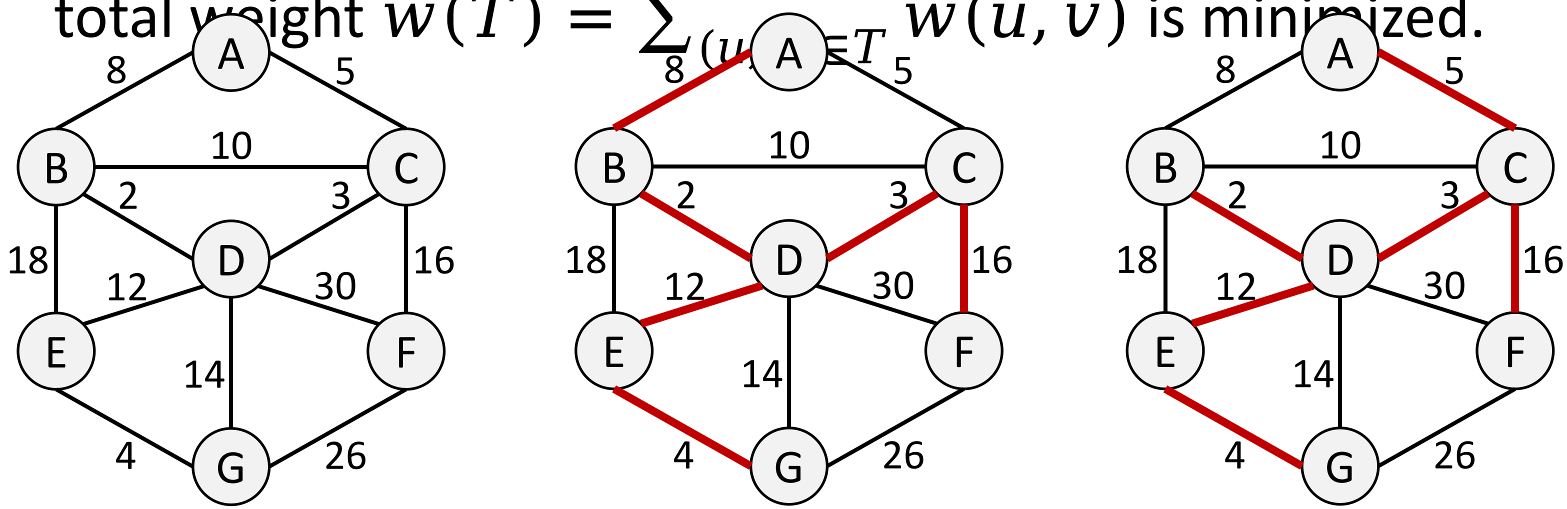
Minimum Spanning Trees

Data Structures and Algorithms

Nanjing University, Fall 2021
郑朝栋

Minimum Spanning Trees (MST)

- Consider a connected, undirected, *weighted* graph G.
- That is, we have a graph $G = (V, E)$ together with a weight function $w: E \to \mathbb{R}$ that assigns a real weight $w(u, v)$ to each edge $(u, v) \in E$.
- A **spanning tree** is a tree containing all nodes in V and a subset T of all the edges E.
- A **minimum spanning tree (MST)** is a spanning tree whose total weight $w(T) = \sum_{(u,v)\in T} w(u, v)$ is minimized.



Application of MST

- **Network Design:**
  (E.g., build a minimum cost network connecting all nodes.)
  - Transportation networks.
  - Water supply networks.
  - Telecommunication networks.
  - Computer networks.
- **Many other applications...**
  - E.g., important subroutine in more advanced algorithms.
    (E.g., used in a classical approximation algorithm for solving TSP.)

Computing MST

- **Consider the following generic method:**
- Starting with all nodes and an empty set of edges A.
- Find some edge to add to A, maintaining the invariant that "A is a subset of some MST".
  (At anytime, A is the edge set of a spanning forest.)
- Repeat above step until we have a spanning tree.
  (The resulting spanning tree must be a MST.)

These edges also called "**safe edges**".

Easy to determine.
(E.g., when $|A| = n - 1$.)

How to identify "safe edges"?

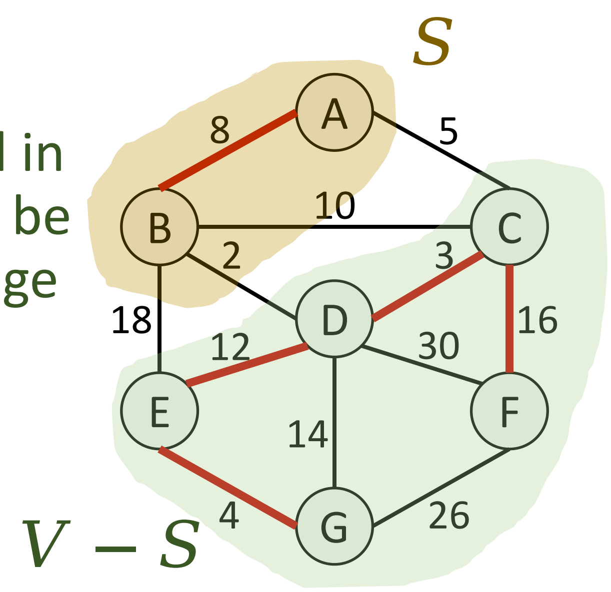
GenericMST(G,w):

```
A = ∅
while (A is not a spanning tree)
  Find edge (u,v) maintaining the invariant
  A = A ∪ {(u,v)}
return A
```

Identifying Safe Edges

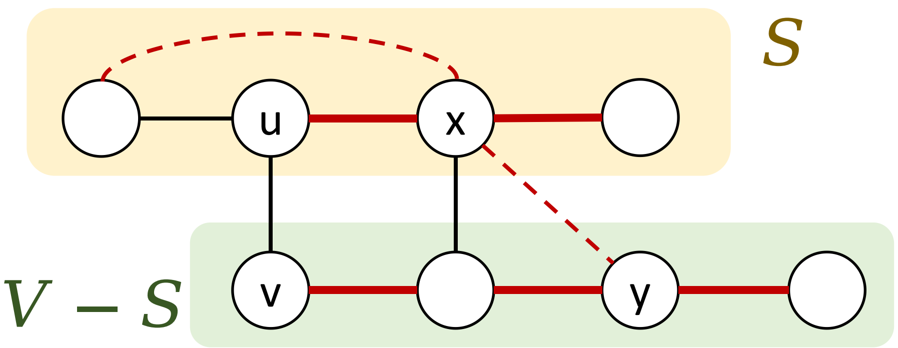
- A **cut** $(S, V - S)$ of $G = (V, E)$ is a partition of V into two parts.
- An edge **crosses** the cut $(S, V - S)$ if one of its endpoint is in S and the other endpoint is in $V - S$.
- A cut **respects** an edge set A if no edge in A crosses the cut.
- An edge is a **light edge** crossing a cut if the edge has minimum weight among all edges crossing the cut.

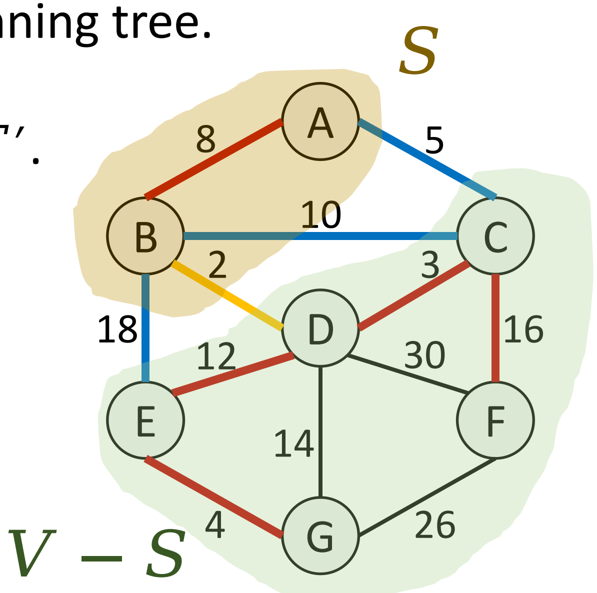
- **Thm [Cut Property]:** Assume A is included in the edge set of some MST, let $(S, V - S)$ be any cut respecting A. If (u, v) is a light edge crossing the cut, then (u, v) is safe for A.



Identifying Safe Edges

- **Thm [Cut Property]:** Assume A is included in the edge set of some MST, let $(S, V - S)$ be any cut respecting A. If (u, v) is a light edge crossing the cut, then (u, v) is safe for A.
- **Proof:**
- Let T be a MST containing A, assume it does not include (u, v).
- Then some edge in T must cross the cut, let (x, y) be one such edge.
- $T' = T - \{(x, y)\} + \{(u, v)\}$ must be a spanning tree.
- Since (u, v) is light edge crossing the cut, T' must be a MST, and (u, v) is safe for A in T'.





Computing MST

GenericMST(G,w):

```
A = ∅
while (A is not a spanning tree)
  Find a safe edge (u,v)
  A = A ∪ {(u,v)}
return A
```

- **Generic method for computing MST:**
- Starting with all nodes and an empty set of edges A.
- Find a *safe edge* to add to A,
  maintaining the invariant that "A is a subset of some MST".
  (At anytime, A is the edge set of a spanning forest.)
- Repeat above step until we have a spanning tree.
  (The resulting spanning tree must be a MST.)
- **Thm [Cut Property]:** Assume A is included in some MST, let $(S, V - S)$ be any cut respecting A. If (u, v) is a light edge crossing the cut, then (u, v) is safe for A.
- **Corollary:** Assume A is included in some MST, let $G_A = (V, A)$. Then for any connected component in G_A, its minimum-weight-outgoing-edge (MWOE) in G is safe for A.
  (In G_A, an edge in a CC is "outgoing" if it connects to another CC.)

Computing MST

Kruskal's Algorithm

- **Generic method for computing MST:**
  - Starting with all nodes and an empty set of edges A.
  - Find a *safe edge* to add to A.
  - Repeat above step until we have a spanning tree.
- **Cut Property**: Assume A is included in some MST, let $G_A = (V, A)$. For any CC in G_A, its minimum-weight-outgoing-edge in G is safe for A.
- **Strategy for finding safe edge in Kruskal's algorithm:** Find minimum weight edge connecting two CC in G_A.

KruskalMST(G,w):

```
A = ∅
Sort edges into weight increasing order
for (each edge (u,v) taken in weight increasing order)
  if (adding edge (u,v) does not form cycle in A)
    A = A ∪ {(u,v)}
return A
```

Computing MST

Kruskal's Algorithm

- **Kruskal's algorithm for computing MST:**
  - Starting with all nodes and an empty set of edges A.
  - Find minimum weight edge connecting two CC in $G_A = (V, A)$.
  - Repeat above step until we have a spanning tree.
- **Put another way**:
  - Start with n CC (each node itself is a CC) and $A = \emptyset$.
  - Find minimum weight edge connecting two CC. (# of CC reduce by 1.)
  - Repeat until one CC remains.

Computing MST

Kruskal's Algorithm

- **Kruskal's algorithm for computing MST:**
  - Starting with all nodes and an empty set of edges A.
  - Find minimum weight edge connecting two CC in $G_A = (V, A)$.
  - Repeat above step until we have a spanning tree.

KruskalMST(G,w):

```
A = ∅
Sort edges into weight increasing order
for (each edge (u,v) taken in weight increasing order)
  if (adding edge (u,v) does not form cycle in A)
    A = A ∪ {(u,v)}
return A
```

- How to determine an edge forms a cycle?
  (Put another way, how to determine if the edge is connecting two CC?)
- Use disjoint-set data structure!
  (Each set is a CC, u and v in same CC if `Find(u)==Find(v)`.)

Computing MST

Kruskal's Algorithm

- **Kruskal's algorithm for computing MST:**
  - Starting with all nodes and an empty set of edges A.
  - Find minimum weight edge connecting two CC in $G_A = (V, A)$.
  - Repeat above step until we have a spanning tree.
- **Runtime of Kruskal's algorithm?**
- $O(m\log n)$ when using disjoint-set data structure.

KruskalMST(G,w):

```
A = ∅
Sort edges into weight increasing order
for (each node u in V(G))
  MakeSet(u)
for (each edge (u,v) taken in weight increasing order)
  if (Find(u) != Find(v))
    A = A ∪ {(u,v)}
    Union(u,v)
return A
```

$O(m\log m) = O(m\log n)$

$O(n)$

$O(m\log^* n)$

Computing MST

Prim's Algorithm

- **Generic method for computing MST:**
  - Starting with all nodes and an empty set of edges A.
  - Find a *safe edge* to add to A.
  - Repeat above step until we have a spanning tree.
- **Cut Property**: Assume A is included in some MST, let $G_A = (V, A)$. For any CC in G_A, its minimum-weight-outgoing-edge in G is safe for A.
- **Strategy for finding safe edge in Prim's algorithm:** Keep finding MWOE in one *fixed* CC in G_A.

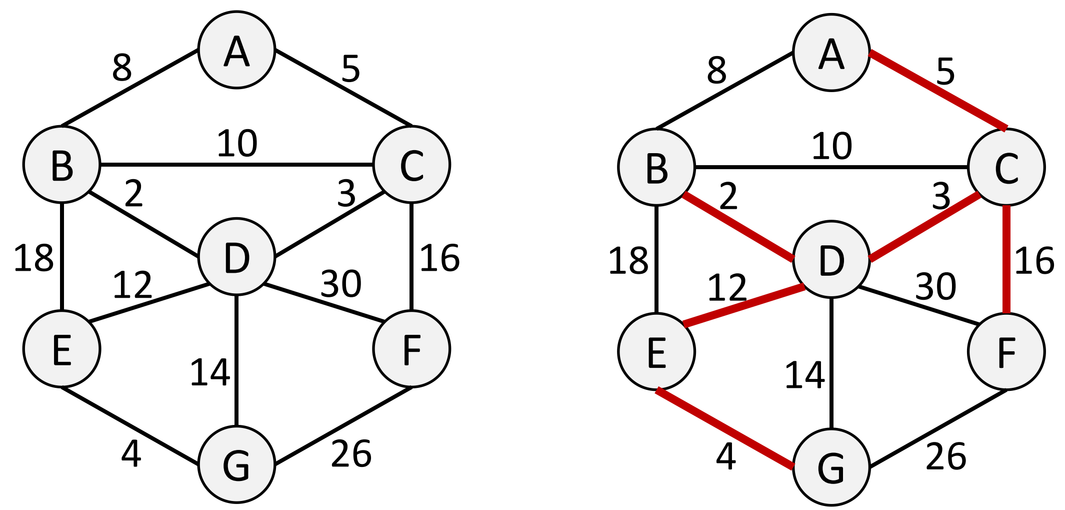
PrimMST(G,w):

```
A = ∅
Cx = {x}
while (Cx is not a spanning tree)
  Find MWOE (u,v) of Cx
  A = A ∪ {(u,v)}
  Cx = Cx ∪ {v}
return A
```

Computing MST

Prim's Algorithm

- **Prim's algorithm for computing MST:**
  - Starting with all nodes and an empty set of edges A.
  - Find MWOE of one fixed CC in $G_A = (V, A)$.
  - Repeat above step until we have a spanning tree.
- **Put another way**:
  - Start with n CC (each node itself is a CC) and $A = \emptyset$. Pick a node x.
  - Find MWOE of the component containing x. (# of CC reduce by 1.)
  - Repeat until one CC remains.



Computing MST

Prim's Algorithm

- **Prim's algorithm for computing MST:**
  - Starting with all nodes and an empty set of edges A.
  - Find MWOE in one *fixed* CC in G_A.
  - Repeat above step until we have a spanning tree.

PrimMST(G,w):

```
A = ∅
C_x = {x}
while (C_x is not a spanning tree)
  Find MWOE (u,v) of C_x
  A = A ∪ {(u,v)}
  C_x = C_x ∪ {v}
return A
```

- How to find MWOE efficiently?
- Put another way: how to find the next node that is closest to C_x?
- Use a priority queue to maintain each remaining node's distance to C_x.

Computing MST

Prim’s Algorithm

- **Prim’s algorithm for computing MST:**
  - Starting with all nodes and an empty set of edges A.
  - Find MWOE in one *fixed* CC in G_A. (Find next node closest to the fixed CC.)
  - Repeat above step until we have a spanning tree.

PrimMST(G,w):

```
Pick an arbitrary node x
for (each node u)
  u.dist = INF, u.parent = NIL, u.in = false
x.dist = 0
Build a priority queue Q based on “dist” values
while (Q is not empty)
  u = Q.ExtractMin()
  u.in = true
  for (each edge (u,v))
    if (v.in==false and w(u,v)<v.dist)
      v.parent = u, v.dist = w(u,v)
      Q.Update(v,w(u,v))
```

Computing MST

Prim's Algorithm

- **Runtime of the Prim's algorithm?**
- $O(m\log n)$ using binary heap to implement priority queue.
- Could be faster using better priority queue implementation.

PrimMST(G,w):

```
Pick an arbitrary node x
for (each node u)
  u.dist = INF, u.parent = NIL, u.in = false
x.dist = 0
Build a priority queue Q based on "dist" values
while (Q is not empty)
  u = Q.ExtractMin()
  u.in = true
  for (each edge (u,v))
    if (v.in==false and w(u,v)<v.dist)
      v.parent = u, v.dist = w(u,v)
      Q.Update(v,w(u,v))
```

$O(n)$

$O(n)$

$O(n\log n)$

$O(m\log n)$

DFS, BFS, Prim, and others…

DFSIterSkeleton(G,s):

```
Stack Q
Q.push(s)
while (!Q.empty())
  u = Q.pop()
  if (!u.visited)
    u.visited = true
    for (each edge (u,v) in E)
      Q.push(v)
```

BFSSkeletonAlt(G,s):

```
FIFOQueue Q
Q.enque(s)
while (!Q.empty())
  u = Q.dequeue()
  if (!u.visited)
    u.visited = true
    for (each edge (u,v) in E)
      Q.enque(v)
```

PrimMSTSkeleton(G,x):

```
PriorityQueue Q
Q.add(x)
while (!Q.empty())
  u = Q.remove()
  if (!u.visited)
    u.visited = true
    for (each edge (u,v) in E)
      if (!v.visited and …)
        Q.update(v,…)
```

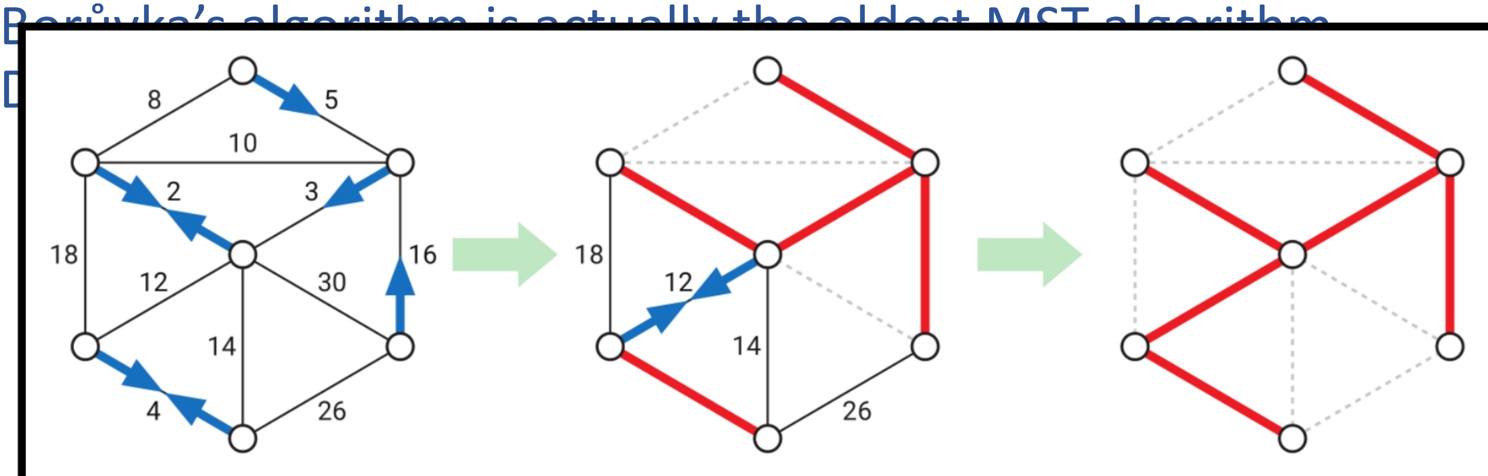
GraphExploreSkeleton(G,s):

```
GenericQueue Q
Q.add(s)
while (!Q.empty())
  u = Q.remove()
  if (!u.visited)
    u.visited = true
    for (each edge (u,v) in E)
      Q.add(v)
```

Computing MST

Borůvka's Algorithm

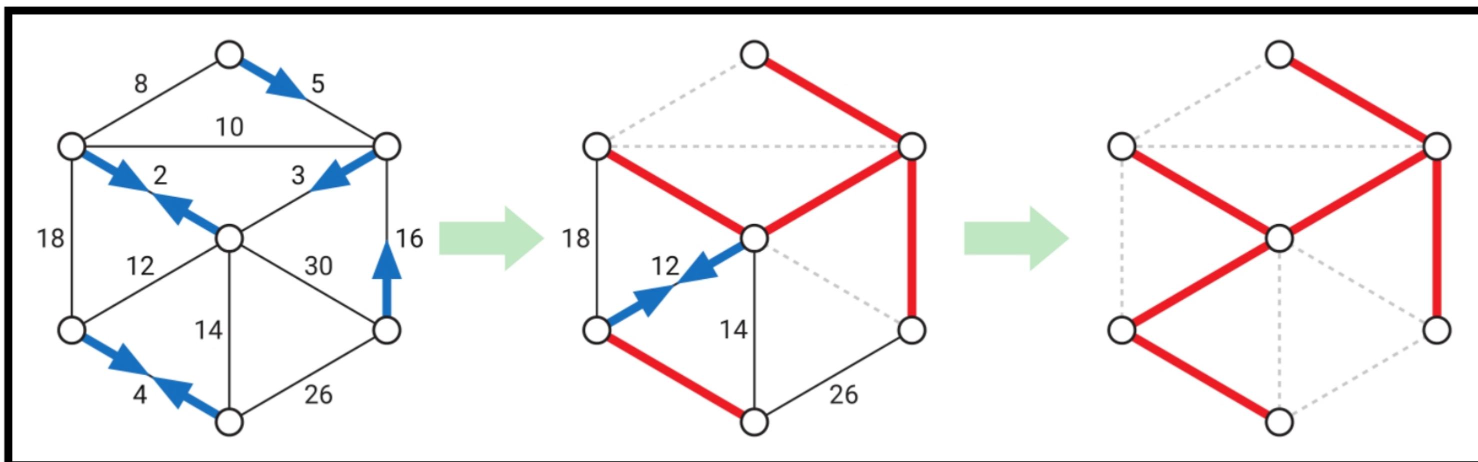
- **Prim's algorithm for computing MST:**
  - Starting with all nodes and an empty set of edges A.
  - Find MWOE in one *fixed* CC in G_A.
  - Repeat above step until we have a spanning tree.
- **Borůvka's algorithm for computing MST:**
  - Starting with all nodes and an empty set of edges A.
  - Find MWOE for *every* remaining CC in G_A, add *all* of them to A.
  - Repeat above step until we have a spanning tree.
- Borůvka's algorithm is actually the oldest MST algorithm.



Computing MST

Borůvka's Algorithm

- **Borůvka's algorithm for computing MST:**
  - Starting with all nodes and an empty set of edges A.
  - Find MWOE for *every* remaining CC in G_A, add *all* of them to A.
  - Repeat above step until we have a spanning tree.
- Is it okay to add multiple edges simultaneously?
- **Yes!** Assuming all edge weights are distinct, if CC C_1 propose MWOE e_1 to connect to C_2, and C_2 propose MWOE e_2 to connect to C_1, then $e_1 = e_2$.



Computing MST

Borůvka's Algorithm

- **Borůvka's algorithm for computing MST:**
  - Starting with all nodes and an empty set of edges A.
  - Find MWOE for *every* remaining CC in G_A, add *all* of them to A.
  - Repeat above step until we have a spanning tree.

BoruvkaMST(G,w):

Total runtime is $O(m \lg n)$.

```
G' = (V,∅)
do
  ccCount = CountCCAndLabel(G')
  for (i=1 to ccCount)
    safeEdge[i] = NIL
  for (each edge (u,v) in E(G))
    if (u.ccNum != v.ccNum)
      if (safeEdge[u.ccNum]==NIL or w(u,v)<w(safeEdge[u.ccNum]))
        safeEdge[u.ccNum] = (u,v)
      if (safeEdge[v.ccNum]==NIL or w(u,v)<w(safeEdge[v.ccNum]))
        safeEdge[v.ccNum] = (u,v)
  for (i=1 to ccCount)
    Add safeEdge[i] to E(G')
while (ccCount > 1)
return E(G')
```

`ccCount = CountCCAndLabel(G')`: $O(n)$ // **Do DFS/BFS, count # of CC, give `ccNum` to nodes.**

`for (i=1 to ccCount) safeEdge[i] = NIL`: $O(n)$

`for (each edge (u,v) in E(G)) ...`: $O(n+m) = O(m)$

`for (i=1 to ccCount) Add safeEdge[i] to E(G')`: $O(n)$

`while (ccCount > 1)`: $O(\lg n)$ iterations.

Borůvka's Algorithm

- **Borůvka's algorithm for computing MST:**
  - Starting with all nodes and an empty set of edges A.
  - Find MWOE for *every* remaining CC in G_A, add *all* of them to A.
  - Repeat above step until we have a spanning tree.

- Why Borůvka's algorithm is interesting?
  - Borůvka's algorithm allows for parallelism naturally;
    while the other two are intrinsically sequential.
    (Can be implemented in distributed/parallel computing systems.)
  - Generalizations of Borůvka's algorithm lead to faster algorithms.

Summary

- The "**Cut Property**" leads to many MST algorithms:
  Assume A is included in some MST, let $(S, V - S)$ be any cut respecting A.
  If (u, v) is a light edge crossing the cut, then (u, v) is safe for A.

- Classical algorithms for MST, all with runtime $O(m \cdot \log n)$:
  - **Kruskal** (UnionFind): keep connecting two CC with min-weight edge.
  - **Prim** (PriorityQueue): grow single CC by adding MWOE.
  - **Borůvka**: add MWOE for all CC in parallel in each iteration.

- Current best-known algorithm runs in $O(m \cdot \alpha(m, n))$.
  - Developed by *Bernard Chazelle* in 2000.

- Can we do MST in $O(m)$ time?
  - Randomized algorithm with expected $O(m)$ runtime exists.
  - Worst-case $O(m)$ runtime?

Reading

- [CLRS] Ch.23
- If you want to know more about Borůvka's MST algorithm: [Erickson v1] Ch.7 (7.3)