Design and Analysis of Algorithms

Presented by Dr. Li Ning

Shenzhen Institutes of Advanced Technology, Chinese Academy of Science
Shenzhen, China

Algorithm with Graphs

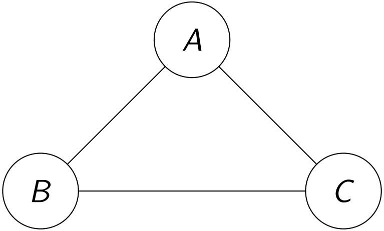
Graph: The Abstract Data Structure

Graph

Graph: a set of **nodes** connected by the **edges**.

$G = (V, E)$

- V: the set of nodes
  - A, B, and C
- E: the set of edges
  - (A, B), (B, C), and (C, A)



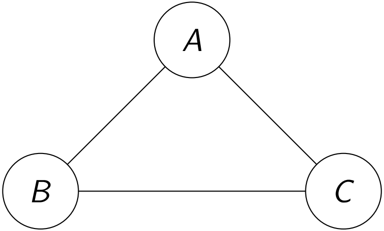
Graph

The Undirected Edges

- $(A, B) = (B, A)$
- $(B, C) = (C, B)$
- $(C, A) = (A, C)$

The Cycle

- $A - B - C - A$
- (A, B), (B, C), (C, A)

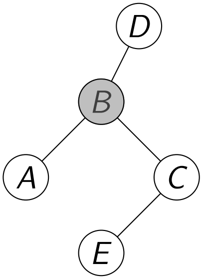


Tree

Tree: The (connected) graph containing no cycle

Take any node as the root, e.g. B

- root B has no parent

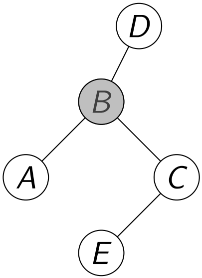


Tree

Tree: The (connected) graph containing no cycle

Take any node as the root, e.g. B

- root B has no parent
- B's children
  - A
    - parent: B
  - C
    - parent: B
    - child: E
  - D
    - parent: B

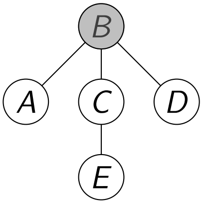


Tree

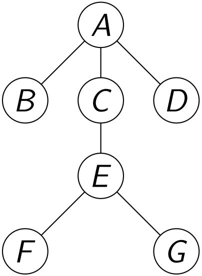
Tree: The (connected) graph containing no cycle

Take any node as the root, e.g. B

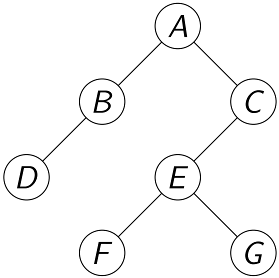
- root B has no parent
- B's children
    - A
        - parent: B
    - C
        - parent: B
        - child: E
    - D
        - parent: B



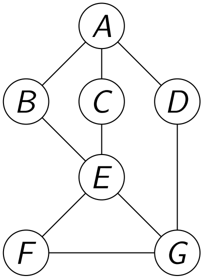
Examples



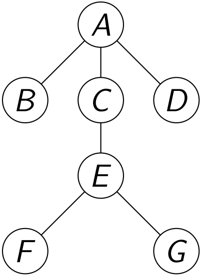
Examples

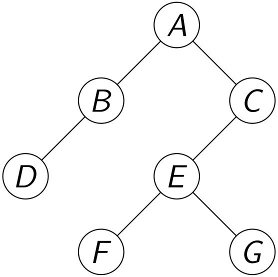


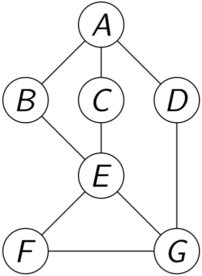
Examples



Examples





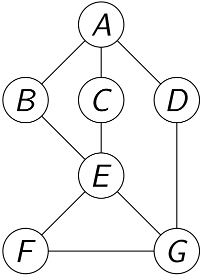


Graph in Code

Listing 1: graph.py

```
G = {
    "A" : ["B", "C", "D"],
    "B" : ["A", "E"],
    "C" : ["A", "E"],
    "D" : ["A", "G"],
    "E" : ["B", "C", "F", "G"],
    "F" : ["E", "G"],
    "G" : ["D", "E", "F"],
}

print("Node D’s neighbors:")
for n in G["D"] :
    print(n)
```

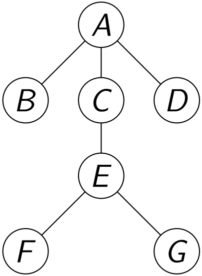



```
>> python graph.py
Node D’s neighbors:
A
G
```

Tree in Code

Listing 2: tree.py

```
T = {
  "A" : (None, ["B", "C", "D"]),
  "B" : ("A", []),
  "C" : ("A", ["E",]),
  "D" : ("A", []),
  "E" : ("C", ["F", "G"]),
  "F" : ("E", []),
  "G" : ("E", []),
}
print("Node E's parent:",
   T["E"][0])
print("Node E's children:")
for c in T["E"][1] :
    print(c)
```




```
>> python tree.py
Node E's parent: C
Node E's children:
F
G
```

Graph Traversal

Visit The Nodes

Given a graph $G = (V, E)$.

Traversal: starting at node v, visit all the nodes of the graph in a sequence.

Visit The Nodes

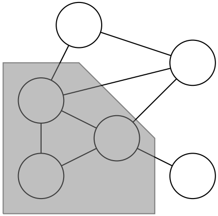
Given a graph $G = (V, E)$.

Traversal: starting at node v, visit all the nodes of the graph in a sequence.

Idea: select the next node to visit among the neighbors of the visited nodes.

Select The Next Node

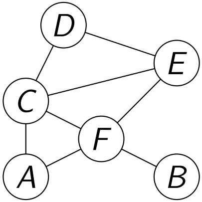
Which neighbor to visit next?



DFS: Depth First Search

Depth-First: go as deep/far as possible.

- visit a node v
  - for each u, s.t. $(v, u) \in E$
    - DFS(u) if not visited

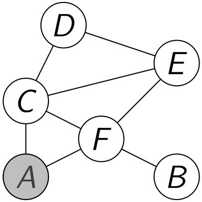


start at A

DFS: Depth First Search

Depth-First: go as deep/far as possible.

- visit a node v
  - for each u, s.t. $(v, u) \in E$
    - DFS(u) if not visited

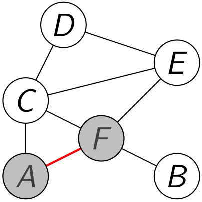


start at A

DFS: Depth First Search

Depth-First: go as deep/far as possible.

- visit a node v
  - for each u, s.t. $(v, u) \in E$
    - DFS(u) if not visited

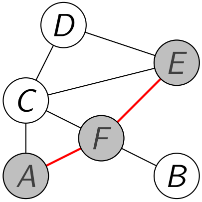


start at A

DFS: Depth First Search

Depth-First: go as deep/far as possible.

- visit a node v
  - for each u, s.t. $(v, u) \in E$
    - DFS(u) if not visited

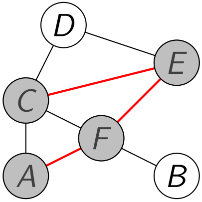


start at A

DFS: Depth First Search

Depth-First: go as deep/far as possible.

- visit a node v
  - for each u, s.t. $(v, u) \in E$
    - DFS(u) if not visited

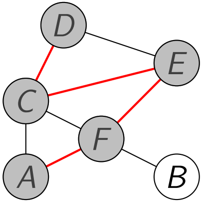


start at A

DFS: Depth First Search

Depth-First: go as deep/far as possible.

- visit a node v
  - for each u, s.t. $(v, u) \in E$
    - DFS(u) if not visited

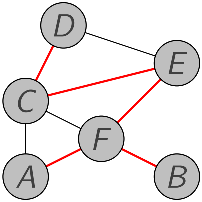


start at A

DFS: Depth First Search

Depth-First: go as deep/far as possible.

- visit a node v
  - for each u, s.t. $(v, u) \in E$
    - DFS(u) if not visited

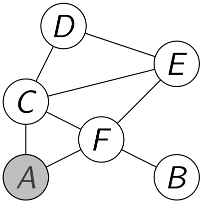


start at A

BFS: Breadth First Search

Breadth-First: explore the nodes layer by layer.

- Starting at the root r.
- Initialize $d = 0$.
- While there is node not visited:
  - visit the nodes that has distance d to the root.
  - $d = d + 1$

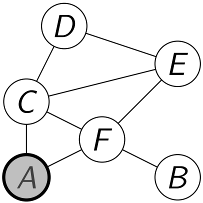


$$q = [A,]$$

BFS: Breadth First Search

Breadth-First: explore the nodes layer by layer.

- Starting at the root r.
- Initialize $d = 0$.
- While there is node not visited:
  - visit the nodes that has distance d to the root.
  - $d = d + 1$

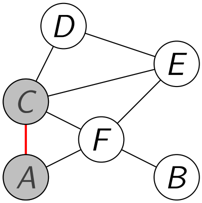


$q = [\ ]$

BFS: Breadth First Search

Breadth-First: explore the nodes layer by layer.

- Starting at the root r.
- Initialize $d = 0$.
- While there is node not visited:
  - visit the nodes that has distance d to the root.
  - $d = d + 1$

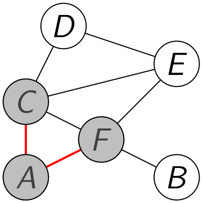


$$q = [C,]$$

BFS: Breadth First Search

Breadth-First: explore the nodes layer by layer.

- Starting at the root r.
- Initialize $d = 0$.
- While there is node not visited:
  - visit the nodes that has distance d to the root.
  - $d = d + 1$

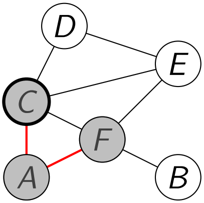


$q = [C, F]$

BFS: Breadth First Search

Breadth-First: explore the nodes layer by layer.

- Starting at the root r.
- Initialize $d = 0$.
- While there is node not visited:
  - visit the nodes that has distance d to the root.
  - $d = d + 1$

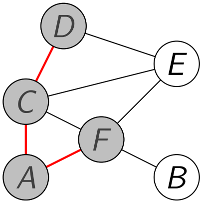


$q = [F]$

BFS: Breadth First Search

Breadth-First: explore the nodes layer by layer.

- Starting at the root r.
- Initialize $d = 0$.
- While there is node not visited:
  - visit the nodes that has distance d to the root.
  - $d = d + 1$

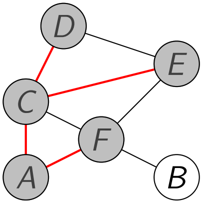


$q = [F, D]$

BFS: Breadth First Search

Breadth-First: explore the nodes layer by layer.

- Starting at the root r.
- Initialize $d = 0$.
- While there is node not visited:
    - visit the nodes that has distance d to the root.
    - $d = d + 1$

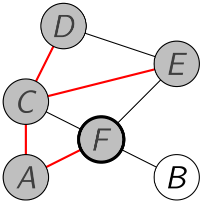


$q = [F, D, E]$

BFS: Breadth First Search

Breadth-First: explore the nodes layer by layer.

- Starting at the root r.
- Initialize $d = 0$.
- While there is node not visited:
    - visit the nodes that has distance d to the root.
    - $d = d + 1$

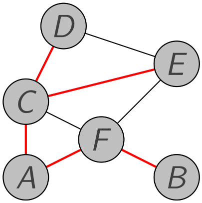


$q = [D, E]$

BFS: Breadth First Search

Breadth-First: explore the nodes layer by layer.

- Starting at the root r.
- Initialize $d = 0$.
- While there is node not visited:
  - visit the nodes that has distance d to the root.
  - $d = d + 1$

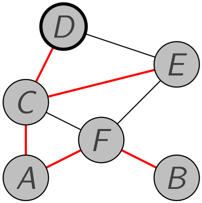


$q = [D, E, B]$

BFS: Breadth First Search

Breadth-First: explore the nodes layer by layer.

- Starting at the root r.
- Initialize $d = 0$.
- While there is node not visited:
    - visit the nodes that has distance d to the root.
    - $d = d + 1$

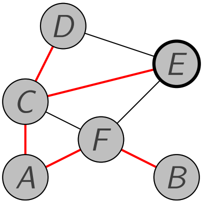


$q = [E, B]$

BFS: Breadth First Search

Breadth-First: explore the nodes layer by layer.

- Starting at the root r.
- Initialize $d = 0$.
- While there is node not visited:
    - visit the nodes that has distance d to the root.
    - $d = d + 1$

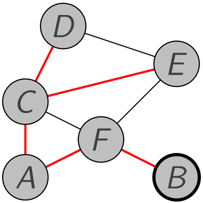


$q = [B]$

BFS: Breadth First Search

Breadth-First: explore the nodes layer by layer.

- Starting at the root r.
- Initialize $d = 0$.
- While there is node not visited:
  - visit the nodes that has distance d to the root.
  - $d = d + 1$



$q = [\,]$

Complexity: $O(|V| + |E|)$

For each node v

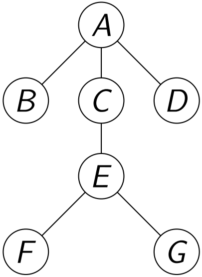
- 1 : visit
- ? : check if v is visited

- $O(|V|)$ visits
- $O(|E|)$ checks

Traversal on Trees

Staring at root, visit the nodes in a sequence.

- **Idea 1**: visit the node before visiting its children
- **Idea 2**: visit the node after visiting its children



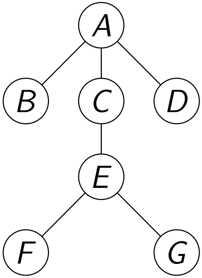
Preorder Traversal on Trees

Preorder traversal: visit the node before visiting its children.

Algorithm: Preorder(v)

visit node v;
for *v's each child u* **do**
    Preorder(u);
end

Preorder(A): A, B, C, E, F, G, D



Preorder Traversal on Trees

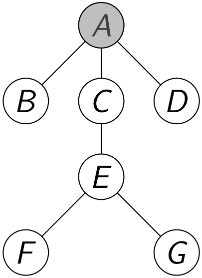
Preorder traversal: visit the node before visiting its children.

Algorithm: Preorder(v)

---

visit node v;
for *v's each child u* **do**
  Preorder(u);
end

Preorder(A): A, B, C, E, F, G, D



Preorder Traversal on Trees

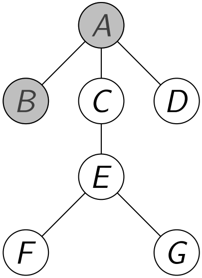
Preorder traversal: visit the node before visiting its children.

Algorithm: Preorder(v)

---

visit node v;
for *v's each child u* **do**
  Preorder(u);
end

Preorder(A): A, B, C, E, F, G, D



Preorder Traversal on Trees

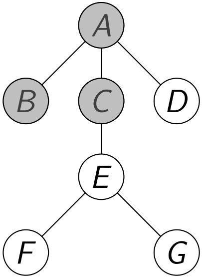
Preorder traversal: visit the node before visiting its children.

Algorithm: Preorder(v)

---

visit node v;
for *v's each child u* **do**
  Preorder(u);
end

Preorder(A): A, B, C, E, F, G, D



Preorder Traversal on Trees

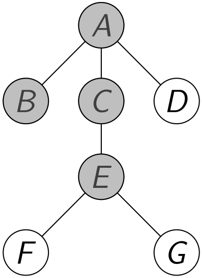
Preorder traversal: visit the node before visiting its children.

Algorithm: Preorder(v)

---

visit node v;
for *v's each child u* **do**
  Preorder(u);
end

Preorder(A): A, B, C, E, F, G, D



Preorder Traversal on Trees

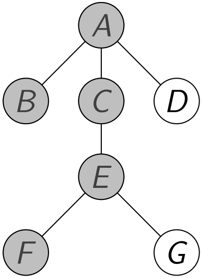
Preorder traversal: visit the node before visiting its children.

Algorithm: Preorder(v)

---

visit node v;
for *v's each child u* **do**
  Preorder(u);
end

Preorder(A): A, B, C, E, F, G, D



Preorder Traversal on Trees

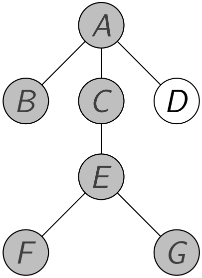
Preorder traversal: visit the node before visiting its children.

Algorithm: Preorder(v)

---

visit node v;
for *v's each child u* **do**
| Preorder(u);
end

Preorder(A): A, B, C, E, F, G, D



Preorder Traversal on Trees

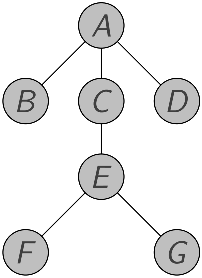
Preorder traversal: visit the node before visiting its children.

Algorithm: Preorder(v)

---

visit node v;
for *v's each child u* **do**
    Preorder(u);
end

Preorder(A): A, B, C, E, F, G, D



Postorder Traversal on Trees

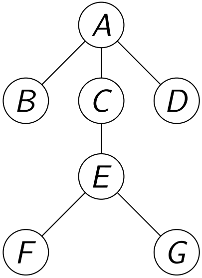
Postorder traversal: visit the node after visiting its children.

Algorithm: Postorder(v)

---

for *v's each child u* **do**
  | Postorder(u);
end
visit node v;

Postorder(A): B, F, G, E, C, D, A



Postorder Traversal on Trees

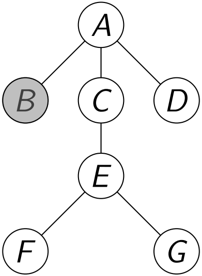
Postorder traversal: visit the node after visiting its children.

Algorithm: Postorder(v)

---

for *v's each child u* **do**
  | Postorder(u);
end
visit node v;

Postorder(A): B, F, G, E, C, D, A



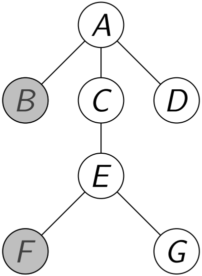
Postorder Traversal on Trees

Postorder traversal: visit the node after visiting its children.

Algorithm: Postorder(v)

---

for *v's each child u* **do**
    Postorder(u);
end
visit node v;



Postorder(A): B, F, G, E, C, D, A

Postorder Traversal on Trees

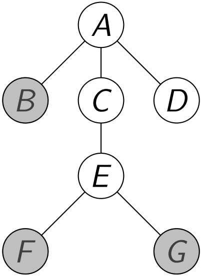
Postorder traversal: visit the node after visiting its children.

Algorithm: Postorder(v)

---

for *v's each child u* **do**
  | Postorder(u);
end
visit node v;

Postorder(A): B, F, G, E, C, D, A



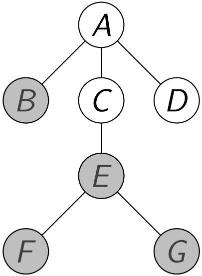
Postorder Traversal on Trees

Postorder traversal: visit the node after visiting its children.

Algorithm: Postorder(v)

for *v's each child u* **do**
  Postorder(u);
end
visit node v;

Postorder(A): B, F, G, E, C, D, A



Postorder Traversal on Trees

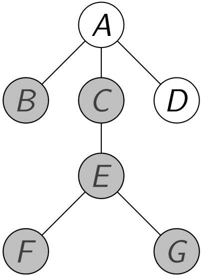
Postorder traversal: visit the node after visiting its children.

Algorithm: Postorder(v)

---

for *v's each child u* **do**
  Postorder(u);
end
visit node v;

Postorder(A): B, F, G, E, C, D, A



Postorder Traversal on Trees

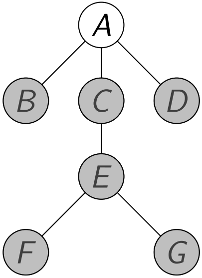
Postorder traversal: visit the node after visiting its children.

Algorithm: Postorder(v)

---

for *v's each child u* **do**
    Postorder(u);
end
visit node v;

Postorder(A): B, F, G, E, C, D, A



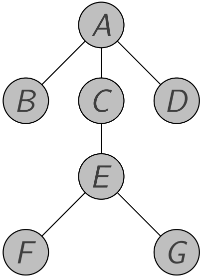
Postorder Traversal on Trees

Postorder traversal: visit the node after visiting its children.

Algorithm: Postorder(v)

for *v's each child u* **do**
    Postorder(u);
end
visit node v;

Postorder(A): B, F, G, E, C, D, A



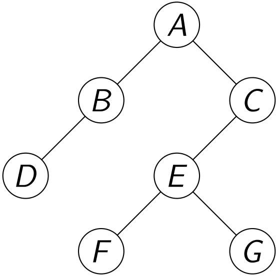
Inorder Traversal on Binary Trees

Inorder traversal: visit the node between the visiting of the left child and the right child.

Algorithm: Inorder(v)

Inorder(u' left child);
visit node v;
Inorder(u' right child);

Inorder(A): D, B, A, F, E, G, C

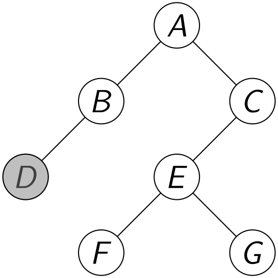


Inorder Traversal on Binary Trees

Inorder traversal: visit the node between the visiting of the left child and the right child.

Algorithm: Inorder(v)

Inorder(u' left child);
visit node v;
Inorder(u' right child);



Inorder(A): D, B, A, F, E, G, C

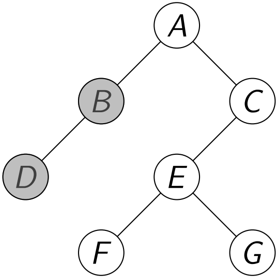
Inorder Traversal on Binary Trees

Inorder traversal: visit the node between the visiting of the left child and the right child.

Algorithm: Inorder(v)

Inorder(u' left child);
visit node v;
Inorder(u' right child);

Inorder(A): D, B, A, F, E, G, C



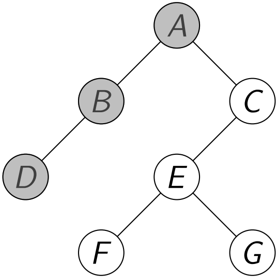
Inorder Traversal on Binary Trees

Inorder traversal: visit the node between the visiting of the left child and the right child.

Algorithm: Inorder(v)

Inorder(u' left child);
visit node v;
Inorder(u' right child);

Inorder(A): D, B, A, F, E, G, C



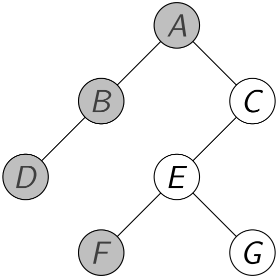
Inorder Traversal on Binary Trees

Inorder traversal: visit the node between the visiting of the left child and the right child.

Algorithm: Inorder(v)

Inorder(u' left child);
visit node v;
Inorder(u' right child);

Inorder(A): D, B, A, F, E, G, C



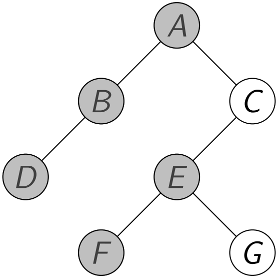
Inorder Traversal on Binary Trees

Inorder traversal: visit the node between the visiting of the left child and the right child.

Algorithm: Inorder(v)

Inorder(u' left child);
visit node v;
Inorder(u' right child);

Inorder(A): D, B, A, F, E, G, C



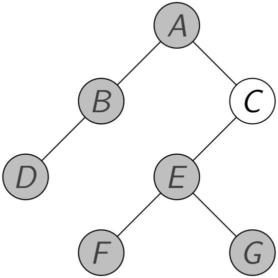
Inorder Traversal on Binary Trees

Inorder traversal: visit the node between the visiting of the left child and the right child.

Algorithm: Inorder(v)

Inorder(u' left child);
visit node v;
Inorder(u' right child);

Inorder(A): D, B, A, F, E, G, C



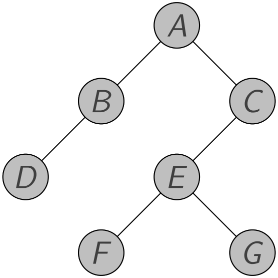
Inorder Traversal on Binary Trees

Inorder traversal: visit the node between the visiting of the left child and the right child.

Algorithm: Inorder(v)

Inorder(u' left child);
visit node v;
Inorder(u' right child);

Inorder(A): D, B, A, F, E, G, C

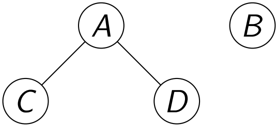


Connectivity

Connected graph: all nodes can be visited through the traversal.

Traversal on graph?

For each node v in $V = [A, B, C, D]$

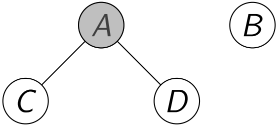


- If v is not visited
  - Traversal starting at v

Connectivity

Connected graph: all nodes can be visited through the traversal.

Traversal on graph?



For each node v in $V = [A, B, C, D]$

- If v is not visited
  - Traversal starting at v

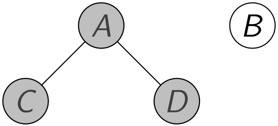
Connectivity

Connected graph: all nodes can be visited through the traversal.

Traversal on graph?

For each node v in $V = [A, B, C, D]$

- If v is not visited
    - Traversal starting at v



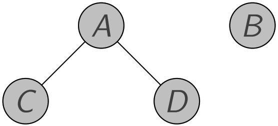
Connectivity

Connected graph: all nodes can be visited through the traversal.

Traversal on graph?

For each node v in $V = [A, B, C, D]$

- If v is not visited
  - Traversal starting at v



Forest and Tree

Forest: the graph containing no cycle.

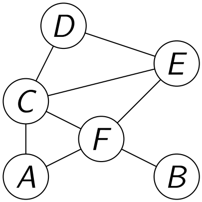
Tree: the **connected** graph containing no cycle.

Shortest Path

Path

Path: a sequence consecutively linked nodes.

- A path: [A, C, E, F]
- Not a path: [A, C, E, B]
  - E is not linked to B.



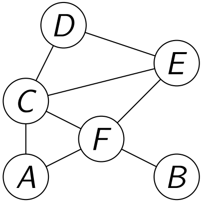
Length: number of the edges in a path.

Shortest Path

Connected graph: for any pair of nodes u and v, there is a path from u to v.

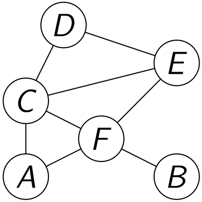
Given two nodes u and v, find the path of minimum length, that connects u and v.

- A to E
  - A, C, E
  - A, F, E
- D to B
  - D, C, F, B
  - D, E, F, B



Shortest Path

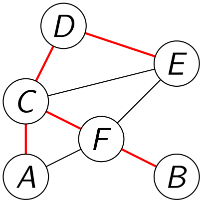
Find the length of the shortest path from D to B.



Shortest Path

Find the length of the shortest path from D to B.

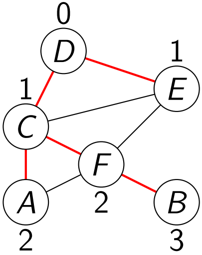
- Perform BFS on the graph, starting at D.



Shortest Path

Find the length of the shortest path from D to B.

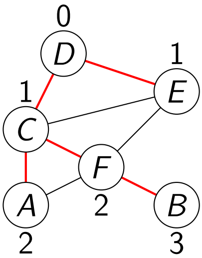
- Perform BFS on the graph, starting at D.
- **Layer** of node v: $1+$ layer of the node that add v to the explored list.



Shortest Path

Find the length of the shortest path from D to B.

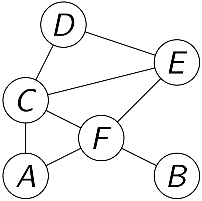
- Perform BFS on the graph, starting at D.
- **Layer** of node v: $1+$ layer of the node that add v to the explored list.
- Layer of node $B =$ length of the shortest path from D to B.



Weighted Edges

A weight function w maps each edge to a real number.

$$w(v, u) \rightarrow \mathbb{R}^+$$

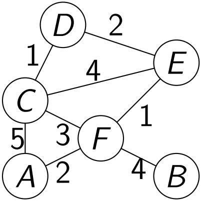


Weighted Edges

A weight function w maps each edge to a real number.

$$w(v, u) \to \mathbb{R}^+$$

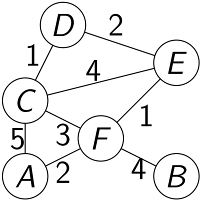
$$w(A, C) = 5;\ w(A, F) = 2$$
$$w(C, D) = 1;\ w(C, E) = 4$$
$$w(C, F) = 3;\ w(D, E) = 2$$
$$w(E, F) = 1;\ w(B, F) = 4$$



Weighted Path

Given the weight of each edge, the (weighted) length of a path is the sum of the weights of the edges in the path.

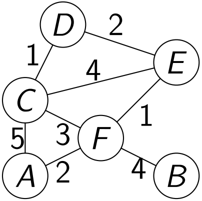
Shortest path: Given a weighted graph and a pair of nodes v and u, find the path of minimum (weighted) length, that connects v and u.



Weighted Path

Shortest path: Given a weighted graph and a pair of nodes v and u, find the path of minimum (weighted) length, that connects v and u.

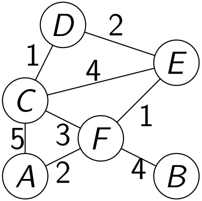
Path $[A, F, E]$ has length $2 + 1 = 3$.



Shortest Path

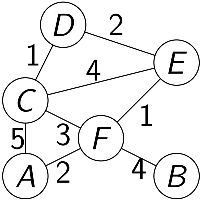
Idea: To calculate the length of the shortest path from s to v, consider an edge (u, v),

- $d(s, v) \leq d(s, u) + w(u, v)$
- $d(s, u)$?



Shortest Path

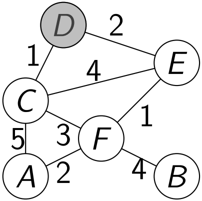
Find the length of the shortest path from D to B.



Shortest Path

Find the length of the shortest path from D to B.

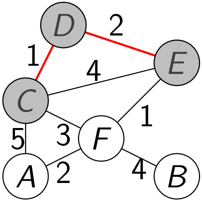
- **Initially**, we know $d(D, D) = 0$



Shortest Path

Find the length of the shortest path from D to B.

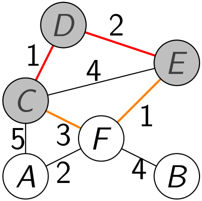
- **Initially**, we know $d(D, D) = 0$
- among all the nodes that $d(D, v)$ is not known
    - $d(D, C) = d(D, D) + w(D, C)$
    - $d(D, E) = d(D, D) + w(D, E)$



Shortest Path

Find the length of the shortest path from D to B.

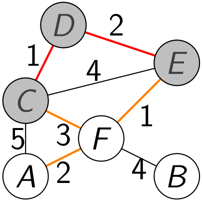
- **Initially**, we know $d(D, D) = 0$
- among all the nodes that $d(D, v)$ is not known
  - $d(D, C) = d(D, D) + w(D, C)$
  - $d(D, E) = d(D, D) + w(D, E)$
- $d(D, F)$
  - $d(D, C) + w(C, F)$
  - $d(D, E) + w(E, F)$
  - $d(D, D) +\ ?$



Shortest Path

Find the length of the shortest path from D to B.

- **Initially**, we know $d(D, D) = 0$
- among all the nodes that $d(D, v)$ is not known
  - $d(D, C) = d(D, D) + w(D, C)$
  - $d(D, E) = d(D, D) + w(D, E)$
- $d(D, F)$
  - $d(D, C) + w(C, F)$
  - $d(D, E) + w(E, F)$
  - $d(D, D) +\ ?$
  - $d(D, A) + w(A, F)$?



Dijkstra's Algorithm

$d(s, v)$ length of the shortest path from s to v.

$\delta(s, v)$ length of the shortest path from s to v, **through** S.

To calculate $d(s, v)$ for all nodes $v \in V$.

Dijkstra's Algorithm

$d(s, v)$ length of the shortest path from s to v.

$\delta(s, v)$ length of the shortest path from s to v, **through** S.

To calculate $d(s, v)$ for all nodes $v \in V$.

- Let S be the nodes, s.t.
  - $d(s, u)$ is known, for all $u \in S$
  - $d(s, v)$ for all $v \in V$ are known if $S = V$

Dijkstra's Algorithm

$d(s, v)$ length of the shortest path from s to v.

$\delta(s, v)$ length of the shortest path from s to v, **through** S.

To calculate $d(s, v)$ for all nodes $v \in V$.

- Let S be the nodes, s.t.
  - $d(s, u)$ is known, for all $u \in S$
  - $d(s, v)$ for all $v \in V$ are known if $S = V$
- **initially**, $S = \{s\}$
  - $d(s, s) = 0$
  - $\delta(s, v) = w(s, v)$ if $(s, v) \in E$
  - $\delta(s, v) = \infty$ if $(s, v) \notin E$

Dijkstra's Algorithm

$d(s, v)$ length of the shortest path from s to v.

$\delta(s, v)$ length of the shortest path from s to v, **through** S.

To calculate $d(s, v)$ for all nodes $v \in V$.

- Let S be the nodes, s.t.
  - $d(s, u)$ is known, for all $u \in S$
  - $d(s, v)$ for all $v \in V$ are known if $S = V$
- **initially**, $S = \{s\}$
  - $d(s, s) = 0$
  - $\delta(s, v) = w(s, v)$ if $(s, v) \in E$
  - $\delta(s, v) = \infty$ if $(s, v) \notin E$
- **while** $S \neq V$
  - $u := \arg\min_{u \in V \setminus S} \delta(s, u)$
  - $d(s, u) = \delta(s, u)$

Dijkstra's Algorithm

$u := \arg\min_{u \in V \setminus S} \delta(s, u) \Rightarrow d(s, u) = \delta(s, u)$

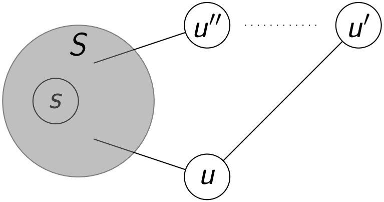
Proof: Assume that $d(s, u) < \delta(s, u)$.

Dijkstra's Algorithm

$u := \arg\min_{u \in V \setminus S} \delta(s, u) \Rightarrow d(s, u) = \delta(s, u)$

Proof: Assume that $d(s, u) < \delta(s, u)$.

- $\exists u' \in V \setminus S,\ d(s, u') + w(u', u) < \delta(s, u)$

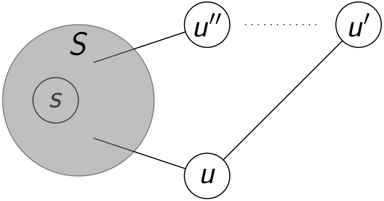


Dijkstra's Algorithm

$u := \arg\min_{u \in V \setminus S} \delta(s, u) \Rightarrow d(s, u) = \delta(s, u)$

Proof: Assume that $d(s, u) < \delta(s, u)$.

- $\exists u' \in V \setminus S,\ d(s, u') + w(u', u) < \delta(s, u)$

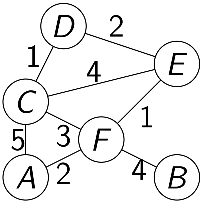


- $\delta(s, u'') = d(s, u'') \leq d(s, u') < \delta(s, u)$
- **Contradiction**!

Dijkstra's Algorithm

Algorithm: Dijkstra(G, s)

$d(s, v) = \infty, \forall v \in V$;
$S = \emptyset$;
$\delta(s, v) = \infty, \forall v \neq s,\ \delta(s, s) = 0$;
while $S \neq V$ **do**
  $u = \arg\min_{u \in V \setminus S} \delta(s, u)$;
  $d(s, u) = \delta(s, u)$;
  add u to S;
  for $v \in V \setminus S$ *with* $(u, v) \in E$ **do**
    if $d(s, u) + w(u, v) < \delta(s, v)$ **then**
      $\delta(s, v) = d(s, u) + w(u, v)$;
    end
  end
end

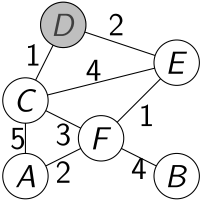


$\delta(D, D) = 0$

Dijkstra's Algorithm

Algorithm: Dijkstra(G, s)

$d(s, v) = \infty, \forall v \in V$;
$S = \emptyset$;
$\delta(s, v) = \infty, \forall v \neq s$, $\delta(s, s) = 0$;
while $S \neq V$ **do**
    $u = \arg\min_{u \in V \setminus S} \delta(s, u)$;
    $d(s, u) = \delta(s, u)$;
    add u to S;
    for $v \in V \setminus S$ *with* $(u, v) \in E$ **do**
        if $d(s, u) + w(u, v) < \delta(s, v)$ **then**
            $\delta(s, v) = d(s, u) + w(u, v)$;
        end
    end
end

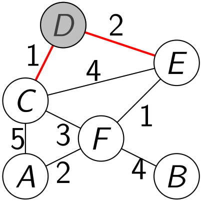


$d(D, D) = 0$

Dijkstra's Algorithm

Algorithm: Dijkstra(G, s)

$d(s, v) = \infty, \forall v \in V$;
$S = \emptyset$;
$\delta(s, v) = \infty, \forall v \neq s,\ \delta(s, s) = 0$;
while $S \neq V$ **do**
  $u = \arg\min_{u \in V \setminus S} \delta(s, u)$;
  $d(s, u) = \delta(s, u)$;
  add u to S;
  for $v \in V \setminus S$ *with* $(u, v) \in E$ **do**
    if $d(s, u) + w(u, v) < \delta(s, v)$ **then**
      $\delta(s, v) = d(s, u) + w(u, v)$;
    end
  end
end



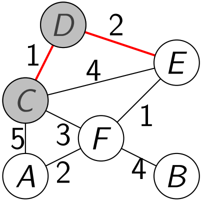
$\delta(D, C) = 1$

$\delta(D, E) = 2$

Dijkstra's Algorithm

Algorithm: Dijkstra(G, s)

$d(s, v) = \infty, \forall v \in V$;
$S = \emptyset$;
$\delta(s, v) = \infty, \forall v \neq s$, $\delta(s, s) = 0$;
while $S \neq V$ **do**
    $u = \arg\min_{u \in V \setminus S} \delta(s, u)$;
    $d(s, u) = \delta(s, u)$;
    add u to S;
    for $v \in V \setminus S$ *with* $(u, v) \in E$ **do**
        if $d(s, u) + w(u, v) < \delta(s, v)$ **then**
            $\delta(s, v) = d(s, u) + w(u, v)$;
        end
    end
end

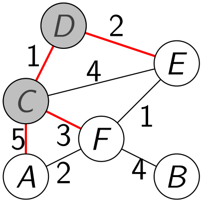


$d(D, C) = 1$

Dijkstra's Algorithm

Algorithm: Dijkstra(G, s)

$d(s,v) = \infty, \forall v \in V$;
$S = \emptyset$;
$\delta(s,v) = \infty, \forall v \neq s$, $\delta(s,s) = 0$;
while $S \neq V$ **do**
  $u = \arg\min_{u \in V \setminus S} \delta(s,u)$;
  $d(s,u) = \delta(s,u)$;
  add u to S;
  for $v \in V \setminus S$ *with* $(u,v) \in E$ **do**
    if $d(s,u) + w(u,v) < \delta(s,v)$ **then**
      $\delta(s,v) = d(s,u) + w(u,v)$;
    end
  end
end



$\delta(D,E) = 2$

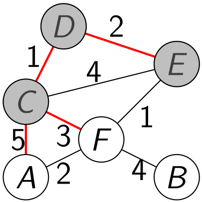
$\delta(D,A) = 6$

$\delta(D,F) = 4$

Dijkstra's Algorithm

Algorithm: Dijkstra(G, s)

$d(s, v) = \infty, \forall v \in V$;
$S = \emptyset$;
$\delta(s, v) = \infty, \forall v \neq s,\ \delta(s, s) = 0$;
while $S \neq V$ **do**
    $u = \arg\min_{u \in V \setminus S} \delta(s, u)$;
    $d(s, u) = \delta(s, u)$;
    add u to S;
    for $v \in V \setminus S$ *with* $(u, v) \in E$ **do**
        if $d(s, u) + w(u, v) < \delta(s, v)$ **then**
            $\delta(s, v) = d(s, u) + w(u, v)$;
        end
    end
end

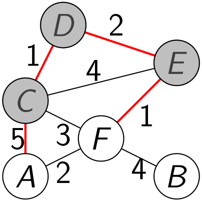


$d(D, E) = 2$

Dijkstra's Algorithm

Algorithm: Dijkstra(G, s)

$d(s, v) = \infty, \forall v \in V$;
$S = \emptyset$;
$\delta(s, v) = \infty, \forall v \neq s$, $\delta(s, s) = 0$;
while $S \neq V$ **do**
    $u = \arg\min_{u \in V \setminus S} \delta(s, u)$;
    $d(s, u) = \delta(s, u)$;
    add u to S;
    for $v \in V \setminus S$ *with* $(u, v) \in E$ **do**
        if $d(s, u) + w(u, v) < \delta(s, v)$ **then**
            $\delta(s, v) = d(s, u) + w(u, v)$;
        end
    end
end



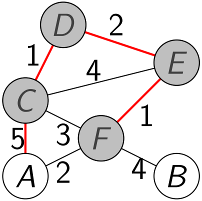
$\delta(D, A) = 6$

$\delta(D, F) = 3$

Dijkstra's Algorithm

Algorithm: Dijkstra(G, s)

$d(s, v) = \infty, \forall v \in V$;
$S = \emptyset$;
$\delta(s, v) = \infty, \forall v \neq s,\ \delta(s, s) = 0$;
while $S \neq V$ **do**
  $u = \arg\min_{u \in V \setminus S} \delta(s, u)$;
  $d(s, u) = \delta(s, u)$;
  add u to S;
  for $v \in V \setminus S$ *with* $(u, v) \in E$ **do**
    if $d(s, u) + w(u, v) < \delta(s, v)$ **then**
      $\delta(s, v) = d(s, u) + w(u, v)$;
    end
  end
end

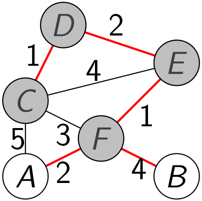


$d(D, F) = 3$

Dijkstra's Algorithm

Algorithm: Dijkstra(G, s)

$d(s, v) = \infty, \forall v \in V$;
$S = \emptyset$;
$\delta(s, v) = \infty, \forall v \neq s,\ \delta(s, s) = 0$;
while $S \neq V$ **do**
  $u = \arg\min_{u \in V \setminus S} \delta(s, u)$;
  $d(s, u) = \delta(s, u)$;
  add u to S;
  for $v \in V \setminus S$ *with* $(u, v) \in E$ **do**
    if $d(s, u) + w(u, v) < \delta(s, v)$ **then**
      $\delta(s, v) = d(s, u) + w(u, v)$;
    end
  end
end



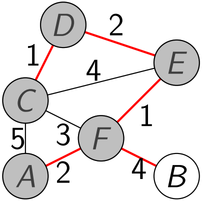
$\delta(D, A) = 5$

$\delta(D, B) = 7$

Dijkstra's Algorithm

Algorithm: Dijkstra(G, s)

$d(s, v) = \infty, \forall v \in V$;
$S = \emptyset$;
$\delta(s, v) = \infty, \forall v \neq s,\ \delta(s, s) = 0$;
while $S \neq V$ **do**
    $u = \arg\min_{u \in V \setminus S} \delta(s, u)$;
    $d(s, u) = \delta(s, u)$;
    add u to S;
    for $v \in V \setminus S$ *with* $(u, v) \in E$ **do**
        if $d(s, u) + w(u, v) < \delta(s, v)$ **then**
            $\delta(s, v) = d(s, u) + w(u, v)$;
        end
    end
end

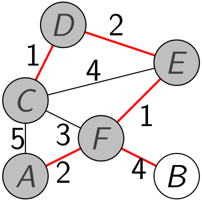


$d(D, A) = 5$

Dijkstra's Algorithm

Algorithm: Dijkstra(G, s)

$d(s, v) = \infty, \forall v \in V$;
$S = \emptyset$;
$\delta(s, v) = \infty, \forall v \neq s,\ \delta(s, s) = 0$;
while $S \neq V$ **do**
  $u = \arg\min_{u \in V \setminus S} \delta(s, u)$;
  $d(s, u) = \delta(s, u)$;
  add u to S;
  for $v \in V \setminus S$ *with* $(u, v) \in E$ **do**
    if $d(s, u) + w(u, v) < \delta(s, v)$ **then**
      $\delta(s, v) = d(s, u) + w(u, v)$;
    end
  end
end

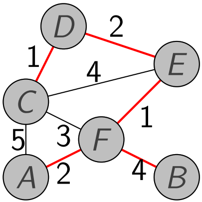


$\delta(D, B) = 7$

Dijkstra's Algorithm

Algorithm: Dijkstra(G, s)

$d(s, v) = \infty, \forall v \in V$;
$S = \emptyset$;
$\delta(s, v) = \infty, \forall v \neq s,\ \delta(s, s) = 0$;
while $S \neq V$ **do**
    $u = \arg\min_{u \in V \setminus S} \delta(s, u)$;
    $d(s, u) = \delta(s, u)$;
    add u to S;
    for $v \in V \setminus S$ *with* $(u, v) \in E$ **do**
        if $d(s, u) + w(u, v) < \delta(s, v)$ **then**
            $\delta(s, v) = d(s, u) + w(u, v)$;
        end
    end
end



$d(D, B) = 7$

Min-Priority Queue

How to find $u = \arg\min_{u \in V \setminus S} \delta(s, u)$?

Priority: $p(v) \rightarrow \mathbb{R}^+$

- is the queue empty: $O(1)$
- add/remove: $O(\log |V|)$
- get the minimum element: $O(1)$

Implementation: binary heap.

Dijkstra's Algorithm

Keep $\delta(s, v)$ in min-priority queue.

- update $\delta(s, v)$: $O(\log |V|)$
- pop the smallest: $O(\log |V|)$

Complexity: $O((|V| + |E|) \log |V|)$

- $O(|V| \log |V|)$: For each node v, $\delta(s, v)$ is popped for at most 1 time.
- $O(|E| \log |V|)$: For each edges (u, v), when u is added to S at first, $\delta(s, v)$ is updated.

All Pairs Shortest Path

Problem: Given a graph $G = (V, E)$, calculate the length of shortest path between all pairs of nodes u and v.

$T = O(|V|) \cdot O((|V| + |E|) \log |V|)$

- for each node $s \in V$,
    - find $d(s, v)$ for all $v \in V$

$T = O(|V|^3 \log |V|)$, if $|E| = \Omega(|V|^2)$

Floyd-Warshall Algorithm

Index the n nodes in V

$$v_0, v_1, v_2, \ldots, v_{n-1}$$

Dynamic programming: let $d_{i,j}^k$ be the length of the shortest path from v_i to v_j, through $\{v_0, v_1, \ldots, v_{k-1}\}$

$$d_{i,j}^k = \begin{cases} w(v_i, v_j) & if\ \ k = 0 \\ \min\{d_{i,j}^{k-1}, d_{i,k-1}^{k-1} + d_{k-1,j}^{k-1}\} & if\ \ k \geq 1 \end{cases}$$

$O(|V|^3)$

Negative Weighted Edge: Bellman-Ford Algorithm

Calculate $d(s, v)$ for all $v \in V$, with

$$w(u, v) \to \mathbb{R}$$

$d^l_{s,v}$: the length of the shortest path from s to v, through at most l nodes.

$$d^l_{s,v} = \begin{cases} w(s, v) & \textit{if } \ l = 0 \\ \min\{d^{l-1}_{s,v}, \min_{(u,v)\in E}\{d^{l-1}_{s,u} + w(u, v)\}\} & \textit{if } \ l \geq 1 \end{cases}$$

Negative weighted cycle: $d^{n-1}_{s,v} < d^{n-2}_{s,v}$ for some v.

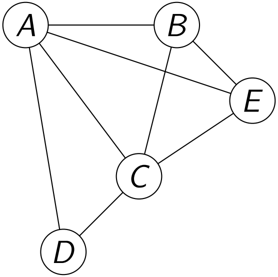
$O(|V||E|)$.

Minimum Spanning Tree

Graph and Spanning Tree

Graph. $G = (V, E)$

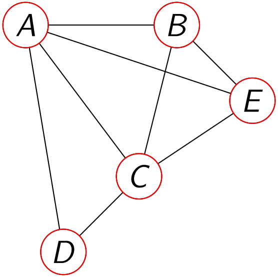
- V: vertices
- E: edges



Graph and Spanning Tree

Graph. $G = (V, E)$

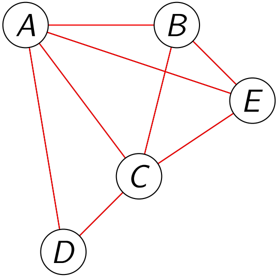
- V: vertices
- E: edges



Graph and Spanning Tree

Graph. $G = (V, E)$

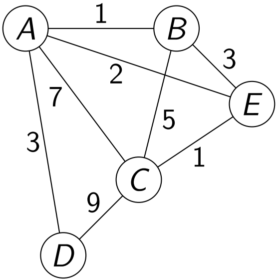
- V: vertices
- E: edges



Graph and Spanning Tree

Graph. $G = (V, E)$

- V: vertices
- E: edges
- W: weights



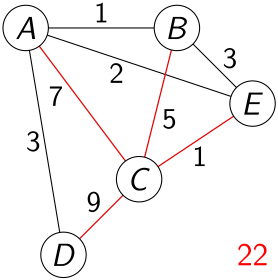
Graph and Spanning Tree

Graph. $G = (V, E)$

- V: vertices
- E: edges
- W: weights

Spanning Tree

- $|V| - 1$ edges
- connect V



Graph and Spanning Tree

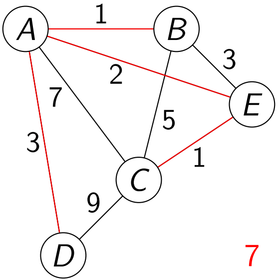
Graph. $G = (V, E)$

- V: vertices
- E: edges
- W: weights

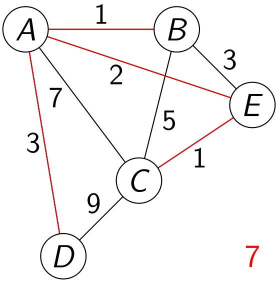
Spanning Tree

- $|V| - 1$ edges
- connect V

Problem: find the spanning tree of the minimum weight (MST).

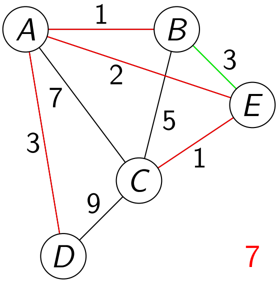


Minimum Spanning Tree: Cycle Property



Minimum Spanning Tree

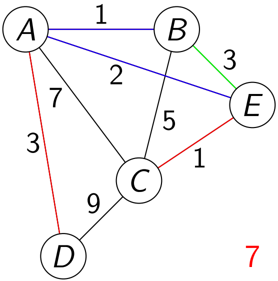
Minimum Spanning Tree: Cycle Property



Minimum Spanning Tree

add one edge to MST

Minimum Spanning Tree: Cycle Property

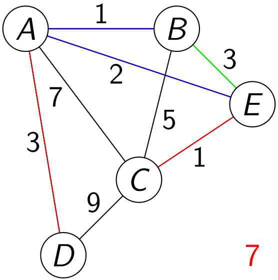


Minimum Spanning Tree

add one edge to MST

get a cycle

Minimum Spanning Tree: Cycle Property



Minimum Spanning Tree

add one edge to MST

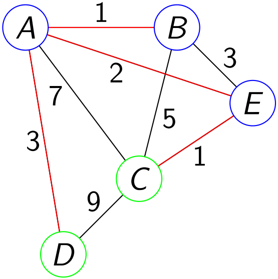
get a cycle

edge has the max weight

Minimum Spanning Tree: Smallest Link

Divide V into two parts

- V_1: A, B, E
- V_2: C, D



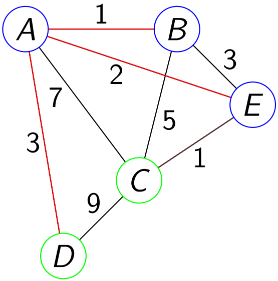
Minimum Spanning Tree: Smallest Link

Divide V into two parts

- V_1: A, B, E
- V_2: C, D

Let e be the smallest-weighted edge between V_1 and V_2

- $e = (C, E)$: weight 1
- for any MST, it must contain an edge e', suth that
  - e' links V_1 and V_2
  - $w(e') = w(e)$

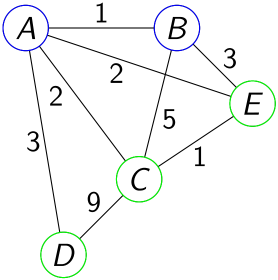


Minimum Spanning Tree: Replacement

Theorem: Given $G = (V, E)$ and w, divide V into two parts V_1 and $V_2 = V \setminus V_1$. Let $e = (u, v)$ be the smallest-weighted edge between V_1 and V_2. Then for any MST T that does not include e, we can construct MST T' that includes e.

Minimum Spanning Tree: Replacement

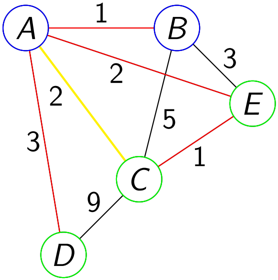
Theorem: Given $G = (V, E)$ and w, divide V into two parts V_1 and $V_2 = V \setminus V_1$. Let $e = (u, v)$ be the smallest-weighted edge between V_1 and V_2. Then for any MST T that does not include e, we can construct MST T' that includes e.



$$V_1 = [A, B]$$
$$V_2 = [C, D, E]$$

Minimum Spanning Tree: Replacement

Theorem: Given $G = (V, E)$ and w, divide V into two parts V_1 and $V_2 = V \setminus V_1$. Let $e = (u, v)$ be the smallest-weighted edge between V_1 and V_2. Then for any MST T that does not include e, we can construct MST T' that includes e.



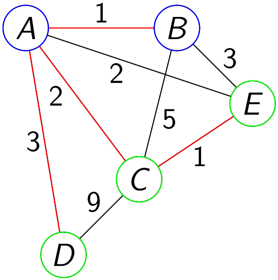
$V_1 = [A, B]$

$V_2 = [C, D, E]$

cycle: A, C, E

Minimum Spanning Tree: Replacement

Theorem: Given $G = (V, E)$ and w, divide V into two parts V_1 and $V_2 = V \setminus V_1$. Let $e = (u, v)$ be the smallest-weighted edge between V_1 and V_2. Then for any MST T that does not include e, we can construct MST T' that includes e.



$V_1 = [A, B]$

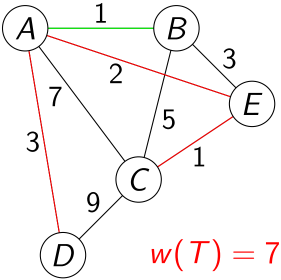
$V_2 = [C, D, E]$

cycle: A, C, E

$w(A, C) \leq w(A, E)$

Minimum Spanning Tree: Optimal Structure

Remove one edge from T

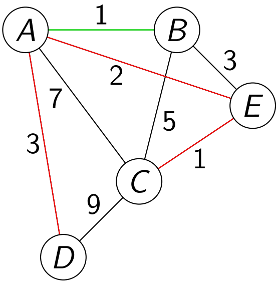


$w(T) = 7$

Minimum Spanning Tree: Optimal Structure

Remove one edge from T

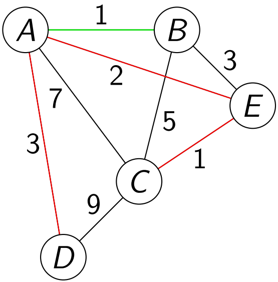
- T_1 : B
- T_2 : (A, E), (A, D) , (C, E)



Minimum Spanning Tree: Optimal Structure

Remove one edge from T

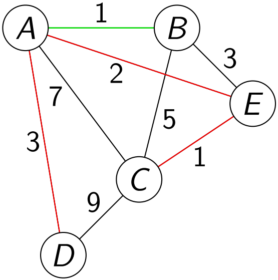
- T_1 : B
- T_2 : (A, E), (A, D) , (C, E)
- T_1 is MST of B



Minimum Spanning Tree: Optimal Structure

Remove one edge from T

- T_1 : B
- T_2 : (A, E), (A, D) , (C, E)
- T_1 is MST of B
- T_2 is MST of $[A, C, D, E]$



Minimum Spanning Tree: Optimal Structure

Theorem: Let T_1 and T_2 be the subtrees derived by removing one edge from the MST T.

- $G_1 = (V_1, E_1)$
  - V_1: the vertices in T_1
  - E_1: edges between V_1
- $G_2 = (V_2, E_2)$
  - V_2: the vertices in T_2
  - E_2: edges between V_2

Then, T_1 is MST of G_1, and T_2 is MST of G_2.

Minimum Spanning Tree: Optimal Structure

T_1 is MST of G_1, and T_2 is MST of G_2.

Proof: Let $e = (u, v)$ be the removed edge.

$$w(T) = w(e) + w(T_1) + w(T_2)$$

Minimum Spanning Tree: Optimal Structure

T_1 is MST of G_1, and T_2 is MST of G_2.

Proof: Let $e = (u, v)$ be the removed edge.

$$w(T) = w(e) + w(T_1) + w(T_2)$$

- Assume that there is MST T_1' for G_1, with

$$w(T_1') < w(T_1)$$

Minimum Spanning Tree: Optimal Structure

T_1 is MST of G_1, and T_2 is MST of G_2.

Proof: Let $e = (u, v)$ be the removed edge.

$$w(T) = w(e) + w(T_1) + w(T_2)$$

- Assume that there is MST T_1' for G_1, with

$$w(T_1') < w(T_1)$$

- Then $T' = \{e\} \cup T_1' \cup T_2$ is a spanning tree, and

$$w(T') = w(e) + w(T_1') + w(T_2) < w(T)$$

Minimum Spanning Tree: Optimal Structure

T_1 is MST of G_1, and T_2 is MST of G_2.

Proof: Let $e = (u, v)$ be the removed edge.

$$w(T) = w(e) + w(T_1) + w(T_2)$$

- Assume that there is MST T_1' for G_1, with

$$w(T_1') < w(T_1)$$

- Then $T' = \{e\} \cup T_1' \cup T_2$ is a spanning tree, and

$$w(T') = w(e) + w(T_1') + w(T_2) < w(T)$$

- **Contradiction** to the fact that T is MST.

Finding MST: Greedy

Matroid $M(S, \mathcal{I})$ for finding MST of $G = (V, E)$: $S = E$.

Finding MST: Greedy

Matroid $M(S, \mathcal{I})$ for finding MST of $G = (V, E)$: $S = E$.

Kruskal's algorithm

- $\mathcal{I}$: the edge subsets that forms no cycle

Finding MST: Greedy

Matroid $M(S, \mathcal{I})$ for finding MST of $G = (V, E)$: $S = E$.

Kruskal's algorithm

- $\mathcal{I}$: the edge subsets that forms no cycle
- **hereditary**: A has no cycle, then $A' \subseteq A$ has no cycle.

Finding MST: Greedy

Matroid $M(S, \mathcal{I})$ for finding MST of $G = (V, E)$: $S = E$.

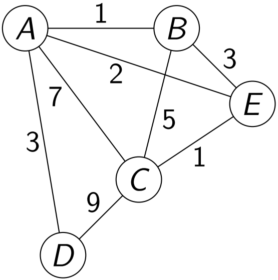
Kruskal's algorithm

- $\mathcal{I}$: the edge subsets that forms no cycle
- **hereditary**: A has no cycle, then $A' \subseteq A$ has no cycle.
- **exchange**: $|A| < |B| \in \mathcal{I} \Rightarrow$
  - find $(u, v) \in B$, such that $A \cup \{(u, v)\}$ has no cycle
    - if for all $(u, v) \in B$, $A \cup \{(u, v)\}$ has a cycle, then B has a cycle.
    - repeat: add an edge of B to A, and remove an edge (of A) in the cycle.

Finding MST: Greedy

Algorithm: Kruskal(G, w)

$M = \emptyset$;
Sort E in the **increasing** order of $w[e]$;
for $i = 0$ *to* $|E| - 1$ **do**
  if $A \cup \{E[i]\}$ *has no cycle* **then** add $E[i]$ to M;
end
Return M;

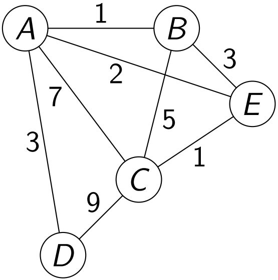


$w(A, B) = 1, w(C, E) = 1, w(A, E) = 2$
$w(B, E) = 3, w(A, D) = 3, w(B, C) = 5$
$w(A, C) = 7, w(C, D) = 9$

Finding MST: Greedy

Algorithm: Kruskal(G, w)

$M = \emptyset$;
Sort E in the **increasing** order of $w[e]$;
for $i = 0$ *to* $|E| - 1$ **do**
    if $A \cup \{E[i]\}$ *has no cycle* **then** add $E[i]$ to M;
end
Return M;



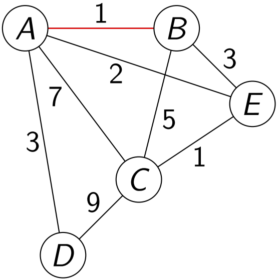
$w(A, B) = 1, w(C, E) = 1, w(A, E) = 2$
$w(B, E) = 3, w(A, D) = 3, w(B, C) = 5$
$w(A, C) = 7, w(C, D) = 9$

(A, B): create no cycle

Finding MST: Greedy

Algorithm: Kruskal(G, w)

$M = \emptyset$;
Sort E in the **increasing** order of $w[e]$;
for $i = 0$ *to* $|E| - 1$ **do**
    if $A \cup \{E[i]\}$ *has no cycle* **then** add $E[i]$ to M;
end
Return M;



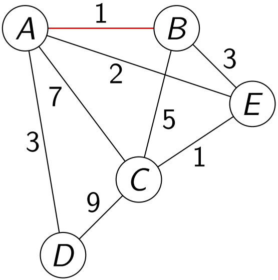
$w(A,B) = 1, w(C,E) = 1, w(A,E) = 2$
$w(B,E) = 3, w(A,D) = 3, w(B,C) = 5$
$w(A,C) = 7, w(C,D) = 9$

(A, B): create no cycle

Finding MST: Greedy

Algorithm: Kruskal(G, w)

$M = \emptyset$;
Sort E in the **increasing** order of $w[e]$;
for $i = 0$ *to* $|E| - 1$ **do**
  if $A \cup \{E[i]\}$ *has no cycle* **then** add $E[i]$ to M;
end
Return M;



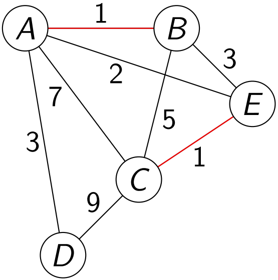
$w(A,B) = 1, w(C,E) = 1, w(A,E) = 2$
$w(B,E) = 3, w(A,D) = 3, w(B,C) = 5$
$w(A,C) = 7, w(C,D) = 9$

(C,E): create no cycle

Finding MST: Greedy

Algorithm: Kruskal(G, w)

$M = \emptyset$;
Sort E in the **increasing** order of $w[e]$;
for $i = 0$ *to* $|E| - 1$ **do**
  if $A \cup \{E[i]\}$ *has no cycle* **then** add $E[i]$ to M;
end
Return M;



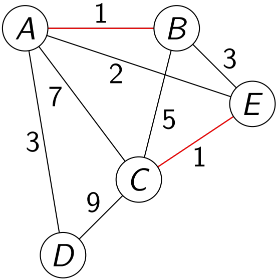
$w(A, B) = 1, w(C, E) = 1, w(A, E) = 2$
$w(B, E) = 3, w(A, D) = 3, w(B, C) = 5$
$w(A, C) = 7, w(C, D) = 9$

(C, E): create no cycle

Finding MST: Greedy

Algorithm: Kruskal(G, w)

$M = \emptyset$;
Sort E in the **increasing** order of $w[e]$;
for $i = 0$ *to* $|E| - 1$ **do**
  if $A \cup \{E[i]\}$ *has no cycle* **then** add $E[i]$ to M;
end
Return M;



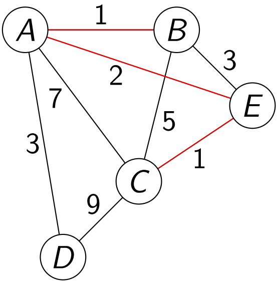
$w(A, B) = 1, w(C, E) = 1, w(A, E) = 2$
$w(B, E) = 3, w(A, D) = 3, w(B, C) = 5$
$w(A, C) = 7, w(C, D) = 9$

(A, E): create no cycle

Finding MST: Greedy

Algorithm: Kruskal(G, w)

$M = \emptyset$;
Sort E in the **increasing** order of $w[e]$;
for $i = 0$ *to* $|E| - 1$ **do**
    if $A \cup \{E[i]\}$ *has no cycle* **then** add $E[i]$ to M;
end
Return M;



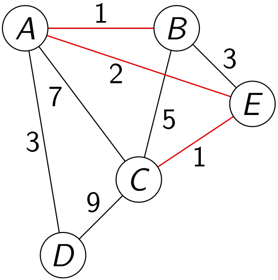
$w(A,B) = 1, w(C,E) = 1, w(A,E) = 2$
$w(B,E) = 3, w(A,D) = 3, w(B,C) = 5$
$w(A,C) = 7, w(C,D) = 9$

(A, E): create no cycle

Finding MST: Greedy

Algorithm: Kruskal(G, w)

$M = \emptyset$;
Sort E in the **increasing** order of $w[e]$;
for $i = 0$ *to* $|E| - 1$ **do**
    if $A \cup \{E[i]\}$ *has no cycle* **then** add $E[i]$ to M;
end
Return M;



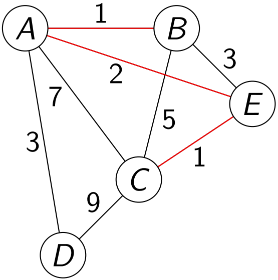
$w(A, B) = 1, w(C, E) = 1, w(A, E) = 2$
$w(B, E) = 3, w(A, D) = 3, w(B, C) = 5$
$w(A, C) = 7, w(C, D) = 9$

(B, E): create a cycle

Finding MST: Greedy

Algorithm: Kruskal(G, w)

$M = \emptyset$;
Sort E in the **increasing** order of $w[e]$;
for $i = 0$ *to* $|E| - 1$ **do**
  if $A \cup \{E[i]\}$ *has no cycle* **then** add $E[i]$ to M;
end
Return M;



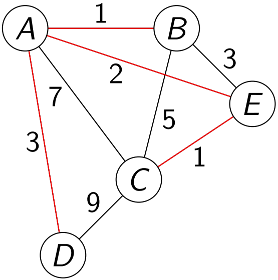
$w(A, B) = 1, w(C, E) = 1, w(A, E) = 2$
$w(B, E) = 3, w(A, D) = 3, w(B, C) = 5$
$w(A, C) = 7, w(C, D) = 9$

(A, D): create no cycle

Finding MST: Greedy

Algorithm: Kruskal(G, w)

$M = \emptyset$;
Sort E in the **increasing** order of $w[e]$;
for $i = 0$ *to* $|E| - 1$ **do**
  if $A \cup \{E[i]\}$ *has no cycle* **then** add $E[i]$ to M;
end
Return M;



$w(A, B) = 1, w(C, E) = 1, w(A, E) = 2$
$w(B, E) = 3, w(A, D) = 3, w(B, C) = 5$
$w(A, C) = 7, w(C, D) = 9$

(A, D): create no cycle

Finding MST: Greedy

Algorithm: Kruskal(G, w)

$M = \emptyset$;

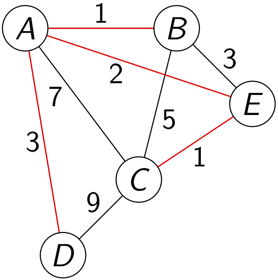
Sort E in the **increasing** order of $w[e]$;

for $i = 0$ *to* $|E| - 1$ **do**

  if $A \cup \{E[i]\}$ *has no cycle* **then** add $E[i]$ to M;

end

Return M;



$w(A, B) = 1, w(C, E) = 1, w(A, E) = 2$
$w(B, E) = 3, w(A, D) = 3, w(B, C) = 5$
$w(A, C) = 7, w(C, D) = 9$

$w(M) = 7$

Finding MST: Greedy

Theorem: Kruskal's algorithm returns a MST.

For each edge $e = (u, v)$ added into M.

- let T be the tree connected to v according to underlying M.
- e is the smallest-weighted edge between $V(T)$ and $V \setminus V(T)$

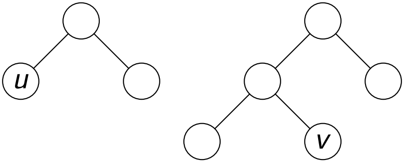
For any MST M' other than M, we can: replace the edges in M' by edges in M without losing any weight.

Finding MST: Greedy

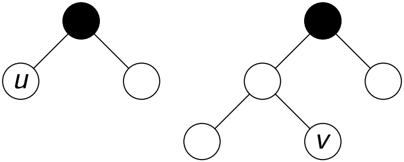
Time Complexity: $O(|E| \log |V|)$

- Sort the edges: $O(|E| \log |E|) = O(|E| \log |V|)$
- Consider the edges one by one : $O(E)$
  - Cycle check: $O(\log |V|)$
  - How?

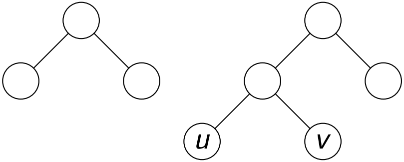
Finding MST: Union of Disjoint Sets



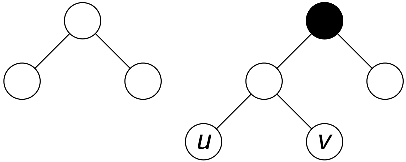
Finding MST: Union of Disjoint Sets



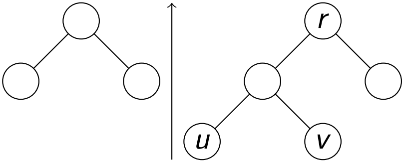
Finding MST: Union of Disjoint Sets



Finding MST: Union of Disjoint Sets

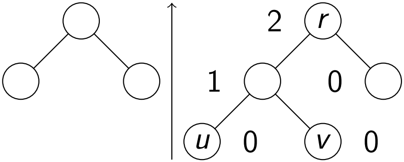


Finding MST: Union of Disjoint Sets



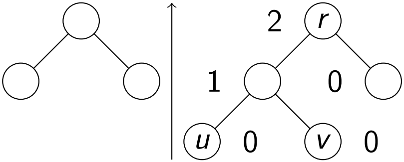
- Rank(r): the height of the subtree rooted at r.

Finding MST: Union of Disjoint Sets



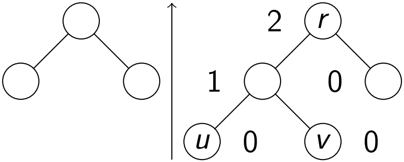
- Rank(r): the height of the subtree rooted at r.

Finding MST: Union of Disjoint Sets



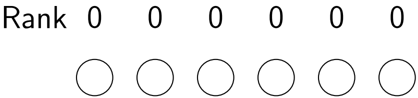
- Rank(r): the height of the subtree rooted at r.
- **Claim**: $\mathrm{R}(r) = O(\log n)$.

Finding MST: Union of Disjoint Sets



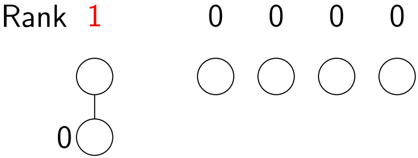
- Rank(r): the height of the subtree rooted at r.
- **Claim**: $\mathrm{R}(r) = O(\log n)$.
- It is sufficient to prove the subtree rooted at r has at least $2^{Rank(r)}$ nodes.

Finding MST: Union of Disjoint Sets



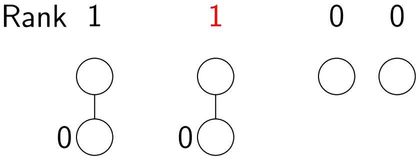
- Initiallly, every node has rank 0.

Finding MST: Union of Disjoint Sets



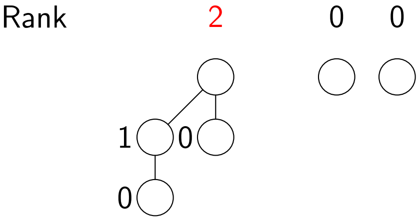
- Initiallly, every node has rank 0.
- **Union**: point the root of the lower rank to the root of the higher rank.

Finding MST: Union of Disjoint Sets



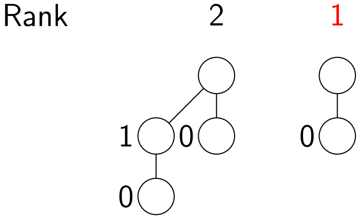
- Initiallly, every node has rank 0.
- **Union**: point the root of the lower rank to the root of the higher rank.

Finding MST: Union of Disjoint Sets



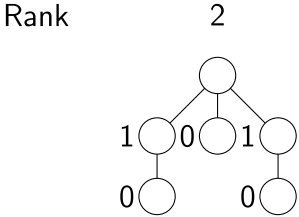
- Initiallly, every node has rank 0.
- **Union**: point the root of the lower rank to the root of the higher rank.

Finding MST: Union of Disjoint Sets



- Initiallly, every node has rank 0.
- **Union**: point the root of the lower rank to the root of the higher rank.

Finding MST: Union of Disjoint Sets



- Initiallly, every node has rank 0.
- **Union**: point the root of the lower rank to the root of the higher rank.

Finding MST: Union of Disjoint Sets

Subtree at node v has at least $2^{Rank(v)}$ nodes.

Proof: Induction on rank k.

Finding MST: Union of Disjoint Sets

Subtree at node v has at least $2^{Rank(v)}$ nodes.

Proof: Induction on rank k.

- **Base case**: for root of rank 0, the tree has $1 \geq 2^0$ nodes.

Finding MST: Union of Disjoint Sets

Subtree at node v has at least $2^{Rank(v)}$ nodes.

Proof: Induction on rank k.

- **Base case**: for root of rank 0, the tree has $1 \geq 2^0$ nodes.
- **Assume**: for root of rank $k-1$, the tree has $\geq 2^{k-1}$ nodes.

Finding MST: Union of Disjoint Sets

Subtree at node v has at least $2^{Rank(v)}$ nodes.

Proof: Induction on rank k.

- **Base case**: for root of rank 0, the tree has $1 \geq 2^0$ nodes.
- **Assume**: for root of rank $k-1$, the tree has $\geq 2^{k-1}$ nodes.
- **Consider** a root of rank k.

Finding MST: Union of Disjoint Sets

Subtree at node v has at least $2^{Rank(v)}$ nodes.

Proof: Induction on rank k.

- **Base case**: for root of rank 0, the tree has $1 \geq 2^0$ nodes.
- **Assume**: for root of rank $k-1$, the tree has $\geq 2^{k-1}$ nodes.
- **Consider** a root of rank k.
  - rank $k-1$ to rank k: point a root of rank $k-1$ to another root of rank $k-1$.

Finding MST: Union of Disjoint Sets

Subtree at node v has at least $2^{Rank(v)}$ nodes.

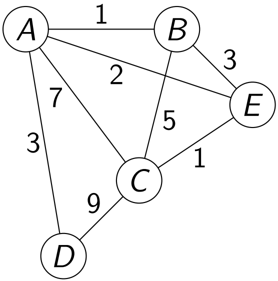
Proof: Induction on rank k.

- **Base case**: for root of rank 0, the tree has $1 \geq 2^0$ nodes.
- **Assume**: for root of rank $k-1$, the tree has $\geq 2^{k-1}$ nodes.
- **Consider** a root of rank k.
  - rank $k-1$ to rank k: point a root of rank $k-1$ to another root of rank $k-1$.
  - # nodes: $\geq 2 \cdot 2^{k-1} = 2^k$.

Finding MST: Another Greedy Idea

Algorithm: Prim(G, w)

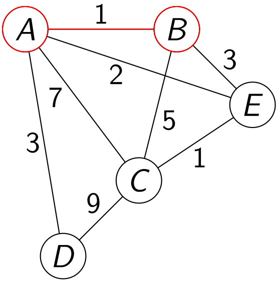
$M = \emptyset$;
while $|M| < |V| - 1$ **do**
  let e be the smallest-weighted edge between $V(E)$ and $V \setminus V(E)$;
  add e to M;
end
Return M;



Finding MST: Another Greedy Idea

Algorithm: Prim(G, w)

$M = \emptyset$;
while $|M| < |V| - 1$ **do**
  let e be the smallest-weighted edge between $V(E)$ and $V \setminus V(E)$;
  add e to M;
end
Return M;

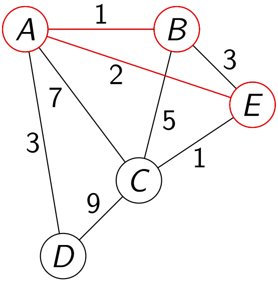


$w(A, B) = 1$

Finding MST: Another Greedy Idea

Algorithm: Prim(G, w)

$M = \emptyset$;
while $|M| < |V| - 1$ **do**
  let e be the smallest-weighted edge between $V(E)$ and $V \setminus V(E)$;
  add e to M;
end
Return M;

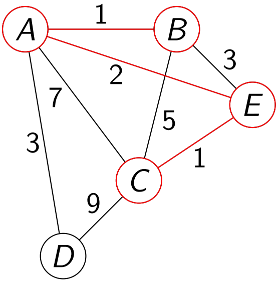


$w(A, E) = 2$

Finding MST: Another Greedy Idea

Algorithm: Prim(G, w)

$M = \emptyset$;
while $|M| < |V| - 1$ **do**
  let e be the smallest-weighted edge between $V(E)$ and $V \setminus V(E)$;
  add e to M;
end
Return M;

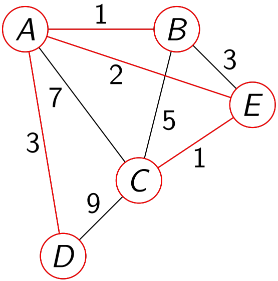


$w(C, E) = 1$

Finding MST: Another Greedy Idea

Algorithm: Prim(G, w)

$M = \emptyset$;
while $|M| < |V| - 1$ **do**
  let e be the smallest-weighted edge between $V(E)$ and $V \setminus V(E)$;
  add e to M;
end
Return M;

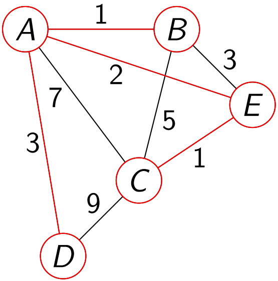


$w(A, D) = 3$

Finding MST: Another Greedy Idea

Algorithm: Prim(G, w)

$M = \emptyset$;
while $|M| < |V| - 1$ **do**
  let e be the smallest-weighted edge between $V(E)$ and $V \setminus V(E)$;
  add e to M;
end
Return M;



$w(M) = 7$

Finding MST: Prim

Theorem: Prim's algorithm returns a MST.

For each edge $e = (u, v)$ added into M.

- M is a tree.
- e is the smallest-weighted edge between $V(M)$ and $V \setminus V(M)$

For any MST M' other than M, we can: replace the edges in M' by edges in M without losing any weight.

Finding MST: Prim

Time Complexity: $O(|E| \log |V|)$

- How to find the smallest edge which extends M?
- **min-priority queue**
- Thus
  - Maintain the priorities of all edges between $V(M)$ and $V \setminus V(M)$
  - Take the one of the minimum priority, and add it to M.
  - Update the edges between $V(M)$ and $V \setminus V(M)$.

Finding MST: Prim

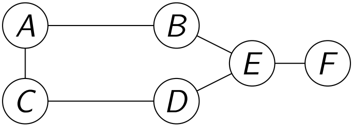
Algorithm: Prim(G, w)

$M = \emptyset$;
$B = [\,]$; # min-priority queue of nodes between $V(M)$ and $V \setminus V(M)$
while $|M| < |V| - 1$ **do**
    let e be the min edge in B;
    let v be e's endpoint outside M;
    add e to M;
    for *each* $u \in V \setminus V(M)$ *and* $(v, u) \in E$ **do**
        add **or** update the priority of u;
    end
    while *endpoins of the min edge are all in* M **do**
        remove the min edge from B;
    end
end
Return M;

Independent Set

Independent Set

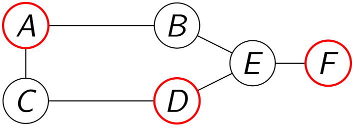
Given a graph $G = (V, E)$, an independent set is a subset of V, such that there are no edges between them.



Independent Set

Given a graph $G = (V, E)$, an independent set is a subset of V, such that there are no edges between them.

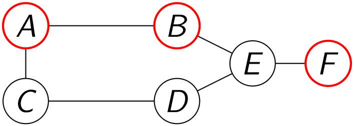
- $\{A, D, F\}$ is an independent set



Independent Set

Given a graph $G = (V, E)$, an independent set is a subset of V, such that there are no edges between them.

- $\{A, D, F\}$ is an independent set
- $\{A, B, F\}$ is not an independent set



Largest Independent Set

Problem: Given a graph $G = (V, E)$, find the largest independent set.

It is believed to be **intractable**, for the general cases.

What about the trees?

Largest Independent Set

Problem: Given a tree rooted at node r, find the largest independent set.

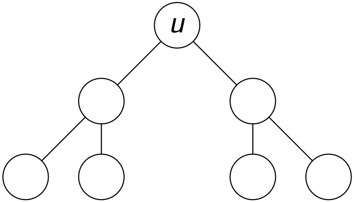
$I(u) =$ size of the largest independent set of subtree rooted at node u.

Solution: $I(r)$.

Calculate $I(u)$

Assume that $I(v)$ is known for all the children v (of node u).

$I(u)$



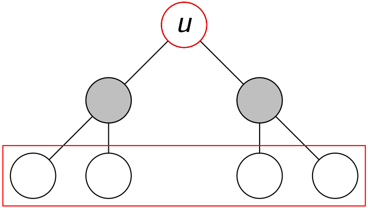
Calculate $I(u)$

Assume that $I(v)$ is known for all the children v (of node u).

$I(u)$

- if u is included

$$I(u) = 1 + \sum_{grandchild\ v} I(v)$$



Calculate $I(u)$

Assume that $I(v)$ is known for all the children v (of node u).

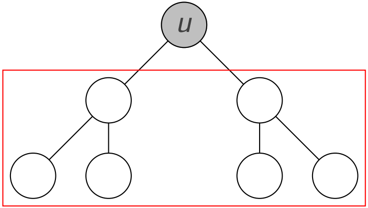
$I(u)$

- if u is included

$$I(u) = 1 + \sum_{grandchild\ \ v} I(v)$$

- if u is not included

$$I(u) = \sum_{child\ \ v} I(v)$$



THANK YOU

中国科学院深圳先进技术研究院
SHENZHEN INSTITUTES OF ADVANCED TECHNOLOGY
CHINESE ACADEMY OF SCIENCES