最小生成树
Minimum Spanning Trees

钮鑫涛
Nanjing University

2024 Fall

The slides are mainly adapted from the original ones shared by Chaodong Zheng and Kevin Wayne. Thanks for their supports!

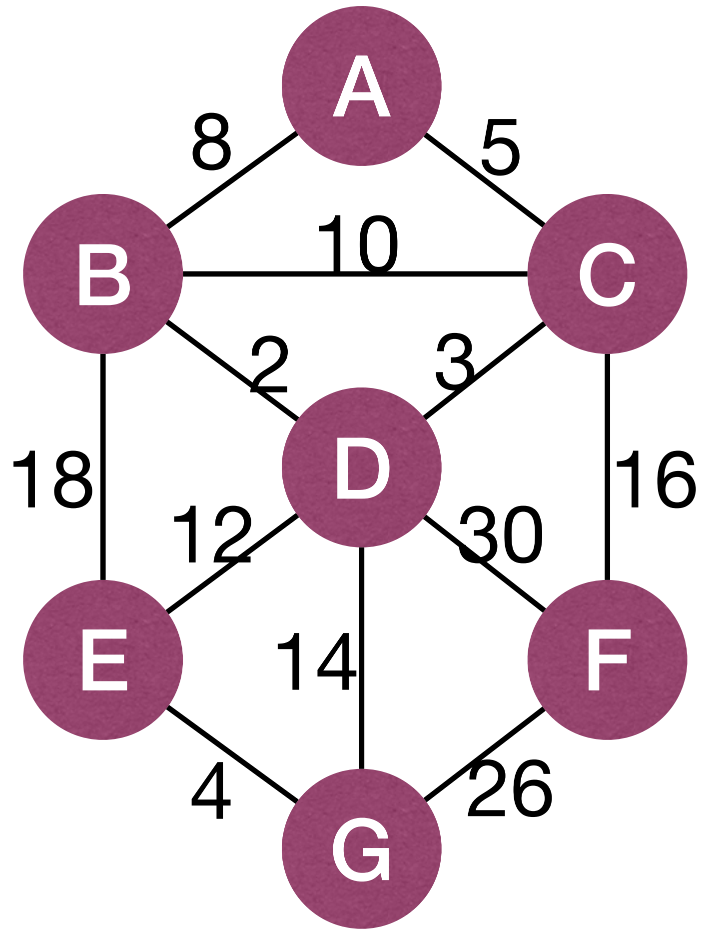
Minimum Spanning Trees (MST)

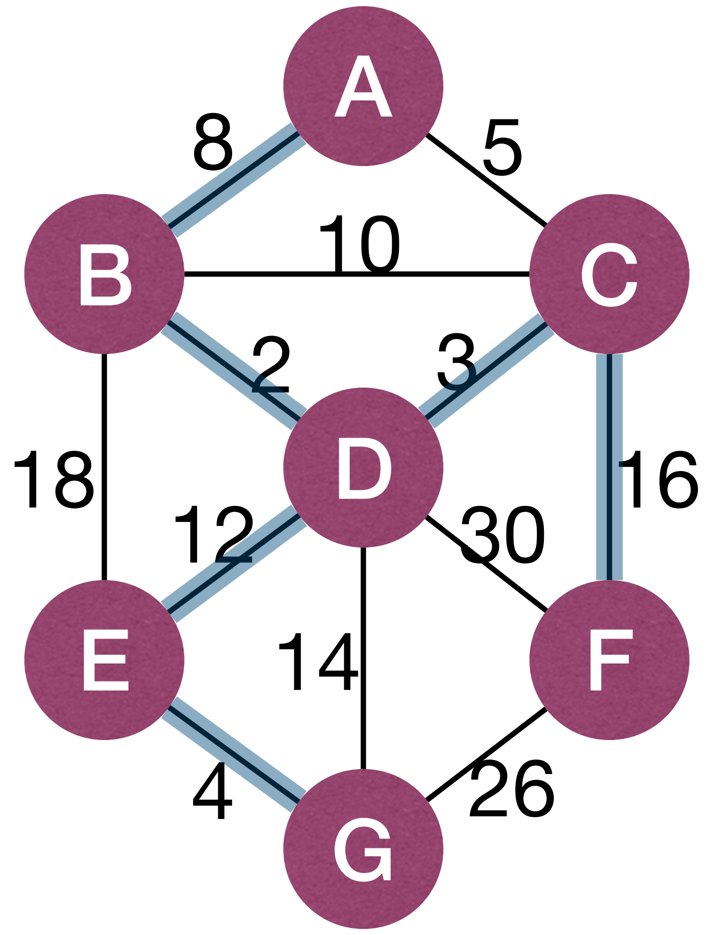
- Consider a connected, undirected, ***weighted*** graph G.

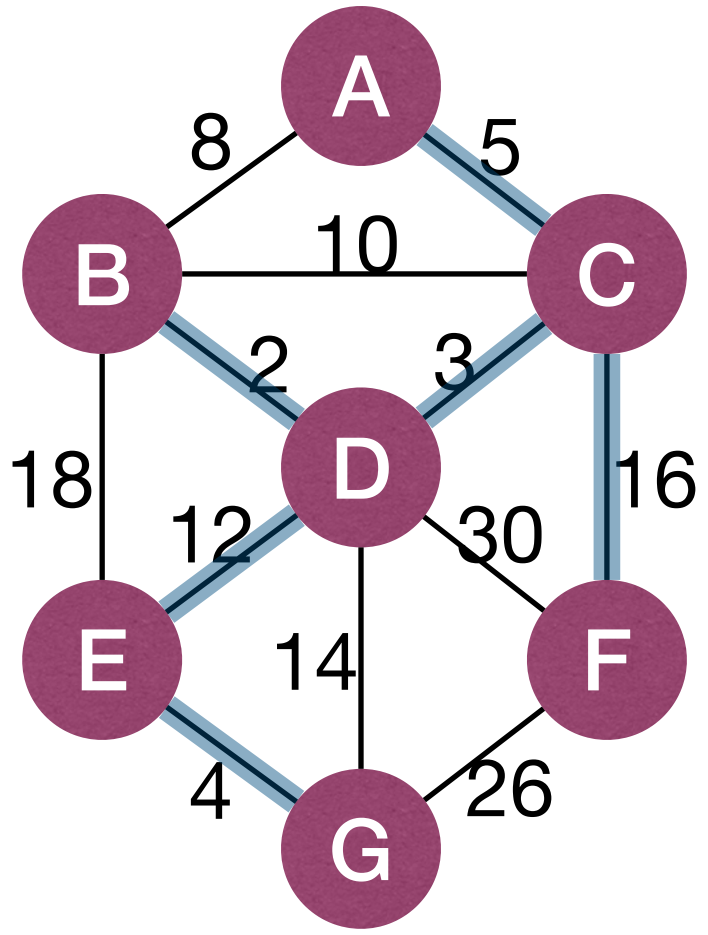
- That is, we have a graph $G = (V, E)$ together with a weight function $w : E \to \mathbb{R}$ that assigns a real weight $w(u, v)$ to each edge $(u, v) \in E$.

- 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.
    - One such application is 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 loop 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 are called "**safe edges**", how to identify them?

Easy to determine, e.g., $|A| = n - 1$

GenericMST(G,w):

$A := \varnothing$

while *A is **not** a spanning tree*

  $(u,v) :=$ *find_a_edge_maintaining_the_loop_invariant*()

  $A := A \cup \{(u, v)\}$

return A

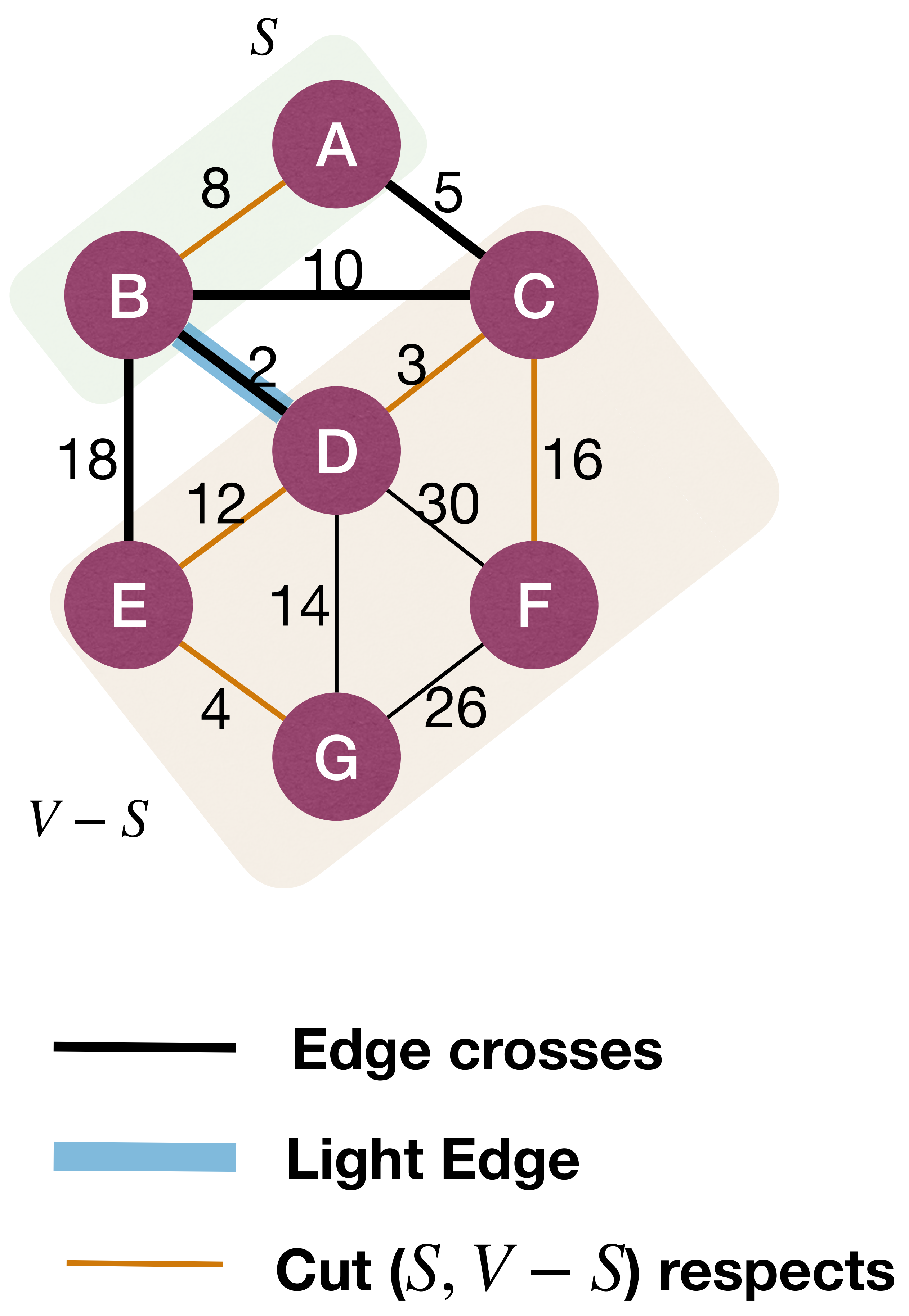
Identifying Safe Edges

- 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.

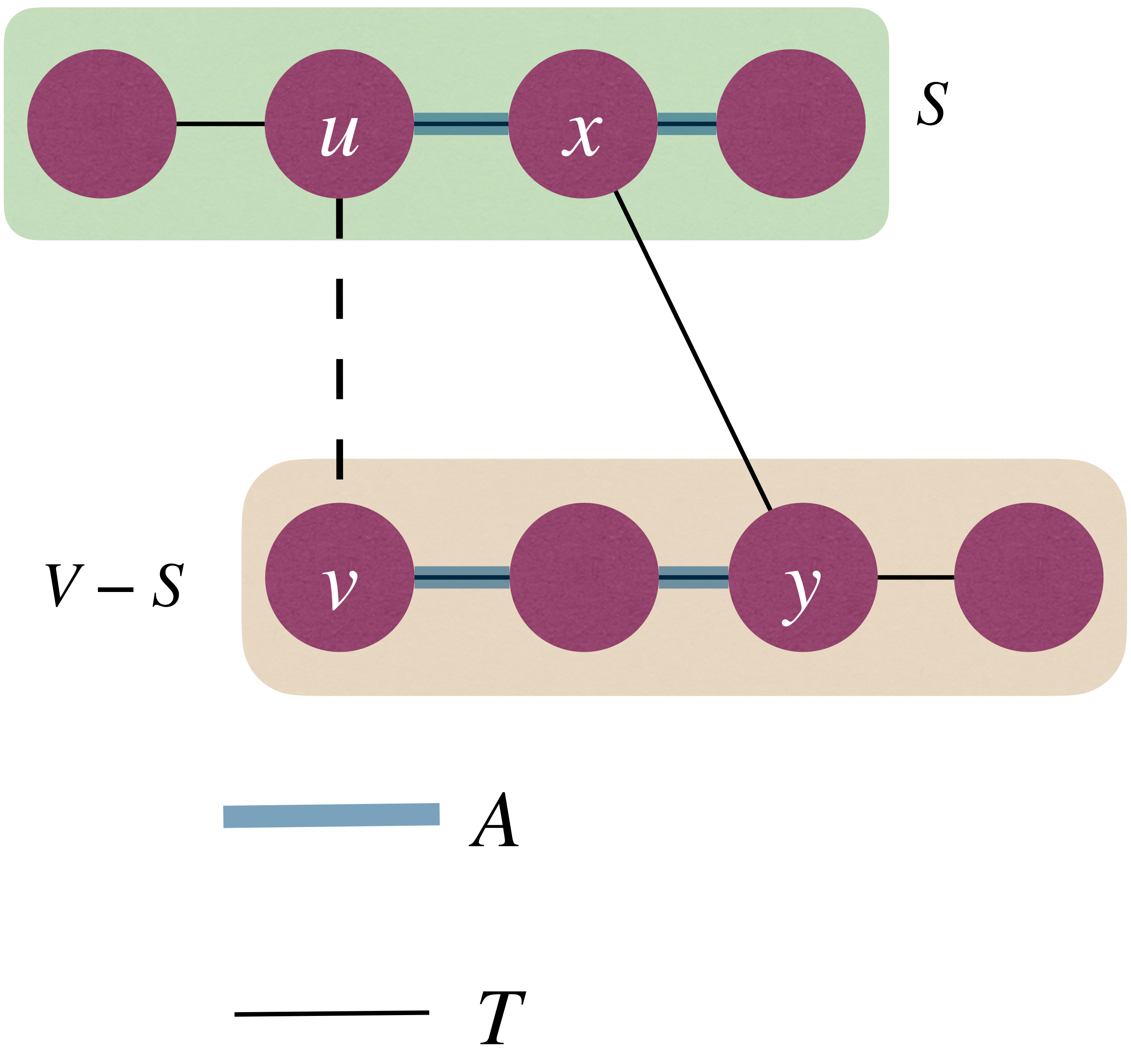
- An edge is a **light edge** crossing a cut if the edge has minimum weight among all edges crossing the cut.



Identifying Safe Edges

Theorem [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 an MST containing A, assume T does not include (u, v).
  - Connecting (u, v) forms a cycle in T, and in that cycle some edge other than (u, v) crosses the cut. Let $(x, y) \in T$ be that edge.
  - $T' = T - (x, y) + (u, v)$ must be a spanning tree.
  - Since (u, v) is a light edge crossing the cut, T' must be an MST, and (u, v) is safe for A in T'.



Computing MST

Theorem [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.

GenericMST(G,w):

$A := \varnothing$
while A *is* ***not*** *a spanning tree*
    $(u,v) :=$ ***find_a_safe_edge***()
    $A := A \cup \{(u, v)\}$
return 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

Kruskal's Algorithm

- Cut property: Assume A is included in some MST, let $G_A = (V, A)$. Then for any connected component in G_A, its MWOE 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.

Joseph Kruskal

<u>KruskalMST(G,w):</u>

$A := \emptyset$
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
    $A := A \cup \{(u, v)\}$
return A

- **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 reduced by 1.)
  - Repeat until one CC remains.

Kruskal's Algorithm

- Eden weights in increasing order: 2 3 4 5 8 10 12 14 16 18 26 30

Kruskal's Algorithm

KruskalMST(G,w):
$A := \varnothing$
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***
        $A := A \cup \{(u, v)\}$
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)`.

Kruskal's Algorithm

<u>KruskalMST(G,w):</u>

$A := \varnothing$

Sort edges into weight increasing order → $O(m \log m) = O(m \log n)$ ($m \leq n^2$)

for *each node u in V*
    MakeSet(u)
→ $O(n)$

for each *edge* (u,v) *taken in weight increasing order*
    if $Find(u) \mathrel{!=} Find(v)$
        $A := A \cup \{(u, v)\}$
        Union(u, v)
→ $O(m \log^* n)$

return A

- Runtime of Kruskal's algorithm?
  - $O(m \log n)$ when using disjoint-set data structure

Prim's Algorithm

- Strategy for finding safe edge in Prim's algorithm: Keep finding MWOE in one fixed CC in G_A.

PrimMST(G,w):

$A := \emptyset$

$C_x := \{x\}$

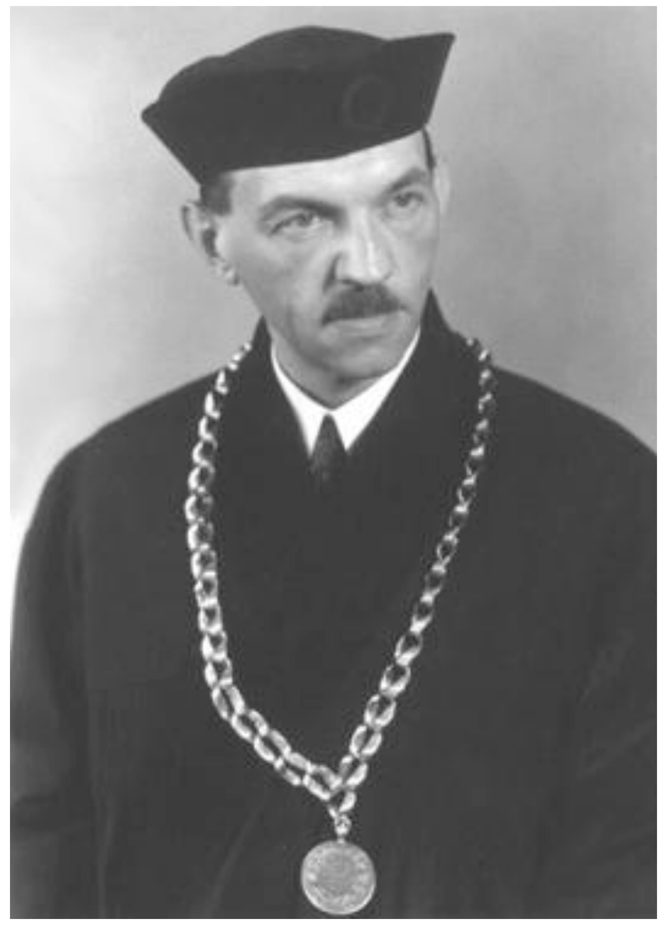
while C_x *is **not** a spanning tree*

  Find *MWOE* (u, v) of C_x

  $A := A \cup \{(u, v)\}$

  $C_x := C_x \cup \{v\}$

return A



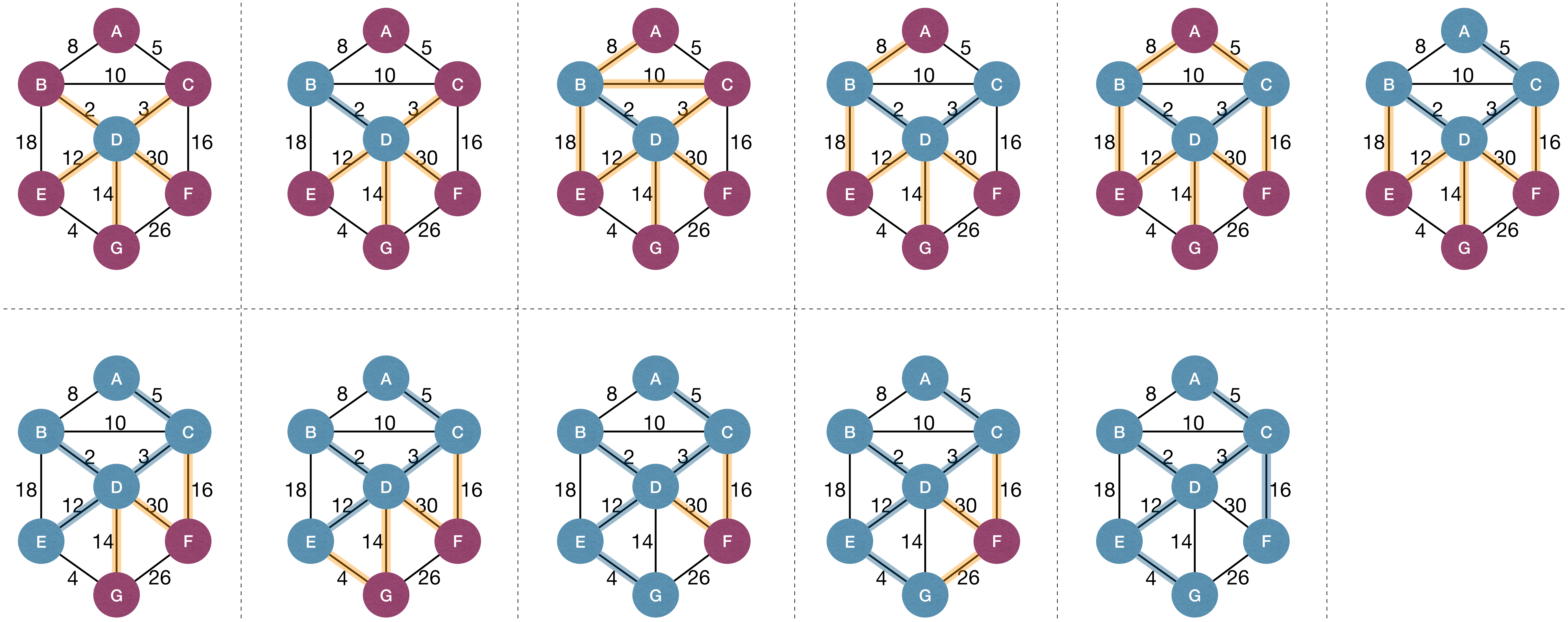
Vojtěch Jarník

Robert C. Prim

Edsger W. Dijkstra

- **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 reduced by 1.)
  - Repeat until one CC remains.

Prim's Algorithm



Prim’s Algorithm

PrimMST(G,w):

$A := \emptyset$

$C_x := \{x\}$

while C_x *is* ***not*** *a spanning tree*

  Find MWOE (u, v) of C_x

  $A := A \cup \{(u, v)\}$

  $C_x := C_x \cup \{v\}$

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.

Prim's Algorithm

PrimMST(G,w): → $O(m \lg n)$ using binary heap to implement priority queue

$x :=$ *Pick an arbitrary node in G*

for each *node u* **in** *V* → $O(n)$

  $u.dist := INF,\ u.parent := NIL,\ u.in := False$

$x.dist := 0$

PriorityQueue *Q := Build a priority queue based on "dist" values* → $O(n)$

while *Q is* **not** *empty*

  $u := Q.ExtractMin()$ → $O(n \lg n)$

  $u.in := True$

  for each *edge (u,v)* **in** *E* → $O(m \lg n)$

    if $v.in = False$ **and** $w(u,v) < v.dist$

      $v.parent := u,\ v.dist := w(u,v)$

      $Q.Update(v, w(u,v))$

Could be faster using better priority queue implementation (By using fibonacci heaps instead)

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)
```

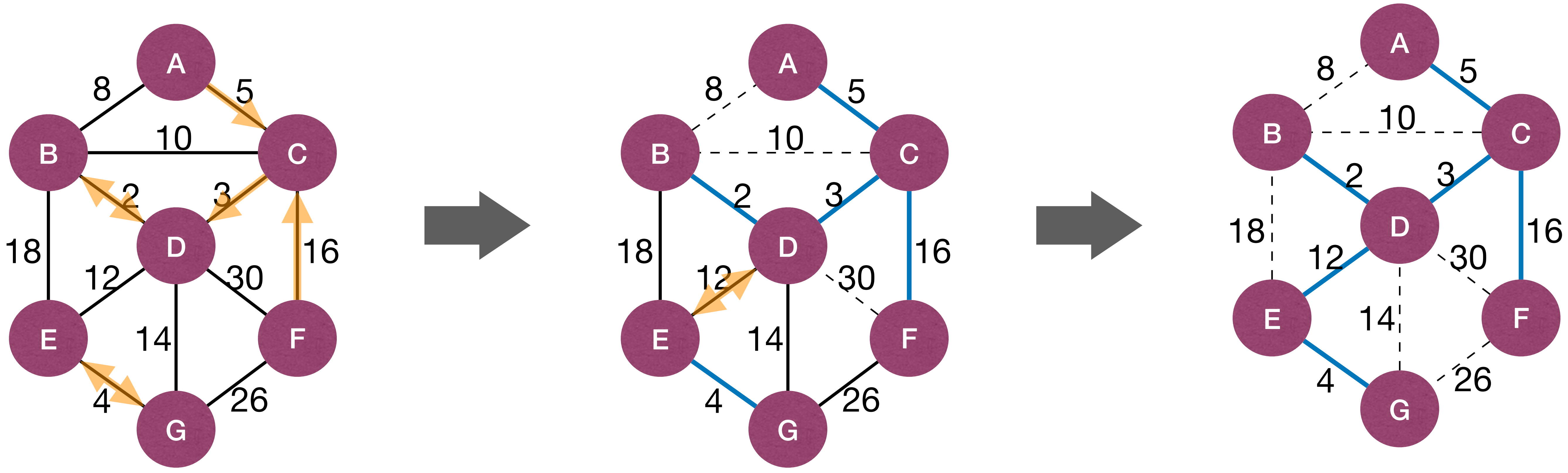
```
PrimMSTSkeleton(G, x):
PriorityQueue Q
Q.add(x)
while !Q.empty()
    u := Q.remove()
    if  !u.visited
        u.visited := True
        for each edge (u, v) in E
            if !v.visited and …
                Q.update(v, …)
```

```
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)
```

Borůvka's Algorithm

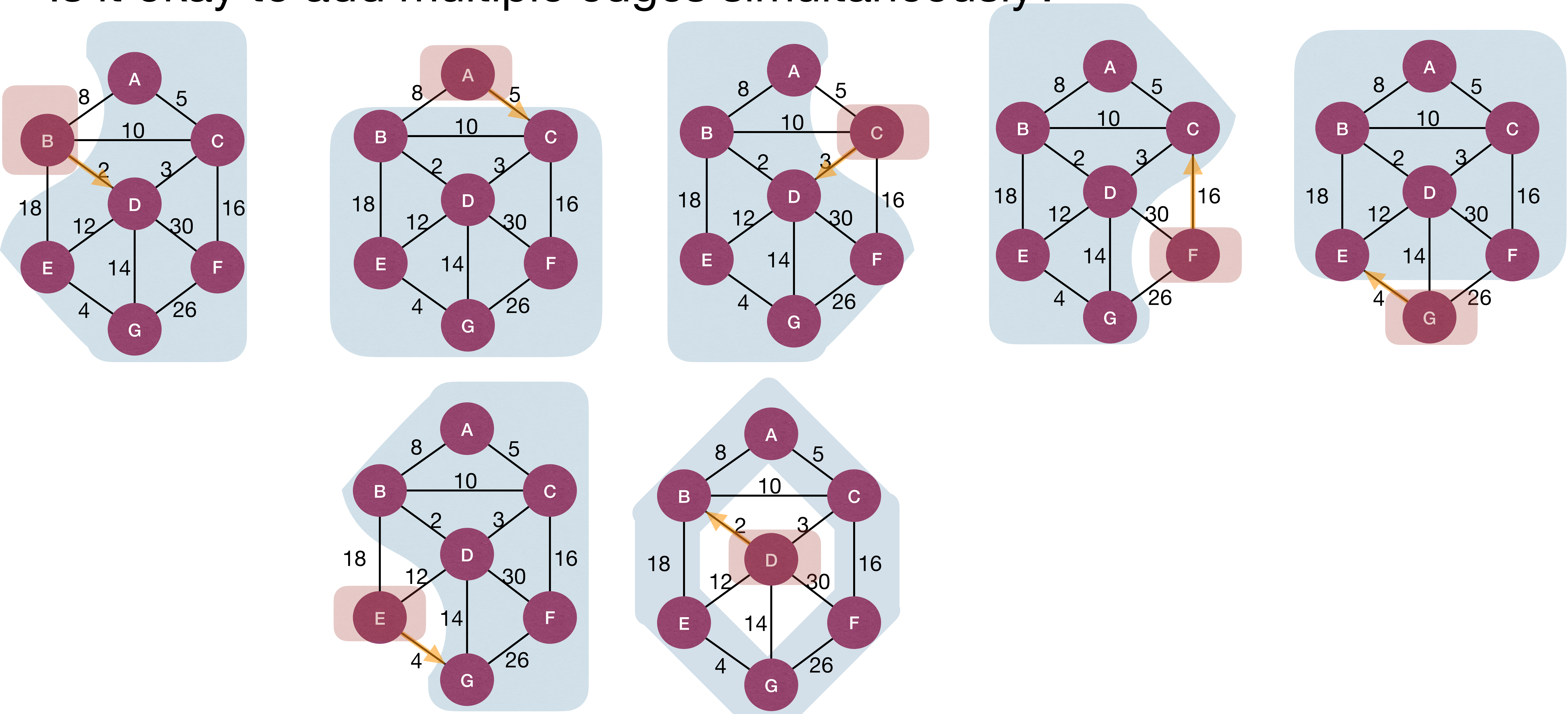
- Borůvka's algorithm for computing MST (actually the <u>earliest</u> MST algorithm):
  - Starting with all nodes and an empty set of edges A.
  - Find MWOE for every remaining CC in G_A, add all of them to A.
  - Repeat above step until we have a spanning tree.

Otakar Borůvka



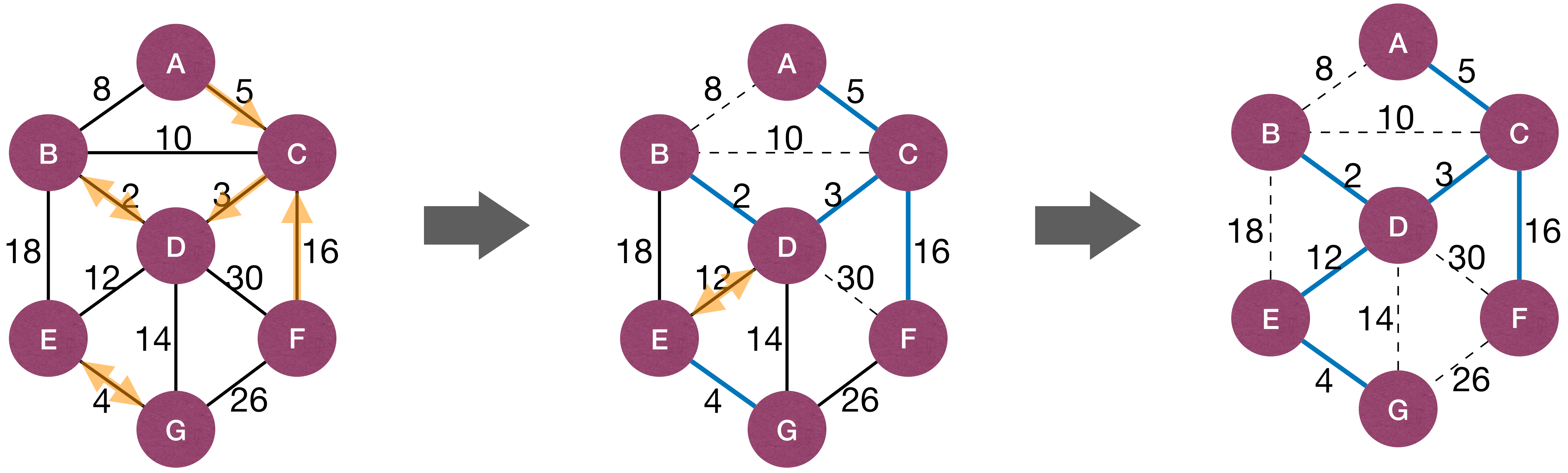
Borůvka's Algorithm

- Is it okay to add multiple edges simultaneously?



Borůvka's Algorithm

- Is it okay to add multiple edges simultaneously?
- But it may result in circles?
    - Assuming all edge weights are distinct, if CC C_1 propose MWOE e_1 to connect to C_2, and C_2 proposes MWOE e_2 to connect to C_1, then $e_1 = e_2$.



Borůvka's Algorithm

KruskalMST(G,w): → Total runtime is $O(m \lg n)$

$G' := (V, \varnothing)$

do

  $ccCount := CountCCAndLabel(G')$ → $O(n)$ //Do DFS/BFS, count #of CC, give **ccNum** to nodes. (belong to the $ccNum^{th}$ CC)

  for $i := 1$ **to** $ccCount$

    $safeEdge[i] := NIL$

  → $O(n)$

  for **each** *edge* (u,v) **in** $E(G)$

    if $u.ccNum \mathrel{!}= v.ccNum$

      if $safeEdge[u.ccNum] = NIL$ **or** $w(u,v) < w(safeEdge[u.ccNum])$

        $safeEdge[u.ccNum] := (u,v)$

      if $safeEdge[v.ccNum] = NIL$ **or** $w(u,v) < w(safeEdge[v.ccNum])$

        $safeEdge[v.ccNum] := (u,v)$

  → $O(m+n) = O(m)$

  for $i := 1$ **to** $ccCount$

    Add $safeEdge[i]$ **to** $E(G')$

  → $O(n)$

while $ccCount > 1$ → $O(\lg n)$ interactions ← WHY?

return $E(G')$

Borůvka’s Algorithm

- Why Borůvka’s algorithm is interesting?
  - The number of components in G' can drop by significantly more than a factor of 2 in a single iteration, reducing the number of iterations below the worst-case $O(\lg n)$.
  - Borůvka’s algorithm allows for parallelism naturally; while the other two are intrinsically sequential.
  - 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.

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

Further reading

- [CLRS] Ch.23
- [Erickson] Ch.7