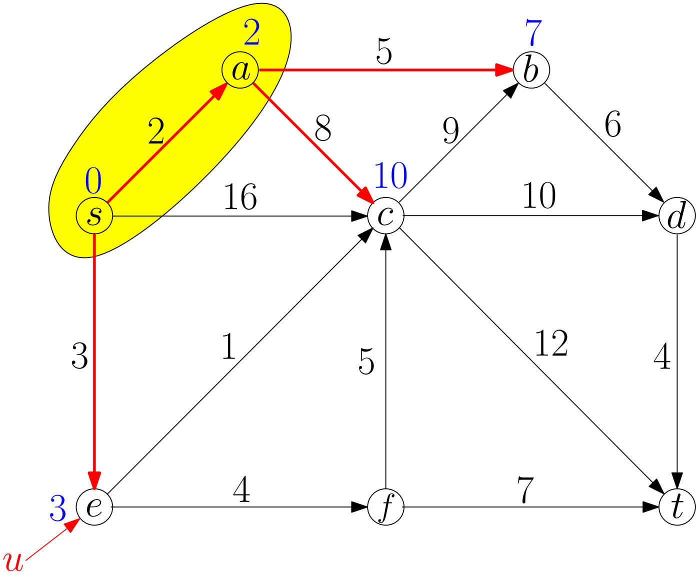

2
5
a
b
u
2
8
9
6
0
16
16
10
s
c
d
3
1
5
12
4
4
7
3
e
f
t

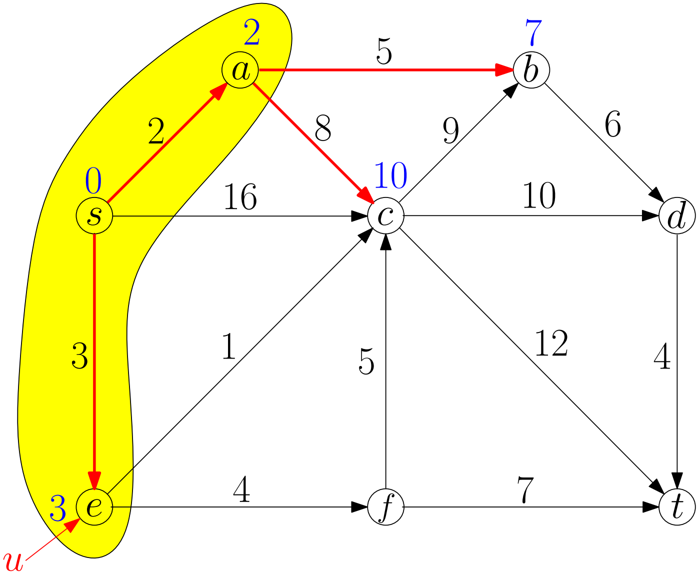


u
2
a
5
b
2
8
9
6
0
16
16
10
s
c
d
3
1
5
12
4
4
7
3
e
f
t

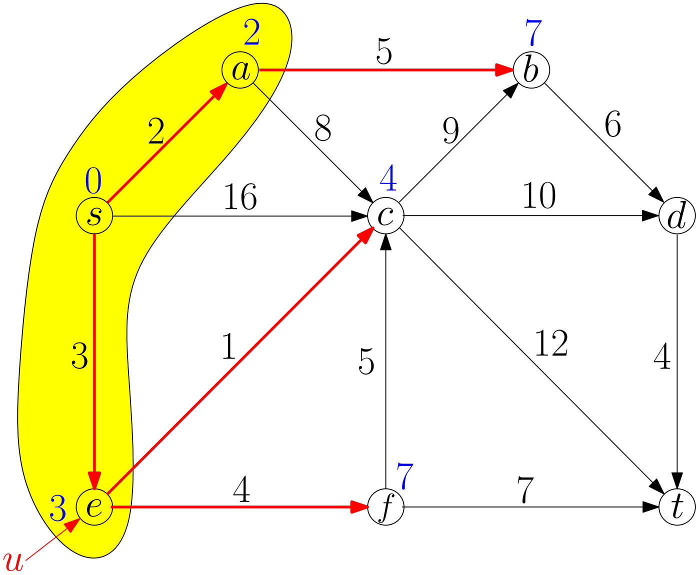

u
2
0
16
3
a
b
c
d
e
f
s
t
5
8
9
6
2
16
10
3
1
5
12
4
4
7


u
2
a
5
7
b
8
9
6
0
s
2
16
10
c
10
d
3
1
5
12
4
3
e
4
f
7
t

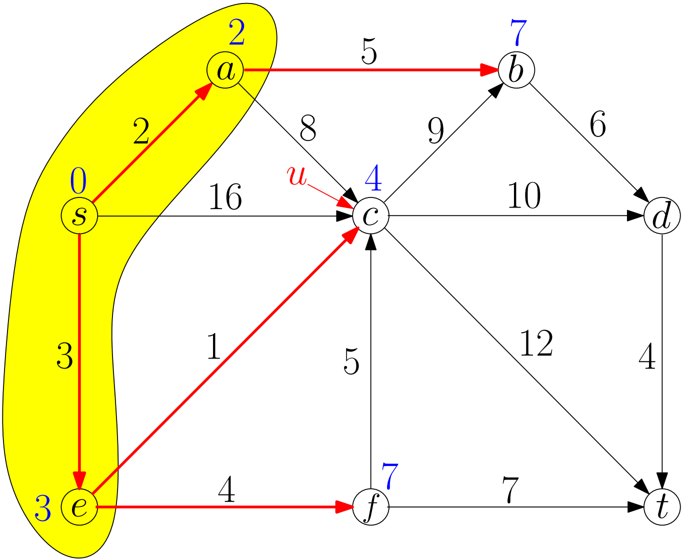

2
7
5
a
b
2
8
9
6
0
10
16
10
s
c
d
3
1
5
12
4
4
7
3
e
f
t
u



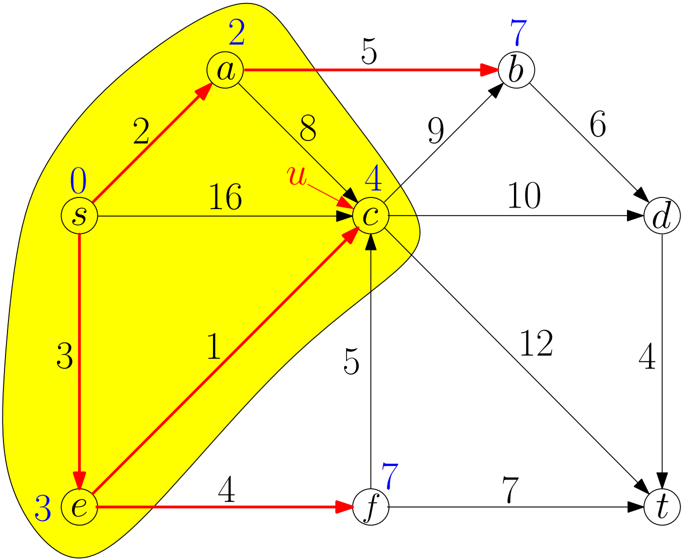
2
7
5
a
b
2
8
9
6
0
10
16
10
s
c
d
3
1
5
12
4
4
7
3
e
f
t
u

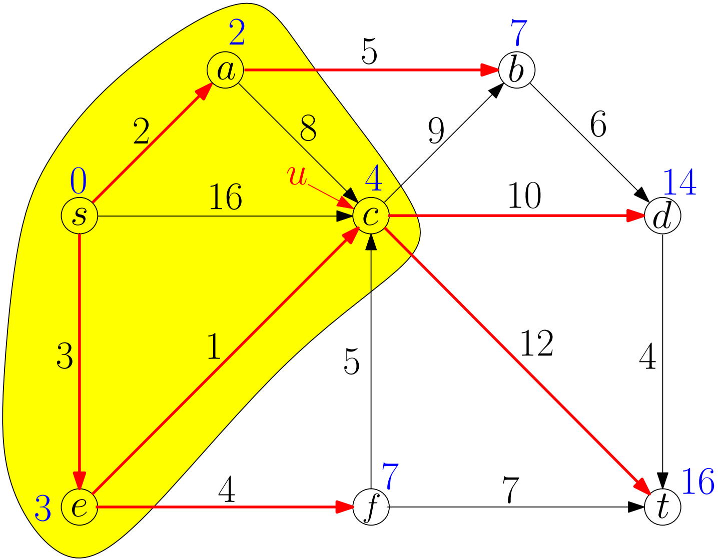


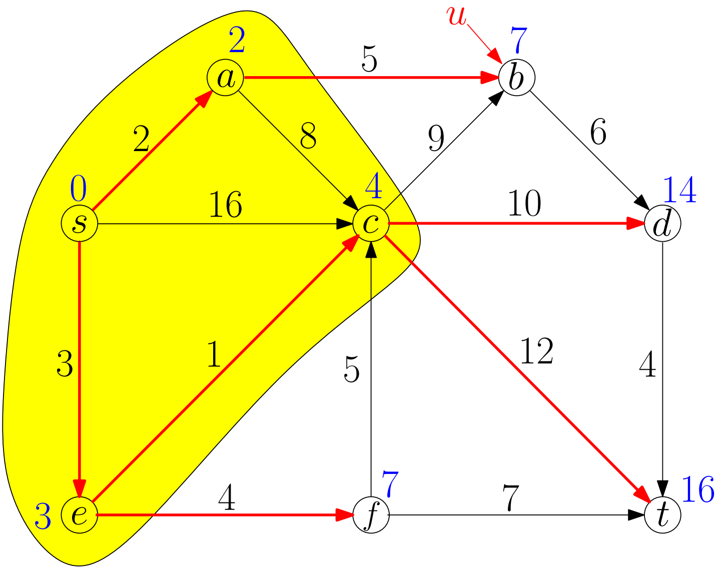
2
7
5
a
b
2
8
9
6
0
4
16
10
s
c
d
3
1
5
12
4
7
4
7
3
e
f
t
u



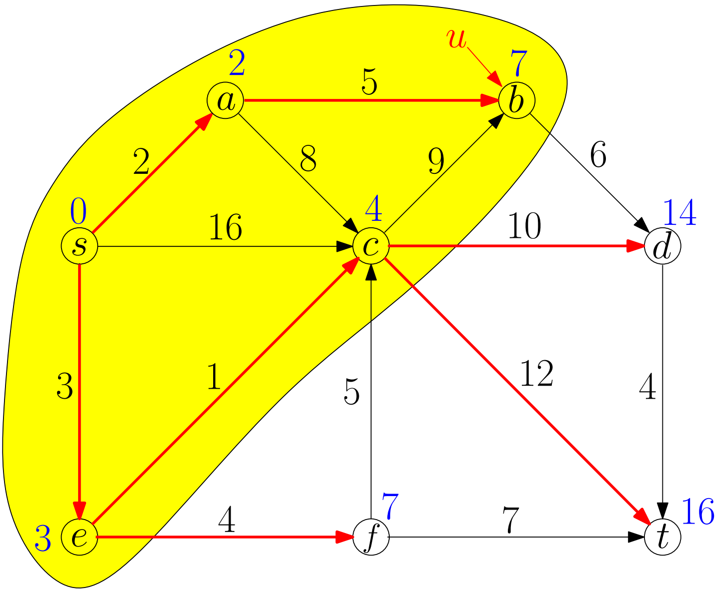
2
a
5
7
b
2
8
9
6
0
s
16
u
4
c
10
d
3
1
5
12
4
7
3
e
4
f
7
t

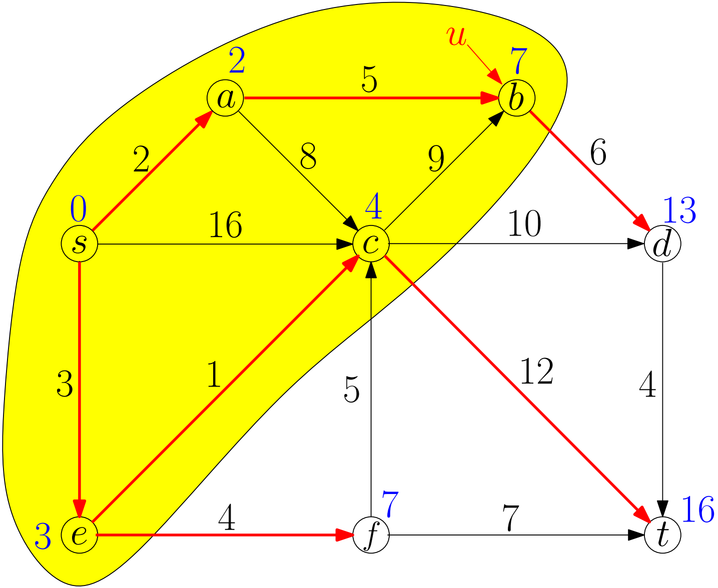

0
2
7
4
7
3
s
a
b
c
d
e
f
t
u
2
5
8
9
6
16
10
3
1
5
12
4
4
7

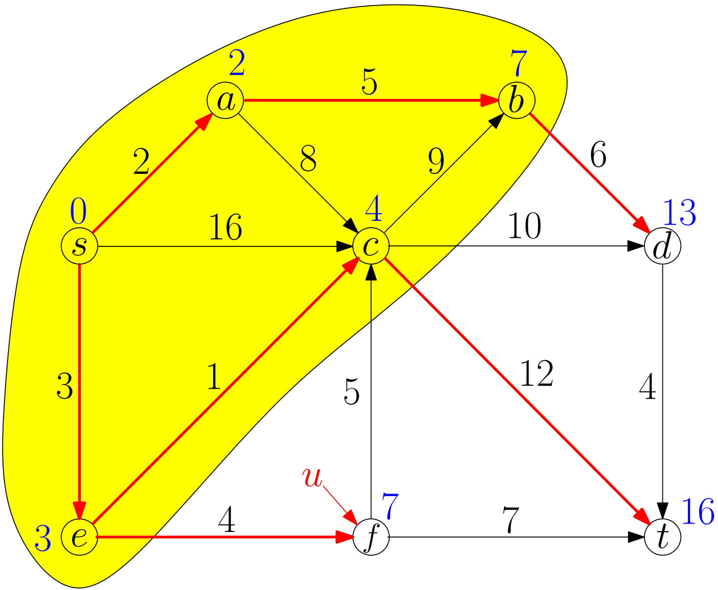

2
7
5
a
b
2
8
9
6
0
u
4
14
16
10
s
c
d
3
1
5
12
4
7
16
3
4
7
e
f
t



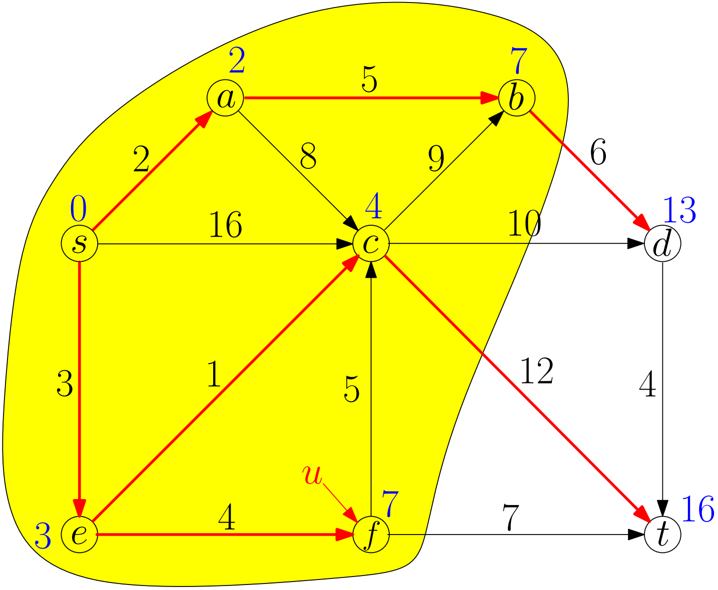
u
2
7
a
b
5
8
9
6
0
4
14
s
c
d
16
10
3
1
5
12
4
3
e
f
t
7
16
4
7


u
2
7
a
5
b
2
8
9
6
0
4
14
s
16
c
10
d
3
1
5
12
4
7
16
3
e
4
f
7
t

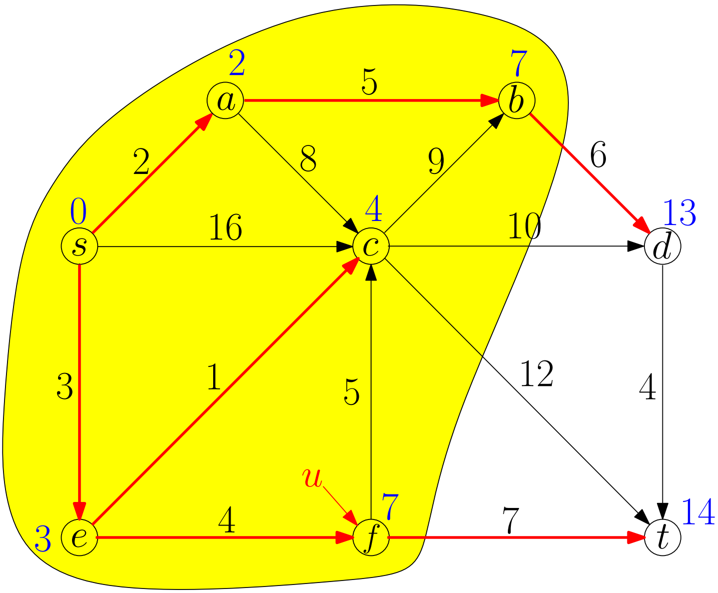

u
2
7
a
b
5
2
8
9
6
0
4
13
s
16
c
10
d
3
1
5
12
4
7
16
3
e
4
f
7
t



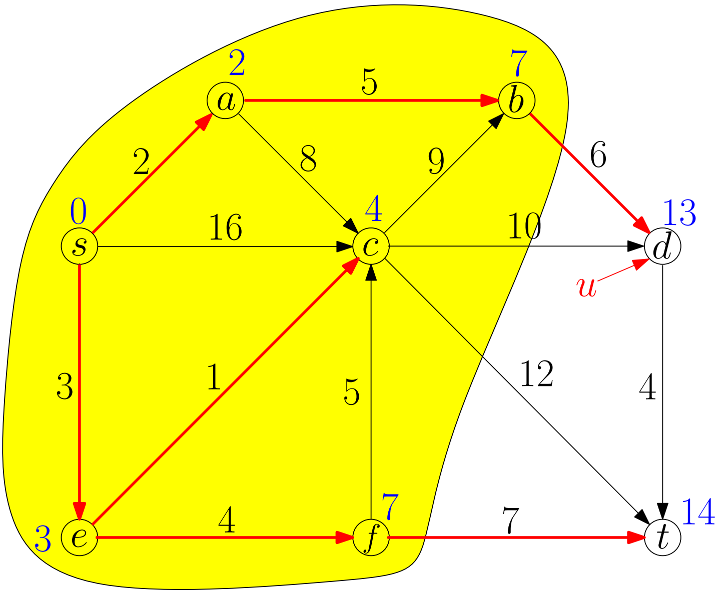
2
7
5
a
b
2
8
9
6
0
4
13
16
10
s
c
d
3
1
5
12
4
u
7
16
4
7
3
e
f
t



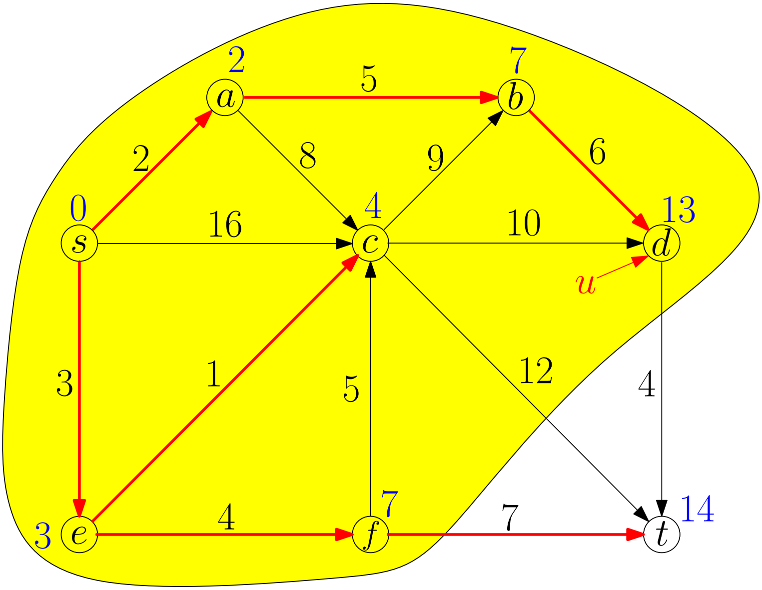
2
7
5
a
b
2
8
9
6
0
4
13
16
10
s
c
d
3
1
5
12
4
u
7
16
3
4
7
e
f
t



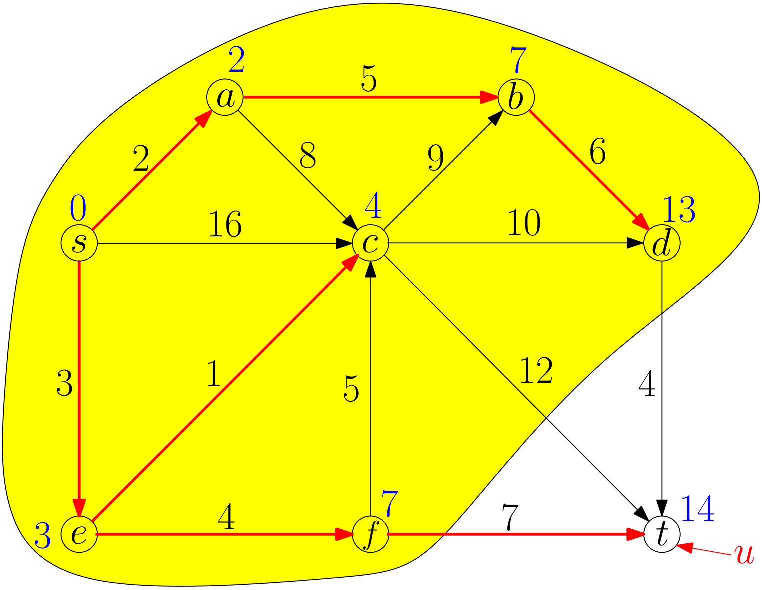
2
7
5
a
b
2
8
9
6
0
4
13
16
10
s
c
d
3
1
5
12
4
u
7
14
4
7
3
e
f
t

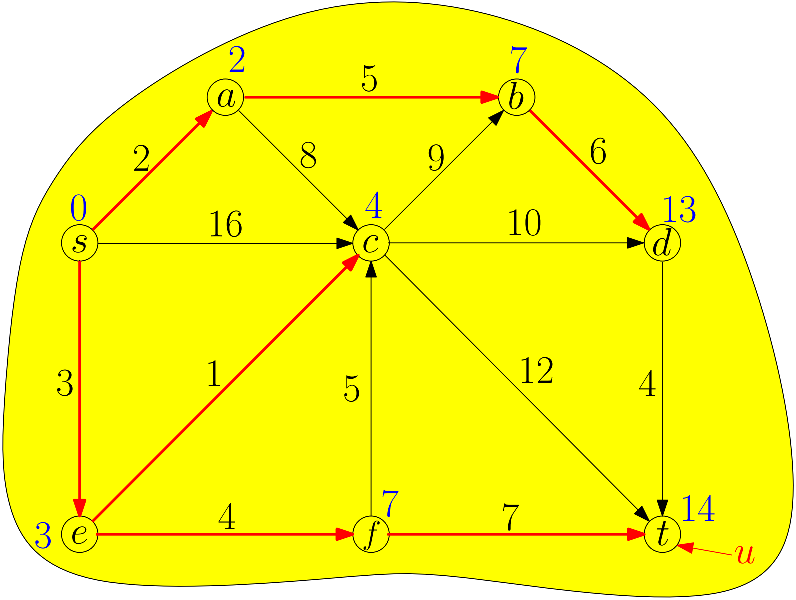


2
7
5
a
b
2
8
9
6
0
4
13
16
10
s
c
d
u
3
1
5
12
4
7
14
3
e
4
f
7
t



2
7
5
a
b
2
8
9
6
0
4
13
16
10
s
c
d
u
3
1
5
12
4
7
14
3
4
7
e
f
t


2
7
5
a
b
2
8
9
6
0
4
13
16
10
s
c
d
3
1
5
12
4
7
7
14
3
4
e
f
t
u



2
7
5
a
b
2
8
9
6
0
4
13
16
10
s
c
d
3
1
5
12
4
7
14
4
7
3
e
f
t
u

Improved Running Time using Priority Queue

Dijkstra(G, w, s)

```
s ← arbitrary vertex in G
S ← ∅, d(s) ← 0 and d(v) ← ∞ for every v ∈ V \ {s}
Q ← empty queue, for each v ∈ V: Q.insert(v, d(v))
while S ≠ V do
    u ← Q.extract_min()
    S ← S ∪ {u}
    for each v ∈ V \ S such that (u, v) ∈ E do
        if d(u) + w(u, v) < d(v) then
            d(v) ← d(u) + w(u, v), Q.decrease_key(v, d(v))
            π(v) ← u
return (π, d)
```

Recall: Prim's Algorithm for MST

MST-Prim(G, w)

$s \leftarrow$ arbitrary vertex in G
$S \leftarrow \emptyset, d(s) \leftarrow 0$ and $d(v) \leftarrow \infty$ for every $v \in V \setminus \{s\}$
$Q \leftarrow$ empty queue, for each $v \in V$: Q.insert$(v, d(v))$
while $S \neq V$ **do**
    $u \leftarrow Q$.extract_min()
    $S \leftarrow S \cup \{u\}$
    for each $v \in V \setminus S$ such that $(u, v) \in E$ **do**
        if $w(u, v) < d(v)$ **then**
            $d(v) \leftarrow w(u, v)$, Q.decrease_key$(v, d(v))$
            $\pi(v) \leftarrow u$
return $\big\{(u, \pi(u)) | u \in V \setminus \{s\}\big\}$

Improved Running Time

Running time:
$O(n) \times$ (time for extract_min) $+ O(m) \times$ (time for decrease_key)

Priority-Queue	extract_min	decrease_key	Time
Heap	$O(\log n)$	$O(\log n)$	$O(m \log n)$
Fibonacci Heap	$O(\log n)$	$O(1)$	$O(n \log n + m)$

Outline

Recall: Single Source Shortest Path Problem

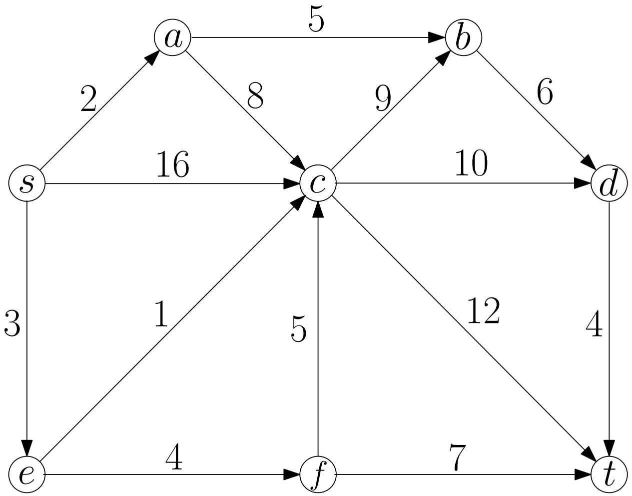
Single Source Shortest Paths

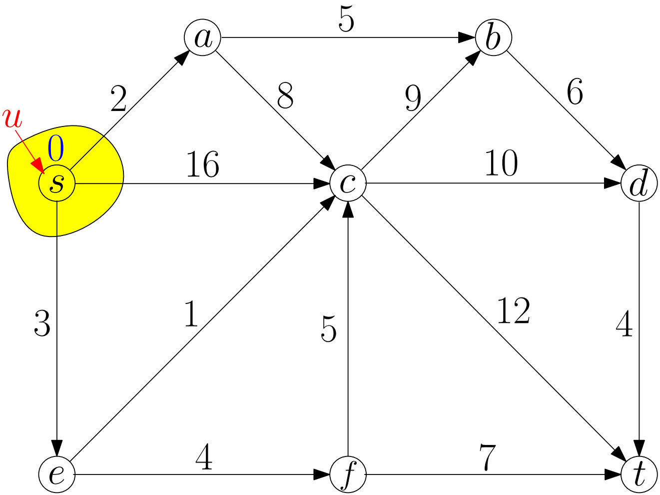
Input: directed graph $G = (V, E)$, $s \in V$

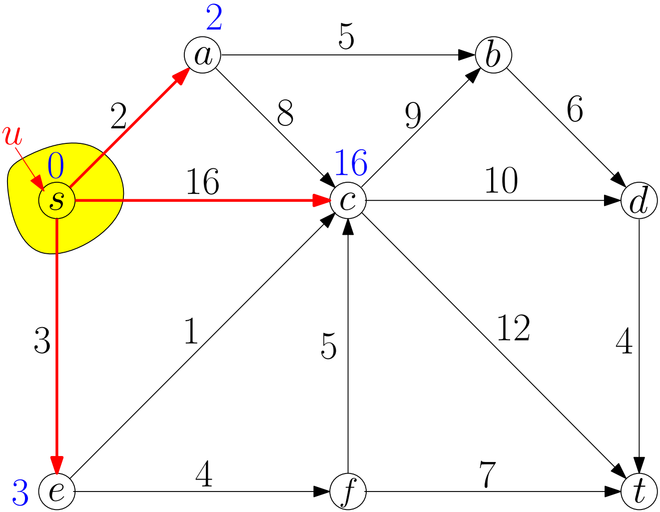
$w : E \to \mathbb{R}_{\geq 0}$

Output: shortest paths from s to all other vertices $v \in V$

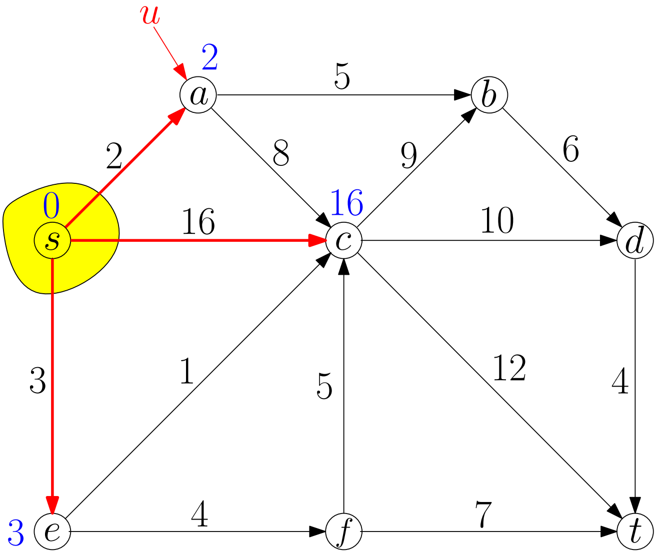
- Algorithm for the problem: Dijkstra's algorithm


a
b
s
c
d
e
f
t
5
2
8
9
6
16
10
3
1
5
12
4
4
7

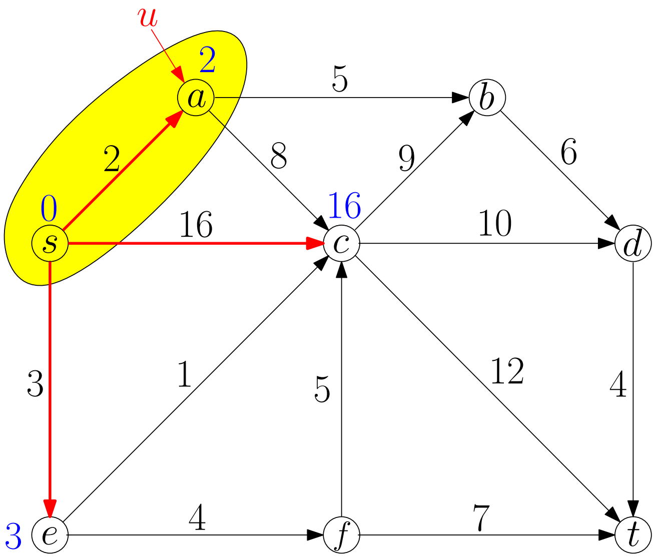

u
0
s
a
b
c
d
e
f
t
2
5
8
9
6
16
10
3
1
5
12
4
4
7



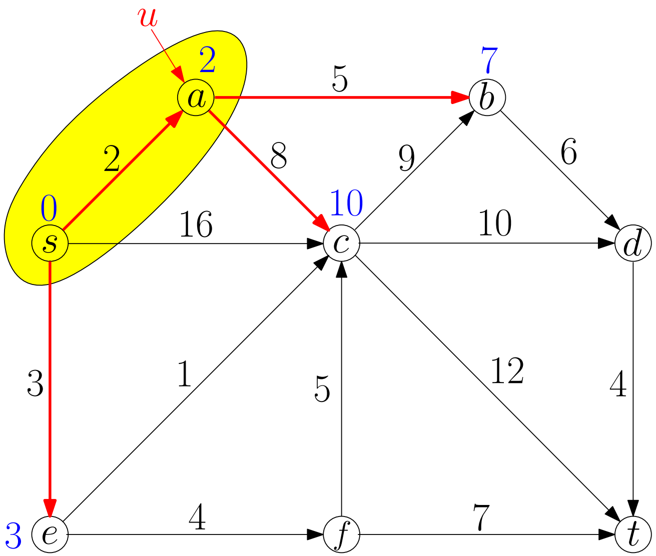
2
5
a
b
2
8
9
6
u
0
16
s
16
c
10
d
3
1
5
12
4
3
e
4
f
7
t

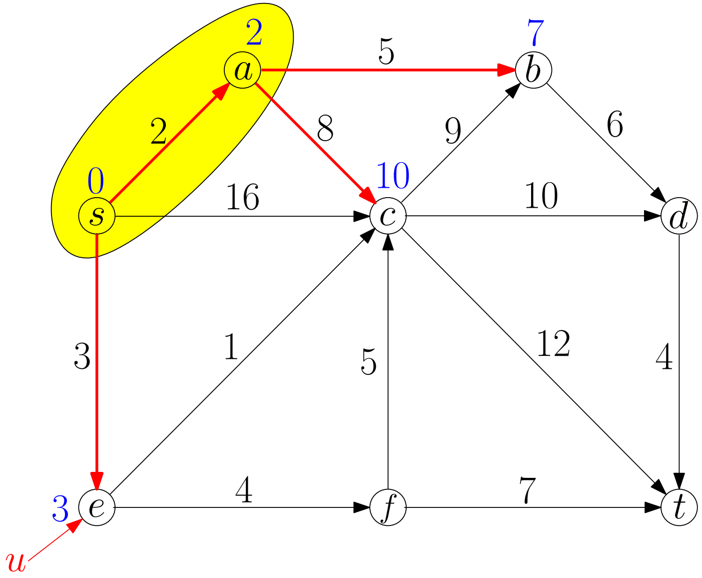


u
2
a
5
b
2
8
9
6
0
s
16
16
c
10
d
3
1
5
12
4
3
e
4
f
7
t

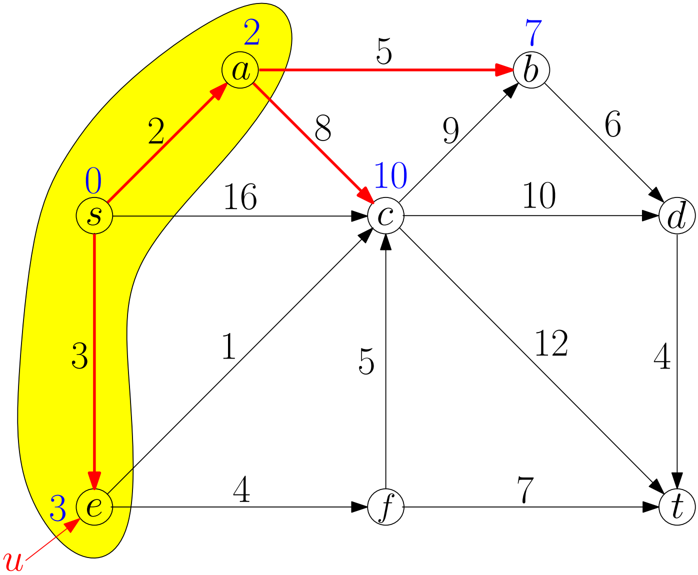


u
2
0
16
3
a
b
c
d
s
e
f
t
5
8
9
6
2
16
10
3
1
5
12
4
4
7

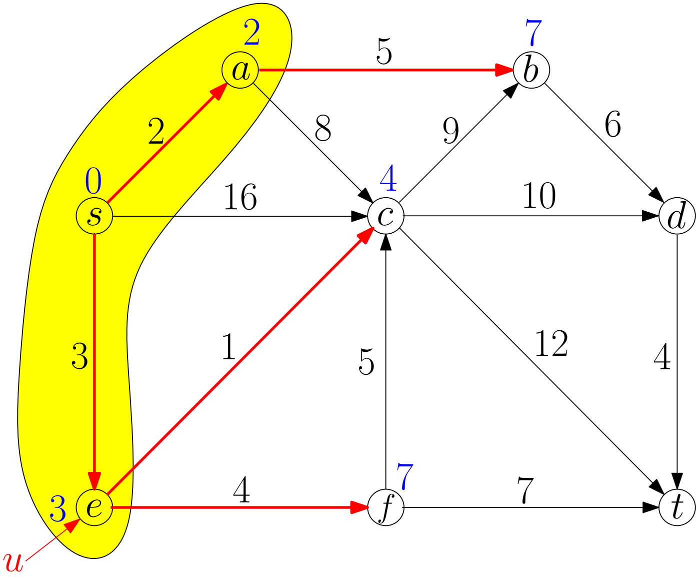

u
2
a
5
7
b
2
8
9
6
0
s
16
10
c
10
d
3
1
5
12
4
3
e
4
f
7
t



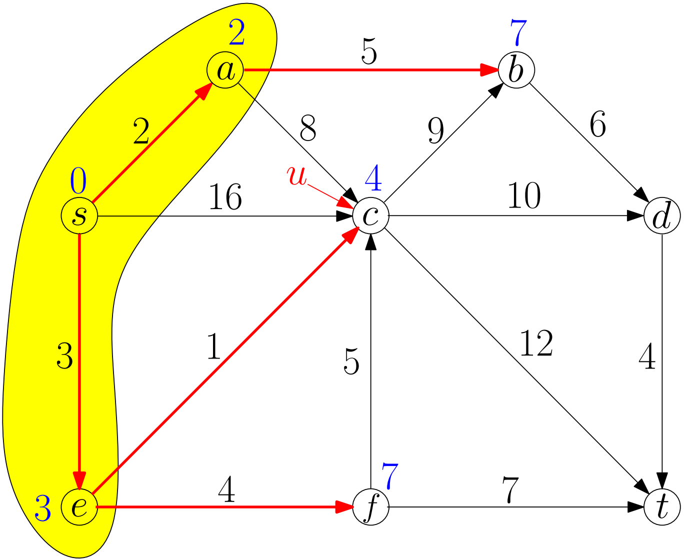
2
7
5
a
b
2
8
9
6
0
10
16
10
s
c
d
3
1
5
12
4
3
e
4
f
7
t
u



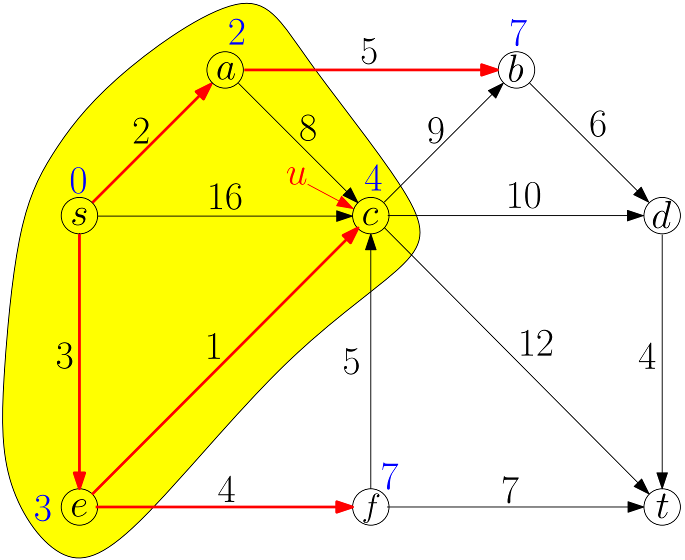
2
a
5
7
b
2
8
9
6
0
s
16
10
c
10
d
3
1
5
12
4
3
e
4
f
7
t
u



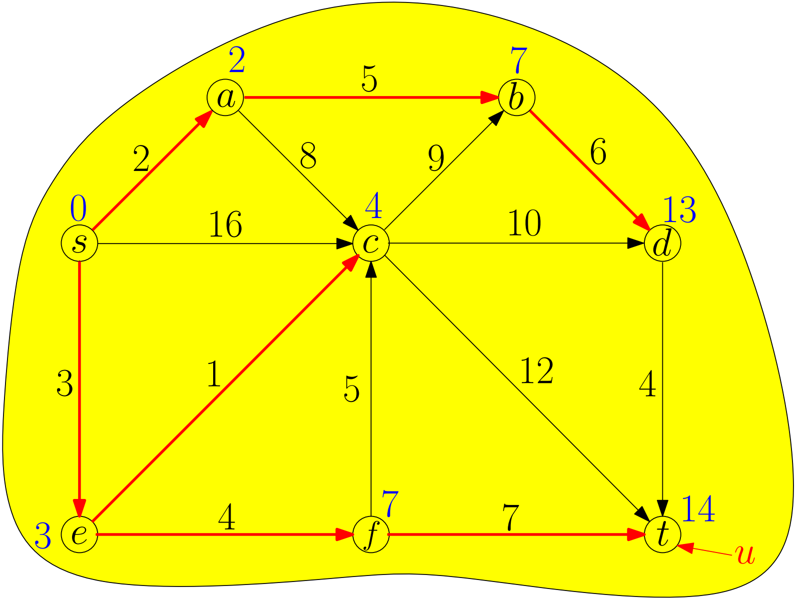
2
7
5
a
b
2
8
9
6
0
4
16
10
s
c
d
3
1
5
12
4
7
4
7
3
e
f
t
u



2
a
7
b
5
8
9
6
2
0
s
16
u
4
c
10
d
3
1
5
12
4
7
3
e
4
f
7
t



2
7
5
a
b
2
8
9
6
0
u
4
16
10
s
c
d
3
1
5
12
4
7
3
e
4
f
7
t


2
7
5
a
b
2
8
9
6
0
4
13
16
10
s
c
d
3
1
5
12
4
7
7
14
4
3
e
f
t
u

Dijkstra's Algorithm Using Priorty Queue

Dijkstra(G, w, s)

$S \leftarrow \emptyset, d(s) \leftarrow 0$ and $d(v) \leftarrow \infty$ for every $v \in V \setminus \{s\}$
$Q \leftarrow$ empty queue, for each $v \in V$: Q.insert$(v, d(v))$
while $S \neq V$ **do**
    $u \leftarrow Q$.extract_min()
    $S \leftarrow S \cup \{u\}$
    for each $v \in V \setminus S$ such that $(u, v) \in E$ **do**
        if $d(u) + w(u, v) < d(v)$ **then**
            $d(v) \leftarrow d(u) + w(u, v)$, Q.decrease_key$(v, d(v))$
            $\pi(v) \leftarrow u$
return (π, d)

- Running time $= O(m + n \lg n)$.

Single Source Shortest Paths

Input: directed graph $G = (V, E)$, $s \in V$

assume all vertices are reachable from s

$w : E \to \mathbb{R}$

Output: shortest paths from s to all other vertices $v \in V$

Single Source Shortest Paths

Input: directed graph $G = (V, E)$, $s \in V$

assume all vertices are reachable from s

$w : E \to \mathbb{R}$

Output: shortest paths from s to all other vertices $v \in V$

Single Source Shortest Paths, Weights May be Negative

Input: directed graph $G = (V, E)$, $s \in V$

assume all vertices are reachable from s

$w : E \to \mathbb{R}$

Output: shortest paths from s to all other vertices $v \in V$

Single Source Shortest Paths, Weights May be Negative

Input: directed graph $G = (V, E)$, $s \in V$

assume all vertices are reachable from s

$w : E \to \mathbb{R}$

Output: shortest paths from s to all other vertices $v \in V$

- In transition graphs, negative weights make sense

Single Source Shortest Paths, Weights May be Negative

Input: directed graph $G = (V, E)$, $s \in V$

assume all vertices are reachable from s

$w : E \to \mathbb{R}$

Output: shortest paths from s to all other vertices $v \in V$

- In transition graphs, negative weights make sense
- If we sell a item: 'having the item' $\to$ 'not having the item', weight is negative (we gain money)

Single Source Shortest Paths, Weights May be Negative

Input: directed graph $G = (V, E)$, $s \in V$

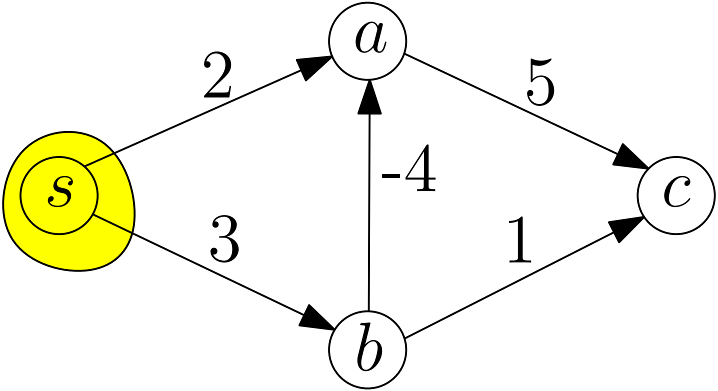
assume all vertices are reachable from s

$w : E \to \mathbb{R}$

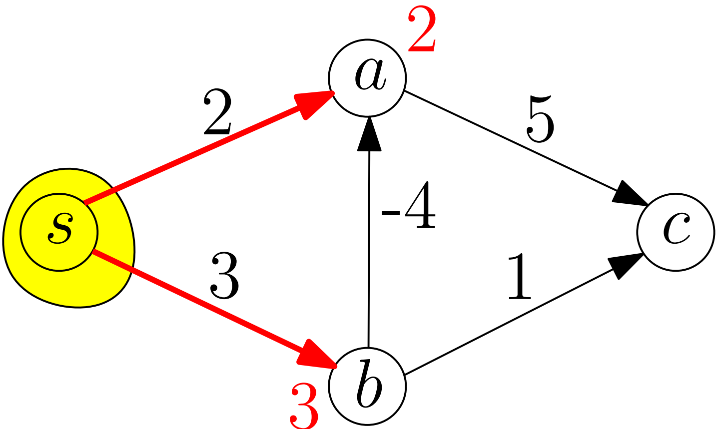
Output: shortest paths from s to all other vertices $v \in V$

- In transition graphs, negative weights make sense
- If we sell a item: 'having the item' $\to$ 'not having the item', weight is negative (we gain money)
- Dijkstra's algorithm does not work any more!

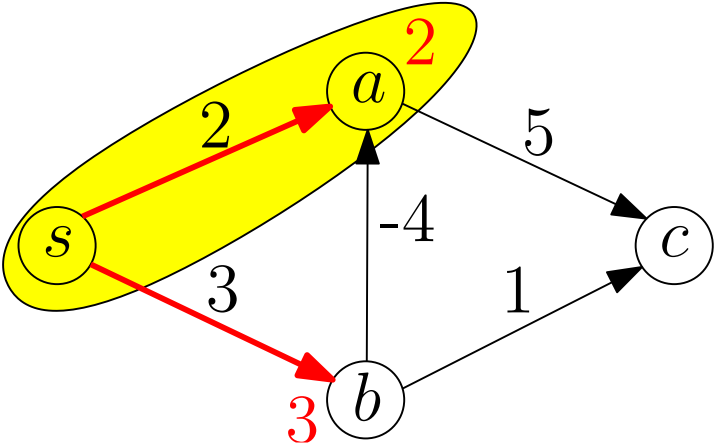
Dijkstra's Algorithm Fails if We Have Negative Weights



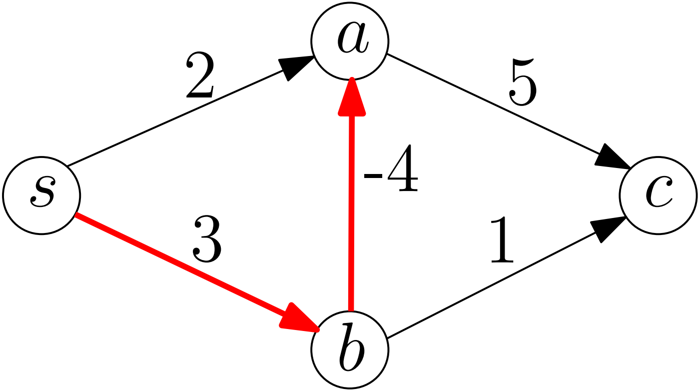
Dijkstra's Algorithm Fails if We Have Negative Weights

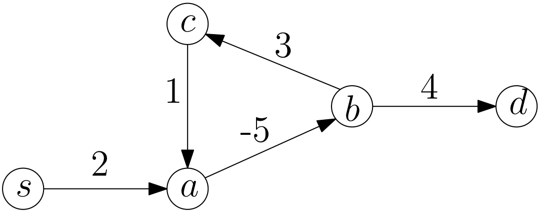


Dijkstra's Algorithm Fails if We Have Negative Weights

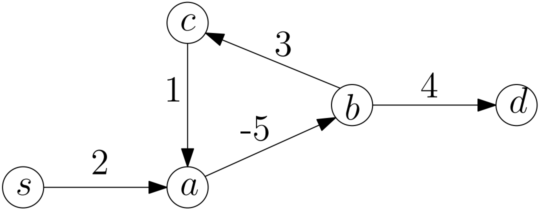


Dijkstra's Algorithm Fails if We Have Negative Weights

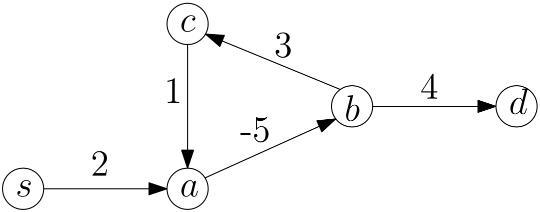




c
3
1
b
4
d
-5
2
s
a

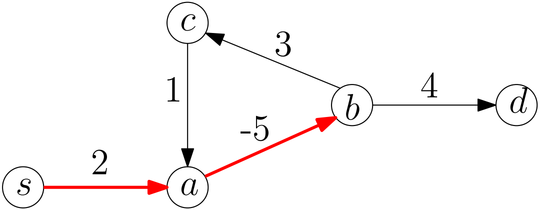


Q: What is the length of the shortest path from s to d?



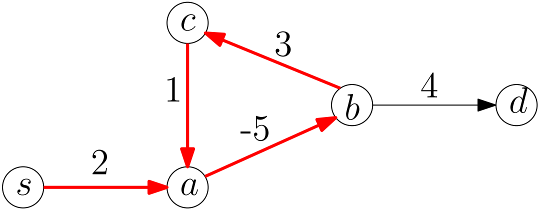
Q: What is the length of the shortest path from s to d?

A: $-\infty$



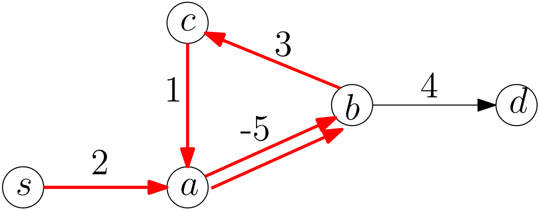
Q: What is the length of the shortest path from s to d?

A: $-\infty$



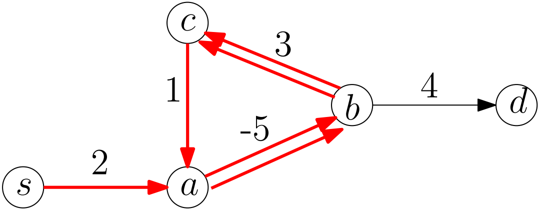
Q: What is the length of the shortest path from s to d?

A: $-\infty$



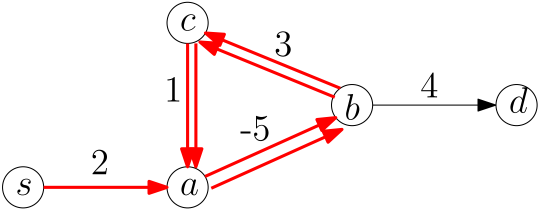
Q: What is the length of the shortest path from s to d?

A: $-\infty$



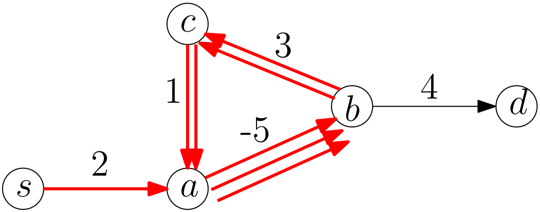
Q: What is the length of the shortest path from s to d?

A: $-\infty$



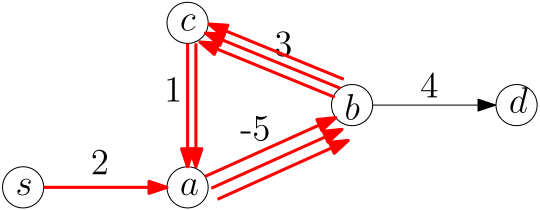
Q: What is the length of the shortest path from s to d?

A: $-\infty$



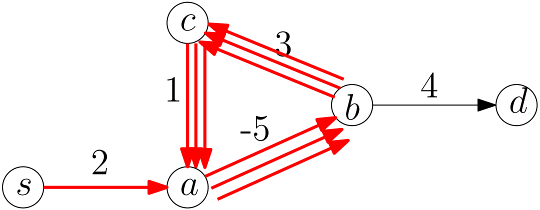
Q: What is the length of the shortest path from s to d?

A: $-\infty$



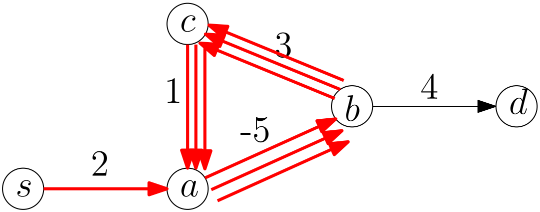
Q: What is the length of the shortest path from s to d?

A: $-\infty$



Q: What is the length of the shortest path from s to d?

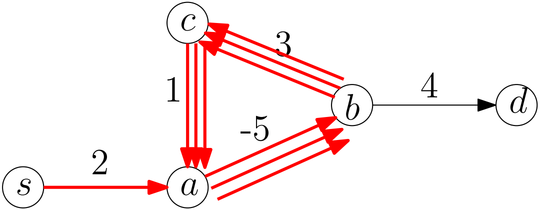
A: $-\infty$



Q: What is the length of the shortest path from s to d?

A: $-\infty$

Def. A negative cycle is a cycle in which the total weight of edges is negative.

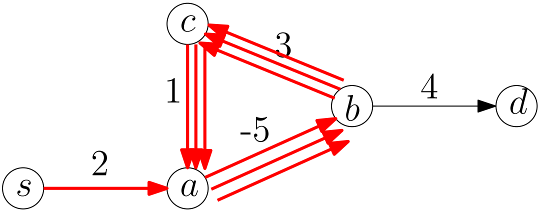


Q: What is the length of the shortest path from s to d?

A: $-\infty$

Def. A negative cycle is a cycle in which the total weight of edges is negative.

Dealing with Negative Cycles



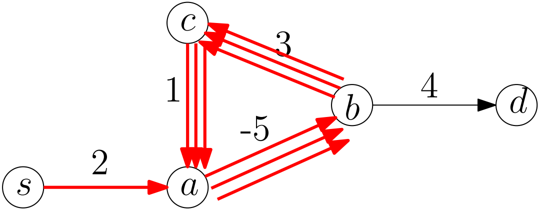
Q: What is the length of the shortest path from s to d?

A: $-\infty$

Def. A negative cycle is a cycle in which the total weight of edges is negative.

Dealing with Negative Cycles

- assume the input graph does not contain negative cycles, or



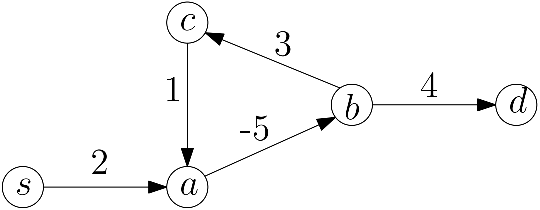
Q: What is the length of the shortest path from s to d?

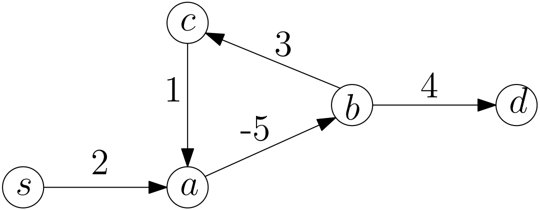
A: $-\infty$

Def. A negative cycle is a cycle in which the total weight of edges is negative.

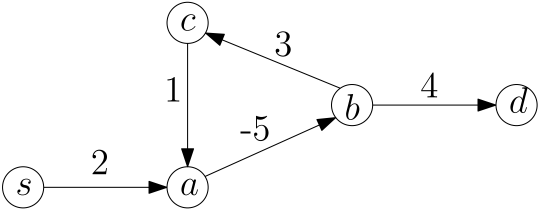
Dealing with Negative Cycles

- assume the input graph does not contain negative cycles, or
- allow algorithm to report "negative cycle exists"


c
3
1
4
b
d
-5
2
s
a

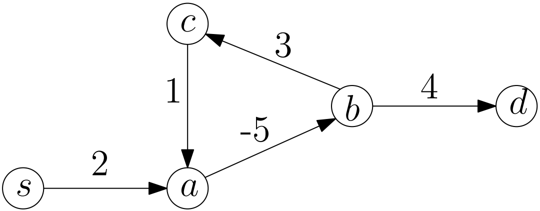


Q: What is the length of the shortest simple path from s to d?



Q: What is the length of the shortest simple path from s to d?

A: 1



Q: What is the length of the shortest simple path from s to d?

A: 1

- Unfortunately, computing the shortest simple path between two vertices is an NP-hard problem.

Outline

Defining Cells of Table

Single Source Shortest Paths, Weights May be Negative

Input: directed graph $G = (V, E)$, $s \in V$

assume all vertices are reachable from s

$w : E \to \mathbb{R}$

Output: shortest paths from s to all other vertices $v \in V$

Defining Cells of Table

Single Source Shortest Paths, Weights May be Negative

Input: directed graph $G = (V, E)$, $s \in V$

assume all vertices are reachable from s

$w : E \to \mathbb{R}$

Output: shortest paths from s to all other vertices $v \in V$

- first try: $f[v]$: length of shortest path from s to v

Defining Cells of Table

Single Source Shortest Paths, Weights May be Negative

Input: directed graph $G = (V, E)$, $s \in V$

assume all vertices are reachable from s

$w : E \to \mathbb{R}$

Output: shortest paths from s to all other vertices $v \in V$

- first try: $f[v]$: length of shortest path from s to v
- issue: do not know in which order we compute $f[v]$'s

Defining Cells of Table

Single Source Shortest Paths, Weights May be Negative

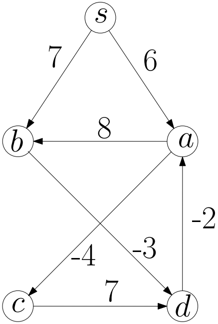
Input: directed graph $G = (V, E)$, $s \in V$

assume all vertices are reachable from s

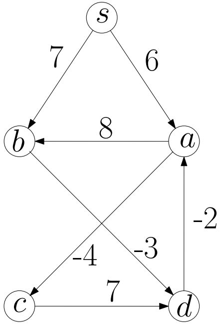
$w : E \to \mathbb{R}$

Output: shortest paths from s to all other vertices $v \in V$

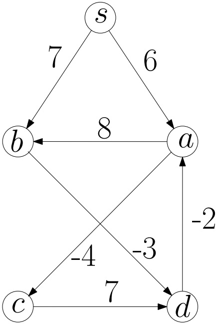
- first try: $f[v]$: length of shortest path from s to v
- issue: do not know in which order we compute $f[v]$'s
- $f^\ell[v]$, $\ell \in \{0, 1, 2, 3 \cdots, n-1\}$, $v \in V$: length of shortest path from s to v that uses at most ℓ edges



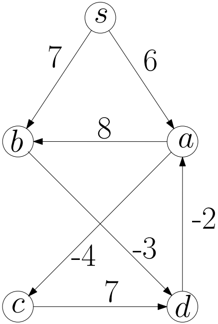
- $f^\ell[v]$, $\ell \in \{0, 1, 2, 3 \cdots, n-1\}$, $v \in V$: length of shortest path from s to v that uses at most ℓ edges



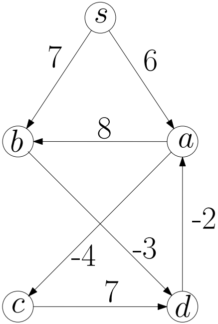
- $f^\ell[v]$, $\ell \in \{0, 1, 2, 3 \cdots, n-1\}$, $v \in V$: length of shortest path from s to v that uses at most ℓ edges
- $f^2[a] =$



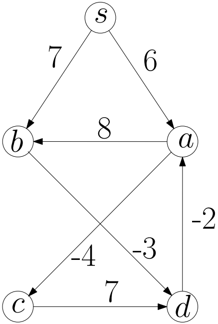
- $f^\ell[v]$, $\ell \in \{0, 1, 2, 3 \cdots, n-1\}$, $v \in V$: length of shortest path from s to v that uses at most ℓ edges
- $f^2[a] = 6$



- $f^\ell[v]$, $\ell \in \{0, 1, 2, 3 \cdots, n-1\}$, $v \in V$: length of shortest path from s to v that uses at most ℓ edges
- $f^2[a] = 6$
- $f^3[a] =$

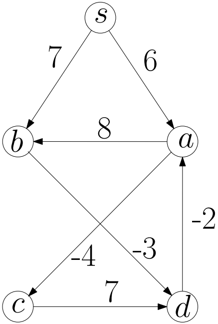


- $f^\ell[v]$, $\ell \in \{0, 1, 2, 3 \cdots, n-1\}$, $v \in V$: length of shortest path from s to v that uses at most ℓ edges
- $f^2[a] = 6$
- $f^3[a] = 2$



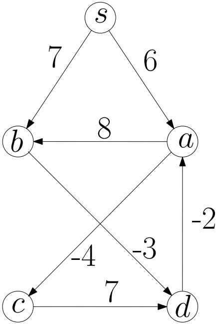
- $f^\ell[v]$, $\ell \in \{0, 1, 2, 3 \cdots, n-1\}$, $v \in V$: length of shortest path from s to v that uses at most ℓ edges
- $f^2[a] = 6$
- $f^3[a] = 2$

$$f^\ell[v] = \begin{cases} & \ell = 0, v = s \\ & \ell = 0, v \neq s \\ & \ell > 0 \end{cases}$$



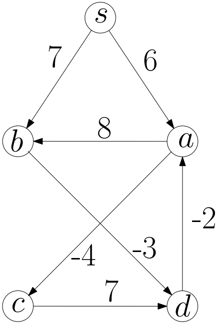
- $f^{\ell}[v]$, $\ell \in \{0, 1, 2, 3 \cdots, n-1\}$, $v \in V$: length of shortest path from s to v that uses at most ℓ edges
- $f^2[a] = 6$
- $f^3[a] = 2$

$$f^{\ell}[v] = \begin{cases} 0 & \ell = 0, v = s \\ & \ell = 0, v \neq s \\ & \ell > 0 \end{cases}$$



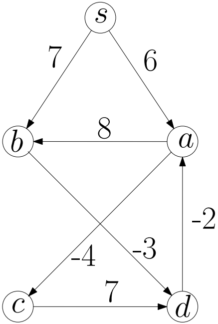
- $f^\ell[v]$, $\ell \in \{0, 1, 2, 3 \cdots, n-1\}$, $v \in V$: length of shortest path from s to v that uses at most ℓ edges
- $f^2[a] = 6$
- $f^3[a] = 2$

$$f^\ell[v] = \begin{cases} 0 & \ell = 0, v = s \\ \infty & \ell = 0, v \neq s \\ & \ell > 0 \end{cases}$$



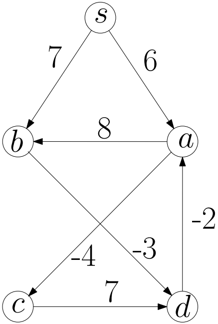
- $f^{\ell}[v]$, $\ell \in \{0, 1, 2, 3 \cdots, n-1\}$, $v \in V$: length of shortest path from s to v that uses at most ℓ edges
- $f^2[a] = 6$
- $f^3[a] = 2$

$$f^{\ell}[v] = \begin{cases} 0 & \ell = 0, v = s \\ \infty & \ell = 0, v \neq s \\ \min \left\{ \right. & \ell > 0 \end{cases}$$



- $f^\ell[v]$, $\ell \in \{0, 1, 2, 3 \cdots, n-1\}$, $v \in V$: length of shortest path from s to v that uses at most ℓ edges
- $f^2[a] = 6$
- $f^3[a] = 2$

$$f^\ell[v] = \begin{cases} 0 & \ell = 0, v = s \\ \infty & \ell = 0, v \neq s \\ \min \left\{ \quad f^{\ell-1}[v] \right. & \ell > 0 \end{cases}$$



- $f^\ell[v]$, $\ell \in \{0, 1, 2, 3 \cdots, n-1\}$, $v \in V$: length of shortest path from s to v that uses at most ℓ edges
- $f^2[a] = 6$
- $f^3[a] = 2$

$$f^\ell[v] = \begin{cases} 0 & \ell = 0, v = s \\ \infty & \ell = 0, v \neq s \\ \min \begin{cases} f^{\ell-1}[v] \\ \min_{u:(u,v)\in E} \left(f^{\ell-1}[u] + w(u,v)\right) \end{cases} & \ell > 0 \end{cases}$$

dynamic-programming(G, w, s)

$f^0[s] \leftarrow 0$ and $f^0[v] \leftarrow \infty$ for any $v \in V \setminus \{s\}$
for $\ell \leftarrow 1$ to $n-1$ **do**
    copy $f^{\ell-1} \to f^{\ell}$
    for each $(u, v) \in E$ **do**
        if $f^{\ell-1}[u] + w(u, v) < f^{\ell}[v]$ **then**
            $f^{\ell}[v] \leftarrow f^{\ell-1}[u] + w(u, v)$
return $(f^{n-1}[v])_{v \in V}$

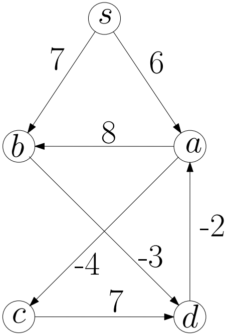
dynamic-programming(G, w, s)

$f^0[s] \leftarrow 0$ and $f^0[v] \leftarrow \infty$ for any $v \in V \setminus \{s\}$
for $\ell \leftarrow 1$ to $n-1$ **do**
  copy $f^{\ell-1} \rightarrow f^{\ell}$
  for each $(u, v) \in E$ **do**
    if $f^{\ell-1}[u] + w(u, v) < f^{\ell}[v]$ **then**
      $f^{\ell}[v] \leftarrow f^{\ell-1}[u] + w(u, v)$
return $(f^{n-1}[v])_{v \in V}$

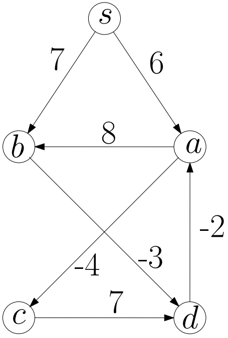
Obs. Assuming there are no negative cycles, then a shortest path contains at most $n-1$ edges

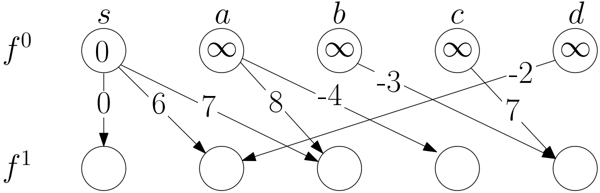
Dynamic Programming: Example

	s	a	b	c	d
f^0	0	∞	∞	∞	∞

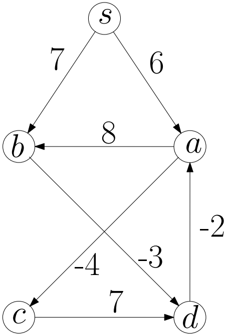


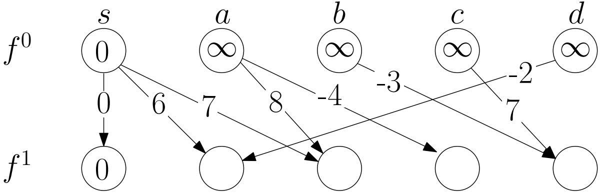
Dynamic Programming: Example



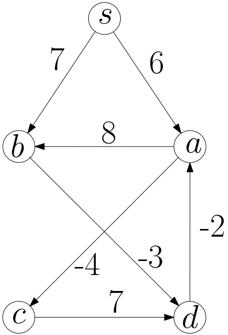


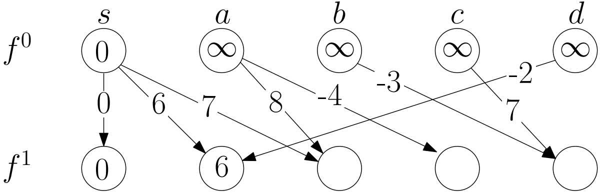
Dynamic Programming: Example



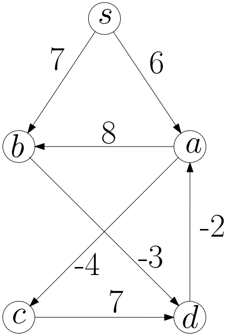


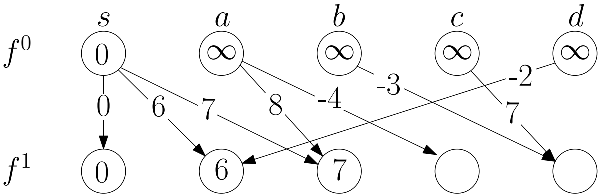
Dynamic Programming: Example



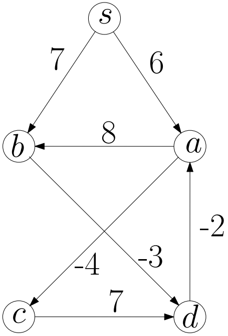


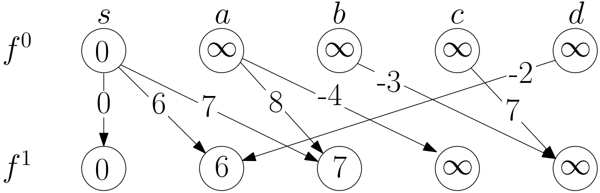
Dynamic Programming: Example



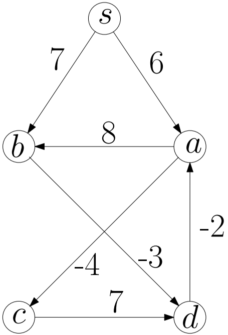


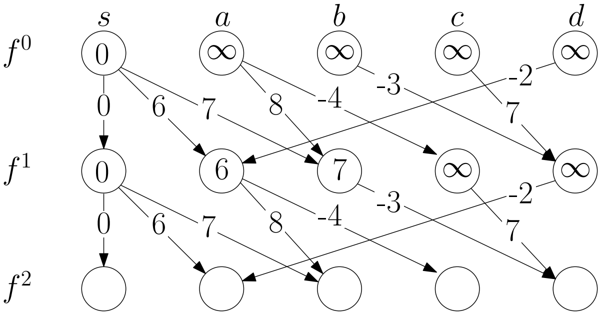
Dynamic Programming: Example



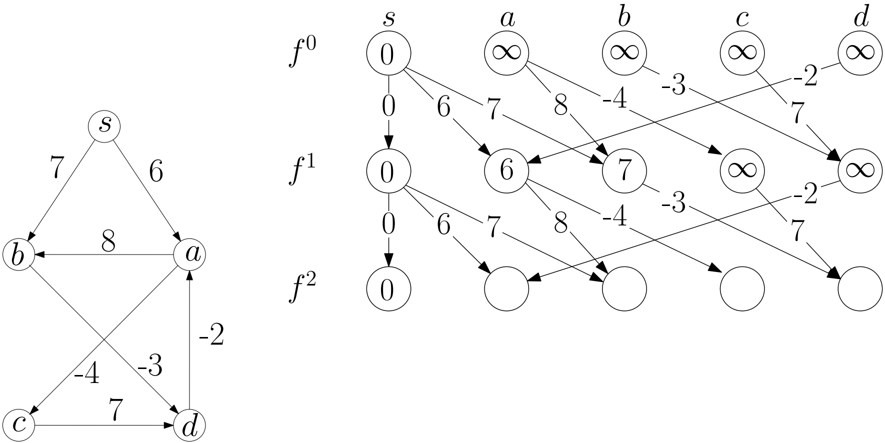


Dynamic Programming: Example

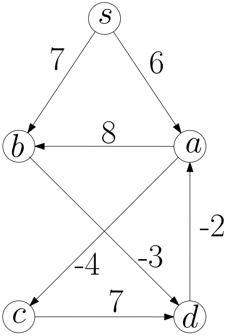


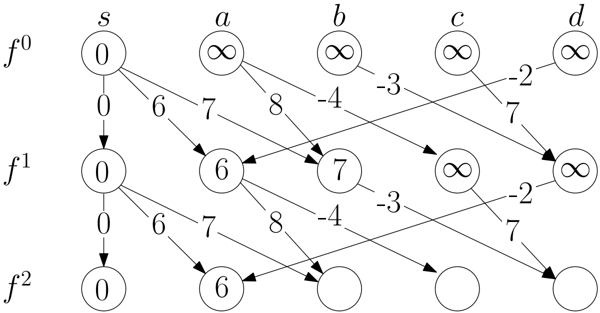


Dynamic Programming: Example

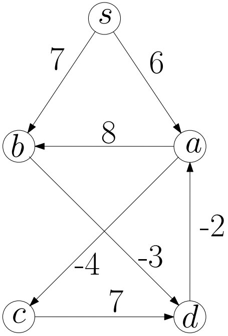


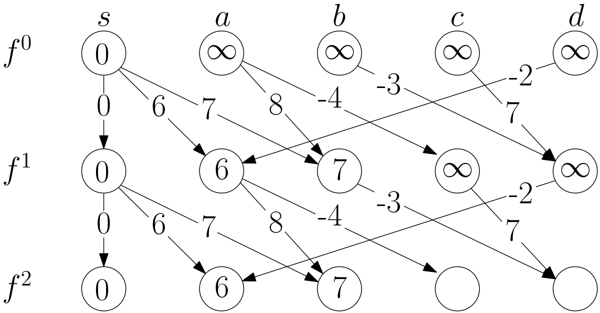
Dynamic Programming: Example



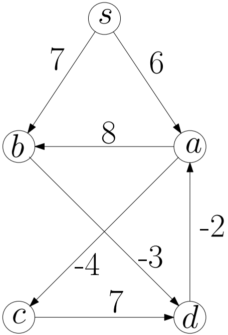


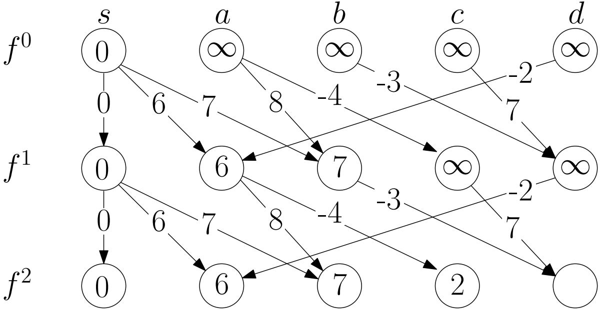
Dynamic Programming: Example



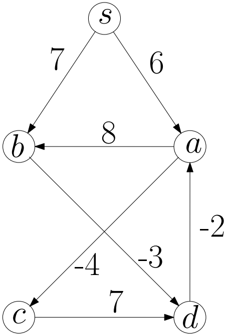


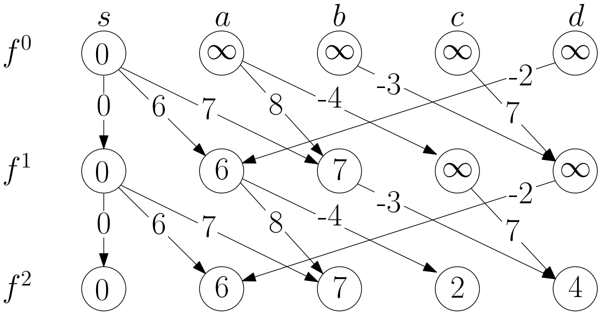
Dynamic Programming: Example



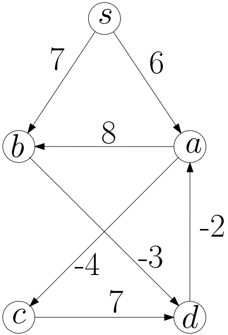


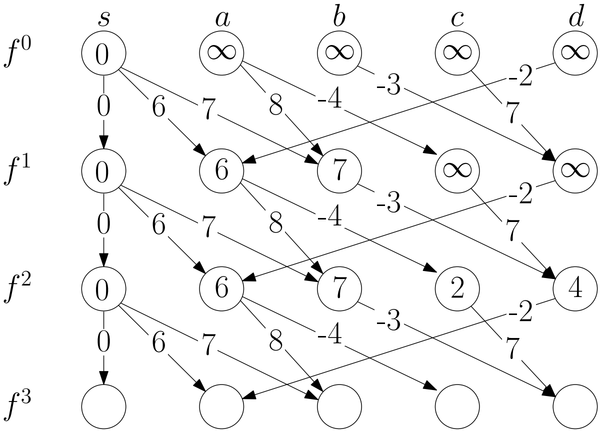
Dynamic Programming: Example



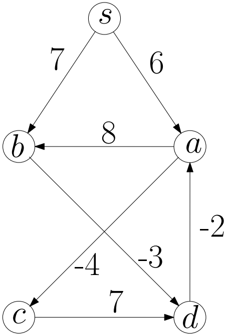


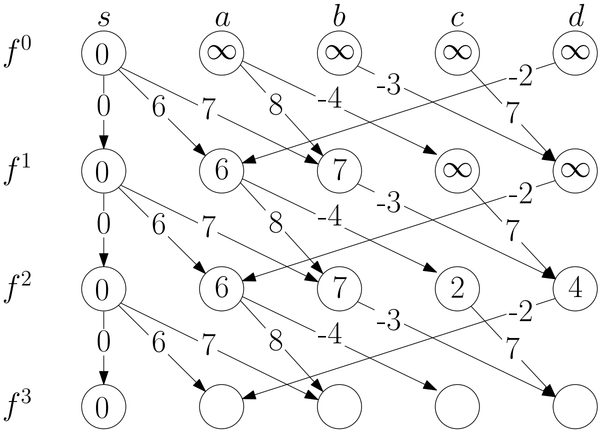
Dynamic Programming: Example



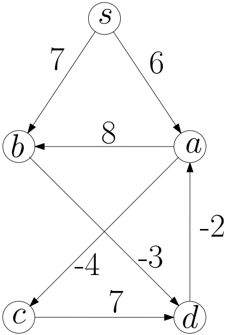


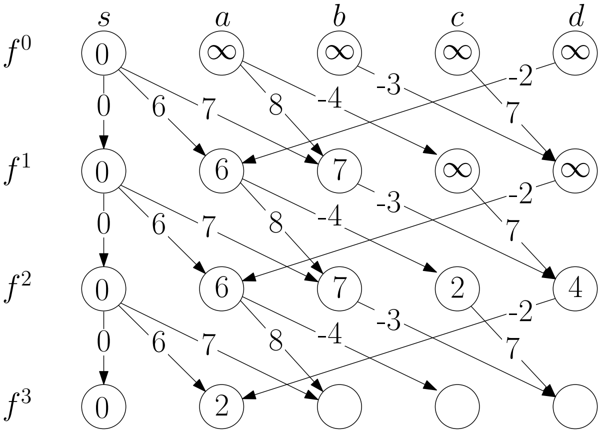
Dynamic Programming: Example



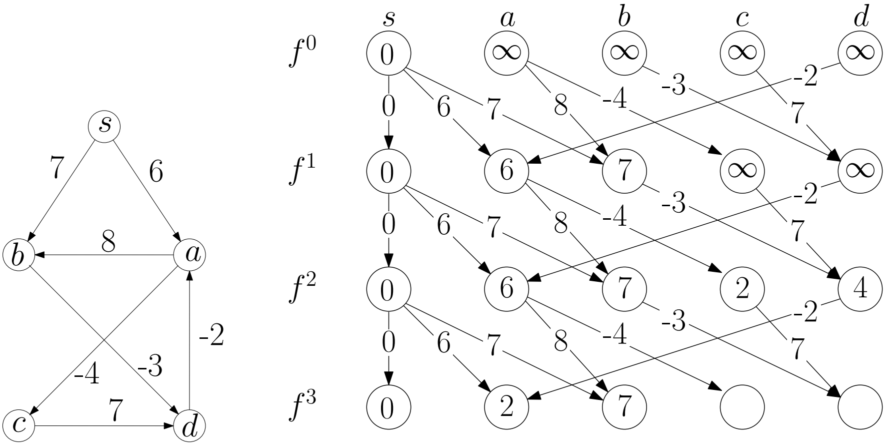


Dynamic Programming: Example

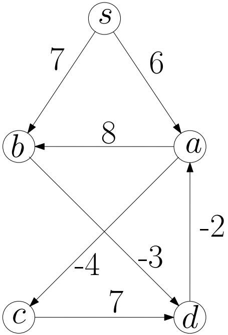


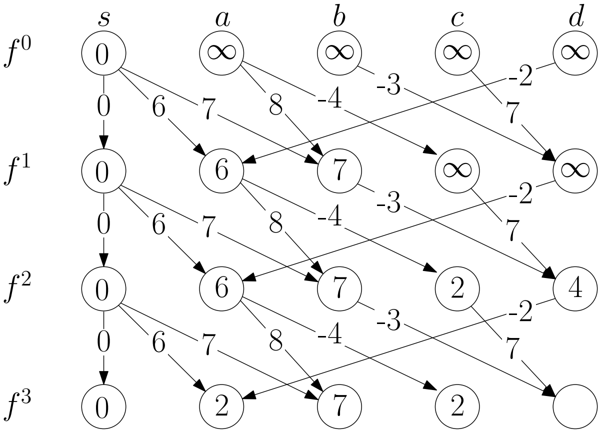


Dynamic Programming: Example

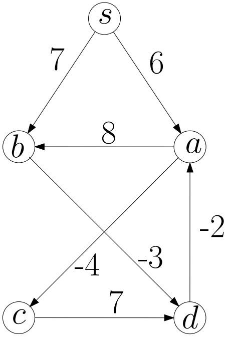


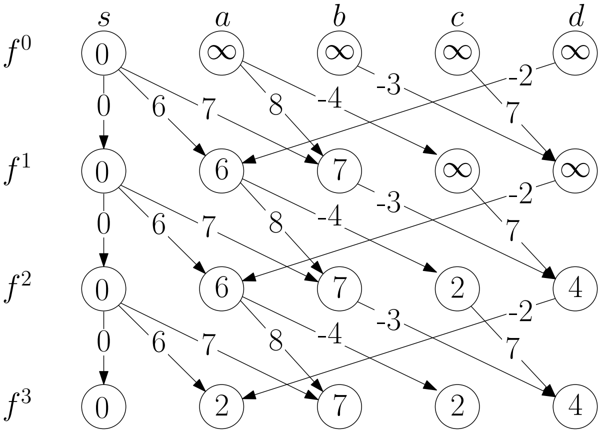
Dynamic Programming: Example



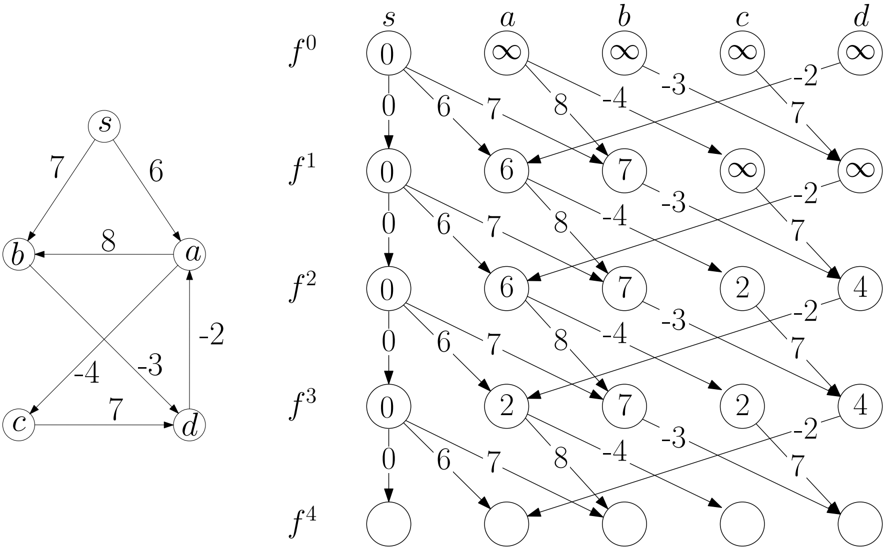


Dynamic Programming: Example

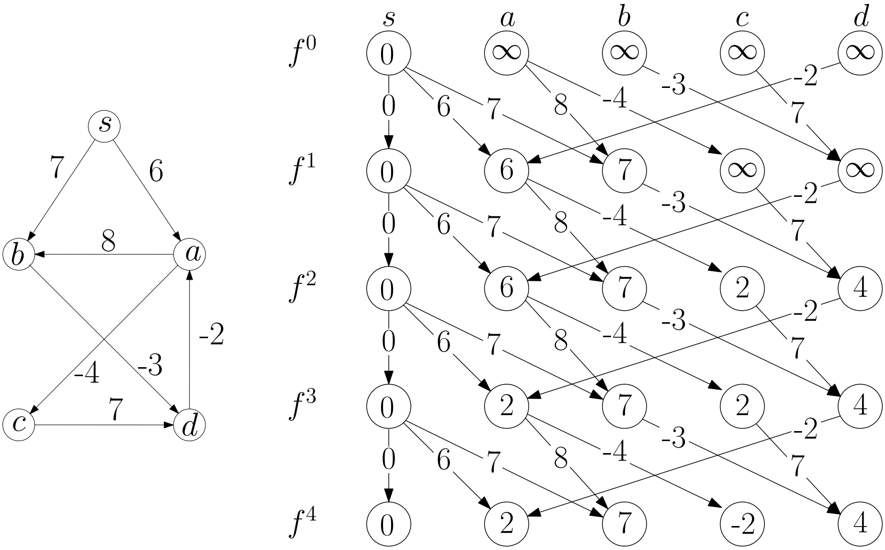




Dynamic Programming: Example



Dynamic Programming: Example



dynamic-programming(G, w, s)

```
f^0[s] ← 0 and f^0[v] ← ∞ for any v ∈ V \ {s}
for ℓ ← 1 to n − 1 do
    copy f^{ℓ−1} → f^ℓ
    for each (u, v) ∈ E do
        if f^{ℓ−1}[u] + w(u, v) < f^ℓ[v] then
            f^ℓ[v] ← f^{ℓ−1}[u] + w(u, v)
return (f^{n−1}[v])_{v∈V}
```

Obs. Assuming there are no negative cycles, then a shortest path contains at most $n-1$ edges

dynamic-programming(G, w, s)

$f^0[s] \leftarrow 0$ and $f^0[v] \leftarrow \infty$ for any $v \in V \setminus \{s\}$
for $\ell \leftarrow 1$ to $n-1$ **do**
    copy $f^{\ell-1} \rightarrow f^{\ell}$
    for each $(u, v) \in E$ **do**
        if $f^{\ell-1}[u] + w(u, v) < f^{\ell}[v]$ **then**
            $f^{\ell}[v] \leftarrow f^{\ell-1}[u] + w(u, v)$
return $(f^{n-1}[v])_{v \in V}$

Obs. Assuming there are no negative cycles, then a shortest path contains at most $n-1$ edges

Q: What if there are negative cycles?

Dynamic Programming With Negative Cycle Detection

dynamic-programming(G, w, s)

$f^0[s] \leftarrow 0$ and $f^0[v] \leftarrow \infty$ for any $v \in V \setminus \{s\}$
for $\ell \leftarrow 1$ to $n-1$ **do**
    copy $f^{\ell-1} \rightarrow f^{\ell}$
    for each $(u, v) \in E$ **do**
        if $f^{\ell-1}[u] + w(u, v) < f^{\ell}[v]$ **then**
            $f^{\ell}[v] \leftarrow f^{\ell-1}[u] + w(u, v)$
for each $(u, v) \in E$ **do**
    if $f^{n-1}[u] + w(u, v) < f^{n-1}[v]$ **then**
        report "negative cycle exists" and exit
return $(f^{n-1}[v])_{v \in V}$

Dynamic Programming with Better Space Usage

dynamic-programming(G, w, s)

```
f^old[s] ← 0 and f^old[v] ← ∞ for any v ∈ V \ {s}
for ℓ ← 1 to n − 1 do
    copy f^old → f^new
    for each (u, v) ∈ E do
        if f^old[u] + w(u, v) < f^new[v] then
            f^new[v] ← f^old[u] + w(u, v)
    copy f^new → f^old
return f^old
```

- f^ℓ only depends on $f^{\ell-1}$: only need 2 vectors

Dynamic Programming with Better Space Usage

dynamic-programming(G, w, s)

$f^{\text{old}}[s] \leftarrow 0$ and $f^{\text{old}}[v] \leftarrow \infty$ for any $v \in V \setminus \{s\}$
for $\ell \leftarrow 1$ to $n-1$ **do**
    copy $f^{\text{old}} \to f^{\text{new}}$
    for each $(u, v) \in E$ **do**
        if $f^{\text{old}}[u] + w(u, v) < f^{\text{new}}[v]$ **then**
            $f^{\text{new}}[v] \leftarrow f^{\text{old}}[u] + w(u, v)$
    copy $f^{\text{new}} \to f^{\text{old}}$
return f^{old}

- f^{ℓ} only depends on $f^{\ell-1}$: only need 2 vectors
- only need 1 vector!

Dynamic Programming with Better Space Usage

dynamic-programming(G, w, s)

```
f[s] ← 0 and f[v] ← ∞ for any v ∈ V \ {s}
for ℓ ← 1 to n − 1 do
    copy f → f
    for each (u, v) ∈ E do
        if f[u] + w(u, v) < f[v] then
            f[v] ← f[u] + w(u, v)
    copy f → f
return f
```

- f^ℓ only depends on $f^{\ell-1}$: only need 2 vectors
- only need 1 vector!

Dynamic Programming with Better Space Usage

dynamic-programming(G, w, s)

```
f[s] ← 0 and f[v] ← ∞ for any v ∈ V \ {s}
for ℓ ← 1 to n − 1 do
    for each (u, v) ∈ E do
        if f[u] + w(u, v) < f[v] then
            f[v] ← f[u] + w(u, v)
return f
```

- f^ℓ only depends on $f^{\ell-1}$: only need 2 vectors
- only need 1 vector!

Bellman-Ford Algorithm

Bellman-Ford(G, w, s)

```
f[s] ← 0 and f[v] ← ∞ for any v ∈ V \ {s}
for ℓ ← 1 to n − 1 do
    for each (u, v) ∈ E do
        if f[u] + w(u, v) < f[v] then
            f[v] ← f[u] + w(u, v)
return f
```

- f^ℓ only depends on $f^{\ell-1}$: only need 2 vectors
- only need 1 vector!

Bellman-Ford Algorithm

Bellman-Ford(G, w, s)

```
f[s] ← 0 and f[v] ← ∞ for any v ∈ V \ {s}
for ℓ ← 1 to n − 1 do
    for each (u, v) ∈ E do
        if f[u] + w(u, v) < f[v] then
            f[v] ← f[u] + w(u, v)
return f
```

- Issue: when we compute $f[u] + w(u, v)$, $f[u]$ may be changed since the end of last iteration

Bellman-Ford Algorithm

Bellman-Ford(G, w, s)

```
f[s] ← 0 and f[v] ← ∞ for any v ∈ V \ {s}
for ℓ ← 1 to n − 1 do
    for each (u, v) ∈ E do
        if f[u] + w(u, v) < f[v] then
            f[v] ← f[u] + w(u, v)
return f
```

- Issue: when we compute $f[u] + w(u, v)$, $f[u]$ may be changed since the end of last iteration
- This is OK: it can only "accelerate" the process!

Bellman-Ford Algorithm

Bellman-Ford(G, w, s)

```
f[s] ← 0 and f[v] ← ∞ for any v ∈ V \ {s}
for ℓ ← 1 to n − 1 do
    for each (u, v) ∈ E do
        if f[u] + w(u, v) < f[v] then
            f[v] ← f[u] + w(u, v)
return f
```

- Issue: when we compute $f[u] + w(u, v)$, $f[u]$ may be changed since the end of last iteration
- This is OK: it can only "accelerate" the process!
- After iteration ℓ, $f[v]$ is at most the length of the shortest path from s to v that uses at most ℓ edges

Bellman-Ford Algorithm

Bellman-Ford(G, w, s)

```
f[s] ← 0 and f[v] ← ∞ for any v ∈ V \ {s}
for ℓ ← 1 to n − 1 do
    for each (u, v) ∈ E do
        if f[u] + w(u, v) < f[v] then
            f[v] ← f[u] + w(u, v)
return f
```

- Issue: when we compute $f[u] + w(u, v)$, $f[u]$ may be changed since the end of last iteration
- This is OK: it can only "accelerate" the process!
- After iteration ℓ, $f[v]$ is at most the length of the shortest path from s to v that uses at most ℓ edges
- $f[v]$ is always the length of some path from s to v

Bellman-Ford Algorithm

Bellman-Ford(G, w, s)

1: $f[s] \leftarrow 0$ and $f[v] \leftarrow \infty$ for any $v \in V \setminus \{s\}$
2: **for** $\ell \leftarrow 1$ to $n-1$ **do**
3:     **for** each $(u, v) \in E$ **do**
4:         **if** $f[u] + w(u, v) < f[v]$ **then**
5:             $f[v] \leftarrow f[u] + w(u, v)$
6: **return** f

- After iteration ℓ, $f[v]$ is at most the length of the shortest path from s to v that uses at most ℓ edges
- $f[v]$ is always the length of some path from s to v

Bellman-Ford Algorithm

Bellman-Ford(G, w, s)

```
f[s] ← 0 and f[v] ← ∞ for any v ∈ V \ {s}
for ℓ ← 1 to n − 1 do
    for each (u, v) ∈ E do
        if f[u] + w(u, v) < f[v] then
            f[v] ← f[u] + w(u, v)
return f
```

- After iteration ℓ, $f[v]$ is at most the length of the shortest path from s to v that uses at most ℓ edges
- $f[v]$ is always the length of some path from s to v
- Assuming there are no negative cycles, after iteration $n-1$, $f[v] =$ length of shortest path from s to v

Bellman-Ford Algorithm

Bellman-Ford(G, w, s)

```
f[s] ← 0 and f[v] ← ∞ for any v ∈ V \ {s}
for ℓ ← 1 to n do
    updated ← false
    for each (u, v) ∈ E do
        if f[u] + w(u, v) < f[v] then
            f[v] ← f[u] + w(u, v)
            updated ← true
    if not updated, then return f
output "negative cycle exists"
```

Bellman-Ford Algorithm

Bellman-Ford(G, w, s)

$f[s] \leftarrow 0$ and $f[v] \leftarrow \infty$ for any $v \in V \setminus \{s\}$
for $\ell \leftarrow 1$ to n **do**
    $updated \leftarrow$ false
    for each $(u, v) \in E$ **do**
        if $f[u] + w(u, v) < f[v]$ **then**
            $f[v] \leftarrow f[u] + w(u, v)$, $\pi[v] \leftarrow u$
            $updated \leftarrow$ true
    if not $updated$, then return f
output "negative cycle exists"

- $\pi[v]$: the parent of v in the shortest path tree

Bellman-Ford Algorithm

Bellman-Ford(G, w, s)

$f[s] \leftarrow 0$ and $f[v] \leftarrow \infty$ for any $v \in V \setminus \{s\}$
for $\ell \leftarrow 1$ to n **do**
    $updated \leftarrow$ false
    for each $(u, v) \in E$ **do**
        if $f[u] + w(u, v) < f[v]$ **then**
            $f[v] \leftarrow f[u] + w(u, v)$, $\pi[v] \leftarrow u$
            $updated \leftarrow$ true
    if not $updated$, then return f
output "negative cycle exists"

- $\pi[v]$: the parent of v in the shortest path tree
- Running time $= O(nm)$

Outline

Summary of Shortest Path Algorithms we learned

algorithm	graph	weights	SS?	running time
Simple DP	DAG	$\mathbb{R}$	SS	$O(n+m)$
Dijkstra	U/D	$\mathbb{R}_{\geq 0}$	SS	$O(n \log n + m)$
Bellman-Ford	U/D	$\mathbb{R}$	SS	$O(nm)$

- DAG = directed acyclic graph U = undirected D = directed
- SS = single source AP = all pairs

Summary of Shortest Path Algorithms we learned

algorithm	graph	weights	SS?	running time
Simple DP	DAG	$\mathbb{R}$	SS	$O(n+m)$
Dijkstra	U/D	$\mathbb{R}_{\geq 0}$	SS	$O(n\log n+m)$
Bellman-Ford	U/D	$\mathbb{R}$	SS	$O(nm)$
Floyd-Warshall	U/D	$\mathbb{R}$	AP	$O(n^3)$

- DAG = directed acyclic graph U = undirected D = directed
- SS = single source AP = all pairs

All-Pair Shortest Paths

All Pair Shortest Paths

Input: directed graph $G = (V, E)$,

$w : E \to \mathbb{R}$ (can be negative)

Output: shortest path from u to v for every $u, v \in V$

All-Pair Shortest Paths

All Pair Shortest Paths

Input: directed graph $G = (V, E)$,

$w : E \to \mathbb{R}$ (can be negative)

Output: shortest path from u to v for every $u, v \in V$

```
for every starting point s ∈ V do
    run Bellman-Ford(G, w, s)
```

All-Pair Shortest Paths

All Pair Shortest Paths

Input: directed graph $G = (V, E)$,

$w : E \to \mathbb{R}$ (can be negative)

Output: shortest path from u to v for every $u, v \in V$

1: **for** every starting point $s \in V$ **do**
2: run Bellman-Ford(G, w, s)

- Running time $= O(n^2 m)$

Design a Dynamic Programming Algorithm

- It is convenient to assume $V = \{1, 2, 3, \cdots, n\}$

Design a Dynamic Programming Algorithm

- It is convenient to assume $V = \{1, 2, 3, \cdots, n\}$
- For simplicity, extend the w values to non-edges:

$$w(i,j) = \begin{cases} 0 & i = j \\ \text{weight of edge } (i,j) & i \neq j, (i,j) \in E \\ \infty & i \neq j, (i,j) \notin E \end{cases}$$

Design a Dynamic Programming Algorithm

- It is convenient to assume $V = \{1, 2, 3, \cdots, n\}$
- For simplicity, extend the w values to non-edges:

$$w(i,j) = \begin{cases} 0 & i = j \\ \text{weight of edge } (i,j) & i \neq j, (i,j) \in E \\ \infty & i \neq j, (i,j) \notin E \end{cases}$$

- For now assume there are no negative cycles

Design a Dynamic Programming Algorithm

- It is convenient to assume $V = \{1, 2, 3, \cdots, n\}$
- For simplicity, extend the w values to non-edges:

$$w(i,j) = \begin{cases} 0 & i = j \\ \text{weight of edge } (i,j) & i \neq j, (i,j) \in E \\ \infty & i \neq j, (i,j) \notin E \end{cases}$$

- For now assume there are no negative cycles

Cells for Floyd-Warshall Algorithm

Design a Dynamic Programming Algorithm

- It is convenient to assume $V = \{1, 2, 3, \cdots, n\}$
- For simplicity, extend the w values to non-edges:

$$w(i,j) = \begin{cases} 0 & i = j \\ \text{weight of edge } (i,j) & i \neq j, (i,j) \in E \\ \infty & i \neq j, (i,j) \notin E \end{cases}$$

- For now assume there are no negative cycles

Cells for Floyd-Warshall Algorithm

- First try: $f[i,j]$ is length of shortest path from i to j

Design a Dynamic Programming Algorithm

- It is convenient to assume $V = \{1, 2, 3, \cdots, n\}$
- For simplicity, extend the w values to non-edges:

$$w(i,j) = \begin{cases} 0 & i = j \\ \text{weight of edge } (i,j) & i \neq j, (i,j) \in E \\ \infty & i \neq j, (i,j) \notin E \end{cases}$$

- For now assume there are no negative cycles

Cells for Floyd-Warshall Algorithm

- First try: $f[i,j]$ is length of shortest path from i to j
- Issue: do not know in which order we compute $f[i,j]$'s

Design a Dynamic Programming Algorithm

- It is convenient to assume $V = \{1, 2, 3, \cdots, n\}$
- For simplicity, extend the w values to non-edges:

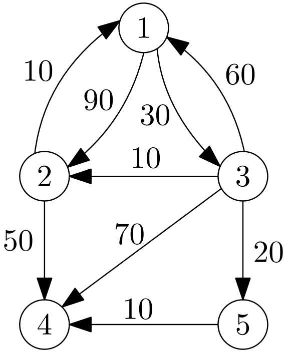
$$w(i,j) = \begin{cases} 0 & i = j \\ \text{weight of edge } (i,j) & i \neq j, (i,j) \in E \\ \infty & i \neq j, (i,j) \notin E \end{cases}$$

- For now assume there are no negative cycles

Cells for Floyd-Warshall Algorithm

- First try: $f[i,j]$ is length of shortest path from i to j
- Issue: do not know in which order we compute $f[i,j]$'s
- $f^k[i,j]$: length of shortest path from i to j that only uses vertices $\{1, 2, 3, \cdots, k\}$ as intermediate vertices

Example for Definition of $f^k[i,j]$'s



$$f^0[1,4] = \infty$$

$$f^1[1,4] = \infty$$

$$f^2[1,4] = 140 \qquad (1 \to 2 \to 4)$$

$$f^3[1,4] = 90 \qquad (1 \to 3 \to 2 \to 4)$$

$$f^4[1,4] = 90 \qquad (1 \to 3 \to 2 \to 4)$$

$$f^5[1,4] = 60 \qquad (1 \to 3 \to 5 \to 4)$$

$$w(i,j) = \begin{cases} 0 & i = j \\ \text{weight of edge } (i,j) & i \neq j, (i,j) \in E \\ \infty & i \neq j, (i,j) \notin E \end{cases}$$

- $f^k[i,j]$: length of shortest path from i to j that only uses vertices $\{1, 2, 3, \cdots, k\}$ as intermediate vertices

$$w(i,j) = \begin{cases} 0 & i = j \\ \text{weight of edge } (i,j) & i \neq j, (i,j) \in E \\ \infty & i \neq j, (i,j) \notin E \end{cases}$$

- $f^k[i,j]$: length of shortest path from i to j that only uses vertices $\{1,2,3,\cdots,k\}$ as intermediate vertices

$$f^k[i,j] = \begin{cases} & k = 0 \\ & k = 1, 2, \cdots, n \end{cases}$$

$$
w(i,j) = \begin{cases} 0 & i = j \\ \text{weight of edge } (i,j) & i \neq j, (i,j) \in E \\ \infty & i \neq j, (i,j) \notin E \end{cases}
$$

- $f^k[i,j]$: length of shortest path from i to j that only uses vertices $\{1, 2, 3, \cdots, k\}$ as intermediate vertices

$$
f^k[i,j] = \begin{cases} w(i,j) & k = 0 \\ & k = 1, 2, \cdots, n \end{cases}
$$

$$w(i,j) = \begin{cases} 0 & i = j \\ \text{weight of edge } (i,j) & i \neq j, (i,j) \in E \\ \infty & i \neq j, (i,j) \notin E \end{cases}$$

- $f^k[i,j]$: length of shortest path from i to j that only uses vertices $\{1, 2, 3, \cdots, k\}$ as intermediate vertices

$$f^k[i,j] = \begin{cases} w(i,j) & k = 0 \\ \min \left\{ \begin{array}{l} \\ \\ \end{array} \right. & k = 1, 2, \cdots, n \end{cases}$$

$$w(i,j) = \begin{cases} 0 & i = j \\ \text{weight of edge } (i,j) & i \neq j, (i,j) \in E \\ \infty & i \neq j, (i,j) \notin E \end{cases}$$

- $f^k[i,j]$: length of shortest path from i to j that only uses vertices $\{1,2,3,\cdots,k\}$ as intermediate vertices

$$f^k[i,j] = \begin{cases} w(i,j) & k = 0 \\ \min \left\{ \begin{array}{l} f^{k-1}[i,j] \end{array} \right. & k = 1,2,\cdots,n \end{cases}$$

$$w(i,j) = \begin{cases} 0 & i = j \\ \text{weight of edge } (i,j) & i \neq j, (i,j) \in E \\ \infty & i \neq j, (i,j) \notin E \end{cases}$$

- $f^k[i,j]$: length of shortest path from i to j that only uses vertices $\{1, 2, 3, \cdots, k\}$ as intermediate vertices

$$f^k[i,j] = \begin{cases} w(i,j) & k = 0 \\ \min \begin{Bmatrix} f^{k-1}[i,j] \\ f^{k-1}[i,k] + f^{k-1}[k,j] \end{Bmatrix} & k = 1, 2, \cdots, n \end{cases}$$

Floyd-Warshall(G, w)

```
f^0 ← w
for k ← 1 to n do
    copy f^{k-1} → f^k
    for i ← 1 to n do
        for j ← 1 to n do
            if f^{k-1}[i,k] + f^{k-1}[k,j] < f^k[i,j] then
                f^k[i,j] ← f^{k-1}[i,k] + f^{k-1}[k,j]
```

Floyd-Warshall(G, w)

```
f^old ← w
for k ← 1 to n do
    copy f^old → f^new
    for i ← 1 to n do
        for j ← 1 to n do
            if f^old[i,k] + f^old[k,j] < f^new[i,j] then
                f^new[i,j] ← f^old[i,k] + f^old[k,j]
```

Floyd-Warshall(G, w)

$f^{\text{old}} \leftarrow w$
for $k \leftarrow 1$ to n **do**
    copy $f^{\text{old}} \rightarrow f^{\text{new}}$
    for $i \leftarrow 1$ to n **do**
        for $j \leftarrow 1$ to n **do**
            if $f^{\text{old}}[i,k] + f^{\text{old}}[k,j] < f^{\text{new}}[i,j]$ **then**
                $f^{\text{new}}[i,j] \leftarrow f^{\text{old}}[i,k] + f^{\text{old}}[k,j]$

Floyd-Warshall(G, w)

$f \leftarrow w$
for $k \leftarrow 1$ to n **do**
    copy $f \rightarrow f$
    for $i \leftarrow 1$ to n **do**
        for $j \leftarrow 1$ to n **do**
            if $f[i,k] + f[k,j] < f[i,j]$ **then**
                $f[i,j] \leftarrow f[i,k] + f[k,j]$

Floyd-Warshall(G, w)

```
f ← w
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            if f[i,k] + f[k,j] < f[i,j] then
                f[i,j] ← f[i,k] + f[k,j]
```

Floyd-Warshall(G, w)

$f \leftarrow w$
for $k \leftarrow 1$ to n **do**
  for $i \leftarrow 1$ to n **do**
    for $j \leftarrow 1$ to n **do**
      if $f[i,k] + f[k,j] < f[i,j]$ **then**
        $f[i,j] \leftarrow f[i,k] + f[k,j]$

Lemma Assume there are no negative cycles in G. After iteration k, for $i, j \in V$, $f[i,j]$ is exactly the length of shortest path from i to j that only uses vertices in $\{1, 2, 3, \cdots, k\}$ as intermediate vertices.

Floyd-Warshall(G, w)

$f \leftarrow w$
for $k \leftarrow 1$ to n **do**
  for $i \leftarrow 1$ to n **do**
    for $j \leftarrow 1$ to n **do**
      if $f[i,k] + f[k,j] < f[i,j]$ **then**
        $f[i,j] \leftarrow f[i,k] + f[k,j]$

Lemma Assume there are no negative cycles in G. After iteration k, for $i, j \in V$, $f[i,j]$ is exactly the length of shortest path from i to j that only uses vertices in $\{1, 2, 3, \cdots, k\}$ as intermediate vertices.

- Running time $= O(n^3)$.

Recovering Shortest Paths

Floyd-Warshall(G, w)

$f \leftarrow w$, $\pi[i,j] \leftarrow \perp$ for every $i, j \in V$
for $k \leftarrow 1$ to n **do**
  for $i \leftarrow 1$ to n **do**
    for $j \leftarrow 1$ to n **do**
      if $f[i,k] + f[k,j] < f[i,j]$ **then**
        $f[i,j] \leftarrow f[i,k] + f[k,j]$, $\pi[i,j] \leftarrow k$

Recovering Shortest Paths

Floyd-Warshall(G, w)

1: $f \leftarrow w$, $\pi[i,j] \leftarrow \perp$ for every $i, j \in V$
2: **for** $k \leftarrow 1$ to n **do**
3:     **for** $i \leftarrow 1$ to n **do**
4:         **for** $j \leftarrow 1$ to n **do**
5:             **if** $f[i,k] + f[k,j] < f[i,j]$ **then**
6:                 $f[i,j] \leftarrow f[i,k] + f[k,j]$, $\pi[i,j] \leftarrow k$

print-path(i, j)

1: **if** $\pi[i,j] = \perp$ **then** then
2:     **if** $i \neq j$ **then** print$(i, \text{“,”})$
3: **else**
4:     print-path$(i, \pi[i,j])$, print-path$(\pi[i,j], j)$

Detecting Negative Cycles

Floyd-Warshall(G, w)

$f \leftarrow w$, $\pi[i,j] \leftarrow \perp$ for every $i, j \in V$
for $k \leftarrow 1$ to n **do**
    for $i \leftarrow 1$ to n **do**
        for $j \leftarrow 1$ to n **do**
            if $f[i,k] + f[k,j] < f[i,j]$ **then**
                $f[i,j] \leftarrow f[i,k] + f[k,j]$, $\pi[i,j] \leftarrow k$

Detecting Negative Cycles

Floyd-Warshall(G, w)

$f \leftarrow w$, $\pi[i,j] \leftarrow \perp$ for every $i, j \in V$
for $k \leftarrow 1$ to n **do**
    for $i \leftarrow 1$ to n **do**
        for $j \leftarrow 1$ to n **do**
            if $f[i,k] + f[k,j] < f[i,j]$ **then**
                $f[i,j] \leftarrow f[i,k] + f[k,j]$, $\pi[i,j] \leftarrow k$
for $k \leftarrow 1$ to n **do**
    for $i \leftarrow 1$ to n **do**
        for $j \leftarrow 1$ to n **do**
            if $f[i,k] + f[k,j] < f[i,j]$ **then**
                report “negative cycle exists” and exit

Summary of Shortest Path Algorithms

algorithm	graph	weights	SS?	running time
Simple DP	DAG	$\mathbb{R}$	SS	$O(n+m)$
Dijkstra	U/D	$\mathbb{R}_{\geq 0}$	SS	$O(n \log n + m)$
Bellman-Ford	U/D	$\mathbb{R}$	SS	$O(nm)$
Floyd-Warshall	U/D	$\mathbb{R}$	AP	$O(n^3)$

- DAG = directed acyclic graph U = undirected D = directed
- SS = single source AP = all pairs