Greedy algorithms MST

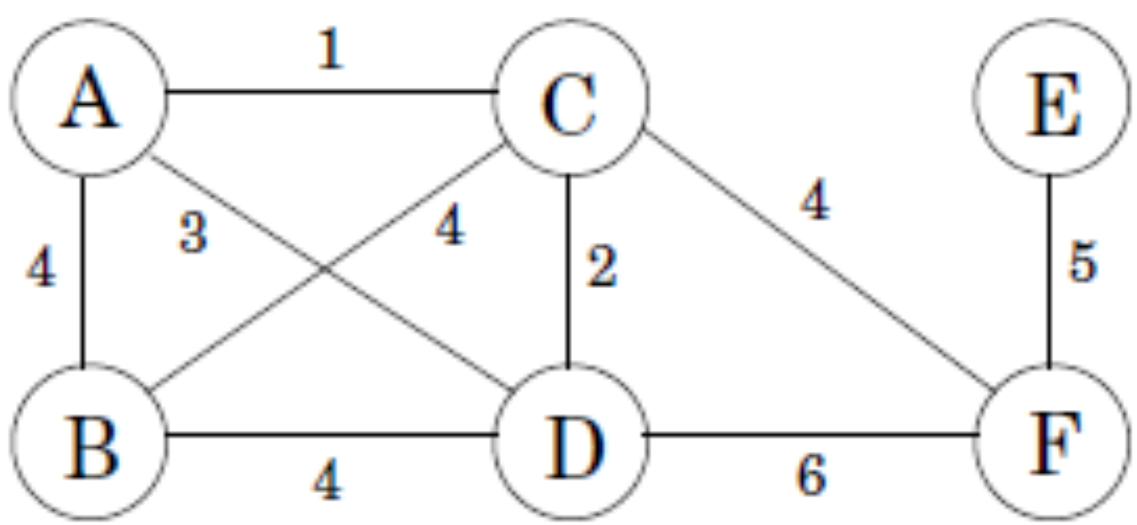
Gou Guanglei(苟光磊)
ggl@cqut.edu.cn

Greedy algorithms

- Algorithm design paradigm
- Idea : when we have a choice to make, make the one that looks best right now. Make a **locally optimal choice** in hope of **getting a globally optimal solution.**

Problem

- Network a collection of computers by linking selected pairs of them.
- Each link has a maintenance cost.
- What is the cheapest network?

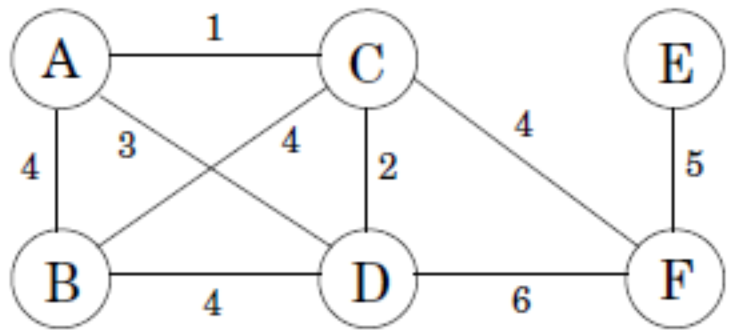


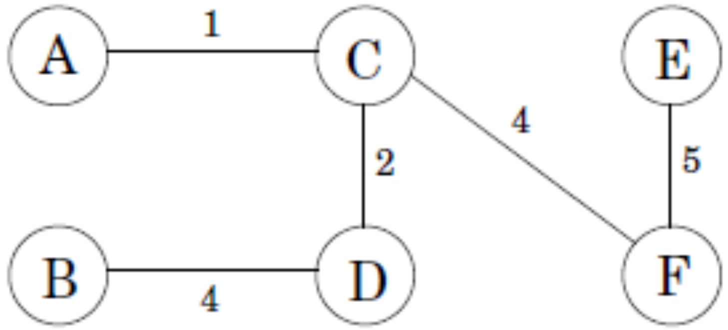
Minimum spanning tree (MST)

- Removing a cycle edge cannot disconnect a graph.
- So the solution must be connected and acyclic : undirected graphs of this kind are called *trees*.

Input: *An undirected graph $G = (V, E)$, edge weights w_e*

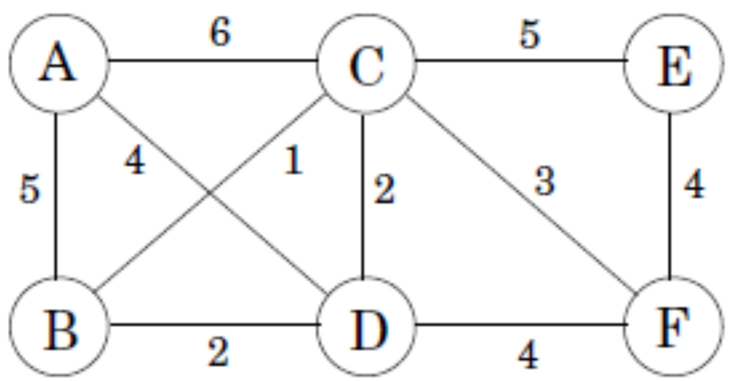
Output: *A tree $T = (V, E')$, with $E' \subseteq E$, that minimizes weight(T) =* $\sum_{e \in E'} w_e$

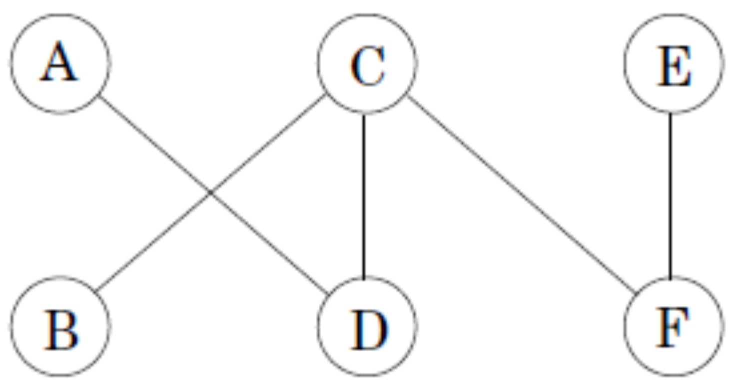




Kruskal’s algorithm

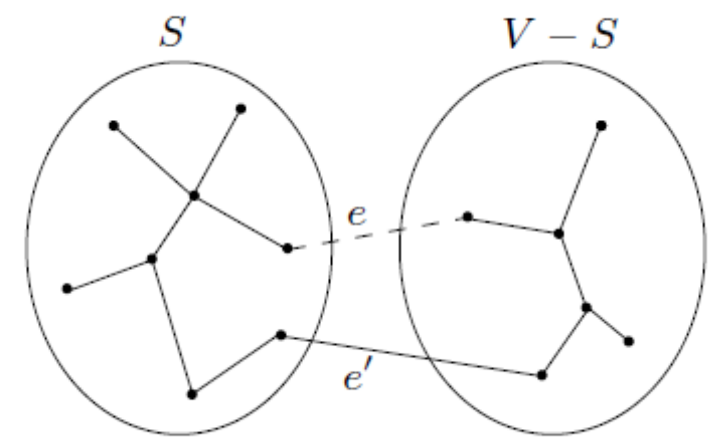
- Kruskal's minimum spanning tree algorithm starts with the empty graph and then selects edges from E according to the following rule.
- Repeatedly add the next ***lightest* edge** that **doesn't produce a cycle**.
- This is a *greedy algorithm*





Cut property

- Suppose edges X are part of a minimum spanning tree of G = (V, E)
- Pick any subset of nodes S for which X does not cross between S and V-S, and let *e* be the lightest edge across this partition.
- Then, $X \cup \{e\}$ is part of some MST.



Pf) Let T be an MST that includes X.

If *e* is in T, done.

Otherwise, add *e* to T. It creates a cycle.

This cycle must have another edge *e*' across the cut (S, V-S).

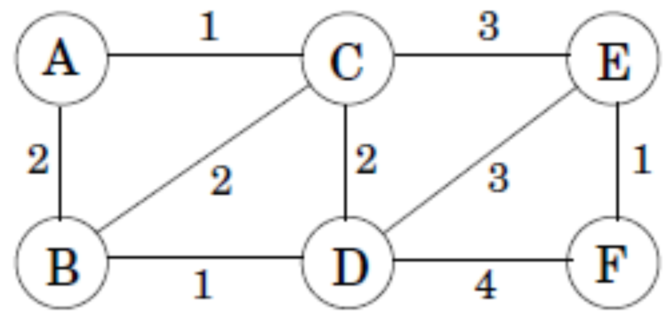
Remove *e*'. Then, we have a new spanning tree $T' = T \cup \{e\} - \{e'\}$. (why is T' a spanning tree?)

weight(T') = weight(T) +w(*e*) – w(*e*')

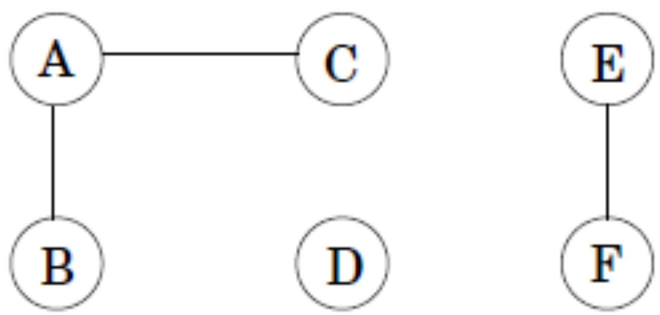
Since e is the lightest edge crossing the cut (S, V-S), $w(e) \leq w(e')$

Thus, weight(T') ≤ weight(T).

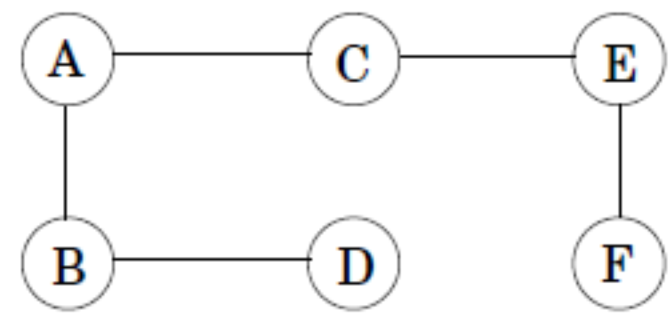
Since T is a MST, T' is also a MST.



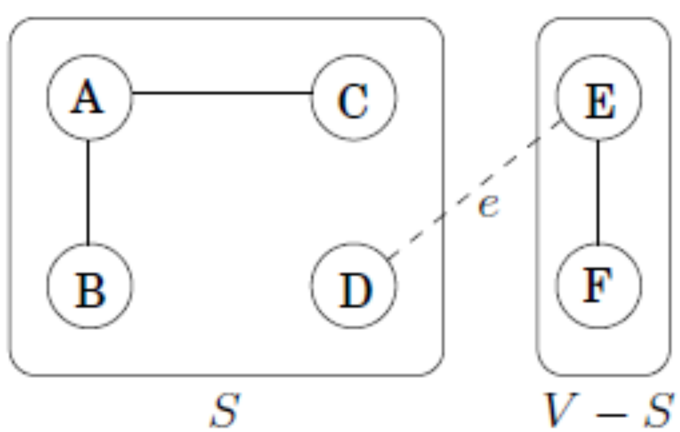
Edges X:



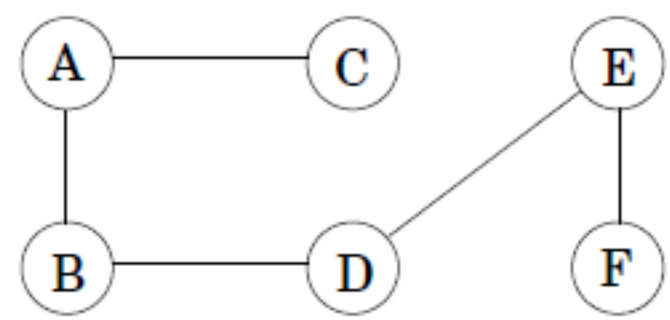
MST T:



The cut:



MST T':



- <u>Kruskal's algorithm</u>

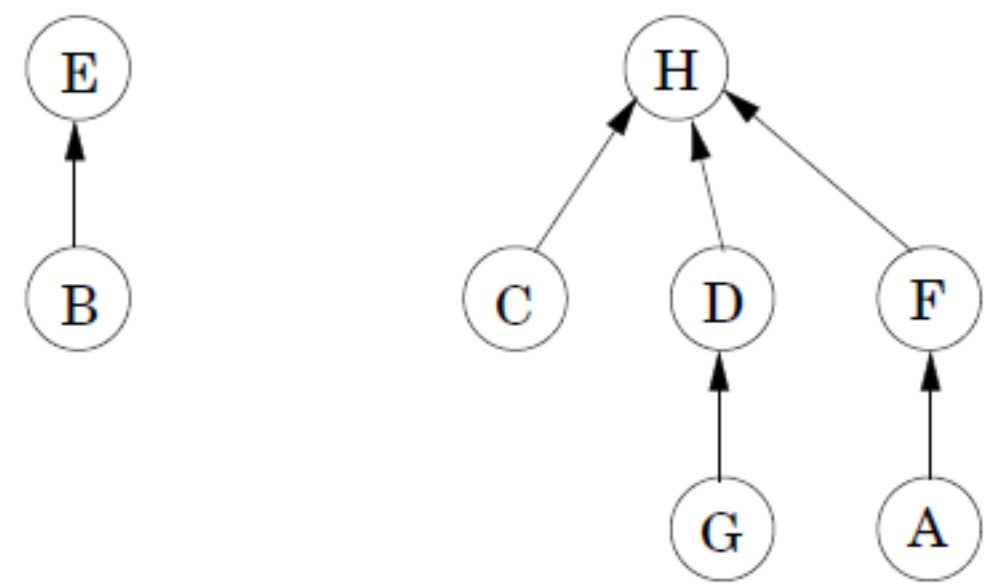
```
procedure kruskal(G, w)
Input:    A connected undirected graph G = (V, E) with edge weights w_e
Output:   A minimum spanning tree defined by the edges X

for all u ∈ V:
   makeset(u)

X = {}
Sort the edges E by weight
for all edges {u, v} ∈ E, in increasing order of weight:
   if find(u) ≠ find(v):
      add edge {u, v} to X
      union(u, v)
```

Uses |V| makeset, 2|E| find, |V|-1 union operations.

- Disjoint-set data structure
  - Store a set as a directed tree.
  - Nodes of the tree are elements of the set, arranged in no particular order.
  - Each has parent pointers π that eventually lead up to the root of the tree.
  - The root is a *representative*, or *name*, for the set.
  - The root has a parent pointer π pointing itself.
  - Each node has *rank* representing the height of the subtree from the node.



- Makeset and find
  - *makeset* is a constant-time operation
  - *find* follows parent pointers to the root of the tree : takes O(height of the tree).

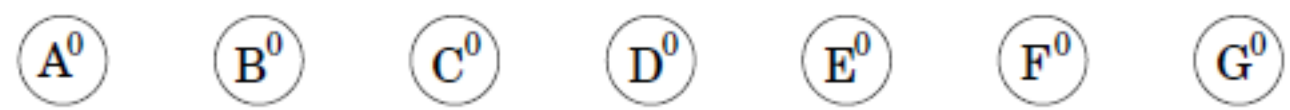
```
procedure makeset(x)
π(x) = x
rank(x) = 0

function find(x)
while x ≠ π(x):  x = π(x)
return x
```

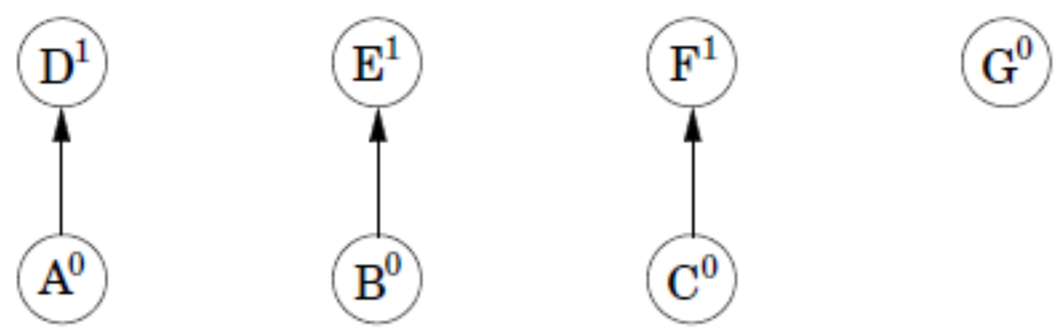
- <u>Union by rank</u>
  - Make the root of the shorter tree point to the root of the taller tree.
  - Then, the overall height increases only if the two trees being merged are equally tall.
  - Instead of explicitly computing heights of trees, we will use the rank numbers of their root nodes - **union by rank.**

<u>procedure union</u>(x, y)
$r_x = \texttt{find}(x)$
$r_y = \texttt{find}(y)$
if $r_x = r_y$: return
if $\texttt{rank}(r_x) > \texttt{rank}(r_y)$:
    $\pi(r_y) = r_x$
else:
    $\pi(r_x) = r_y$
    if $\texttt{rank}(r_x) = \texttt{rank}(r_y)$: $\texttt{rank}(r_y) = \texttt{rank}(r_y) + 1$

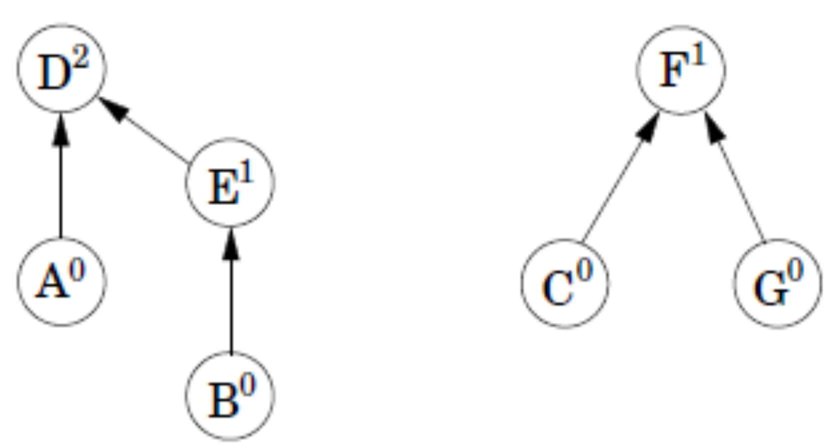
After $\texttt{makeset}(A), \texttt{makeset}(B), \ldots, \texttt{makeset}(G)$:



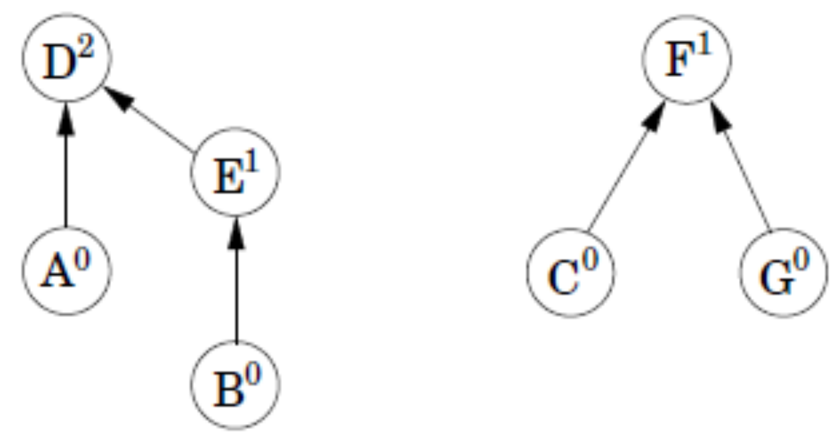
After $\texttt{union}(A, D), \texttt{union}(B, E), \texttt{union}(C, F)$:



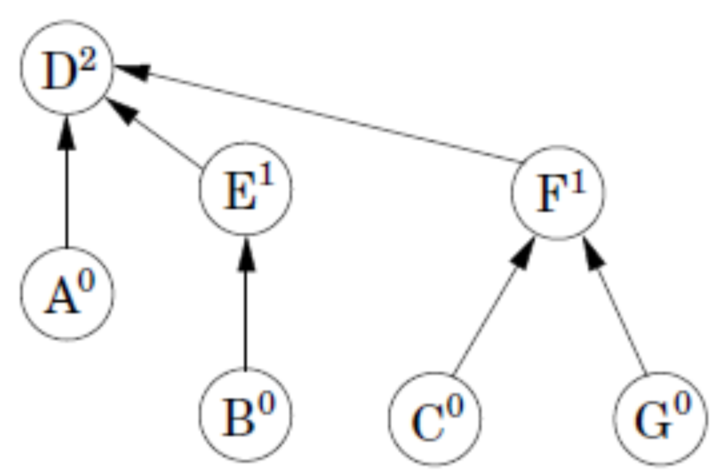
After $\texttt{union}(C, G), \texttt{union}(E, A)$:



After $\texttt{union}(C, G), \texttt{union}(E, A)$:



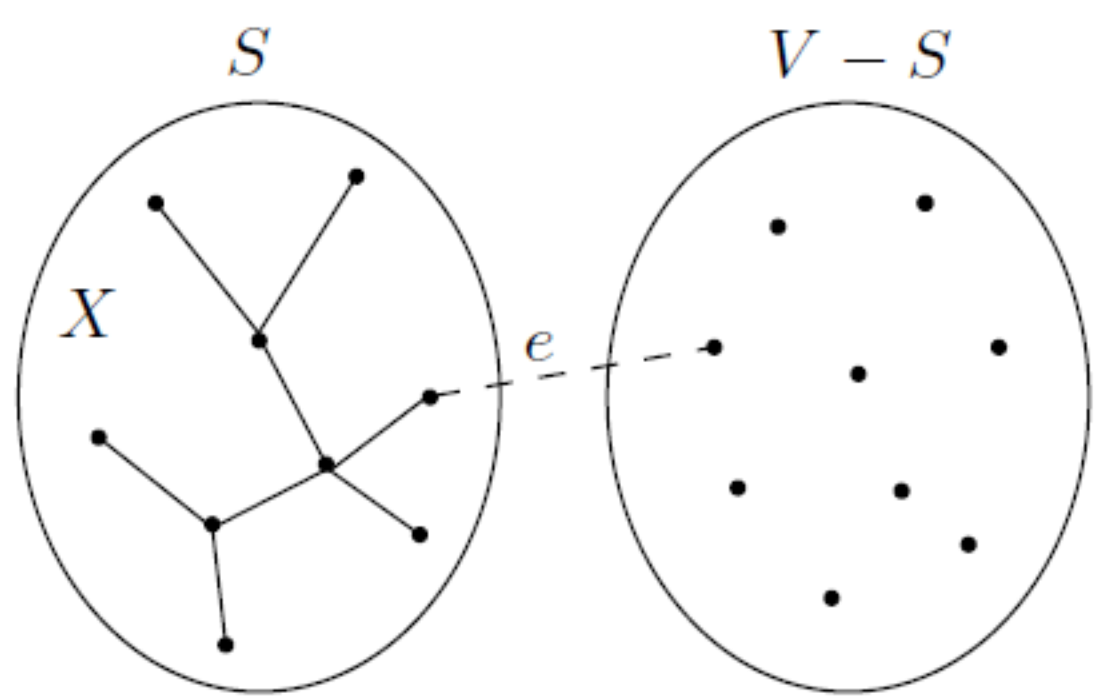
After $\texttt{union}(B, G)$:



- <u>Analysis of Kruskal's algorithm</u>
  - Kruskal's algorithm uses $|V|$ makeset, $2|E|$ find, $|V|-1$ union operations.
  - We need $O(|E| \lg |V|)$ to sort the edges. ($\lg |E| = \Theta(\lg |V|)$)
  - $O(|E| \log |V|)$ for find and union operations.

- <u>Prim's algorithm</u>

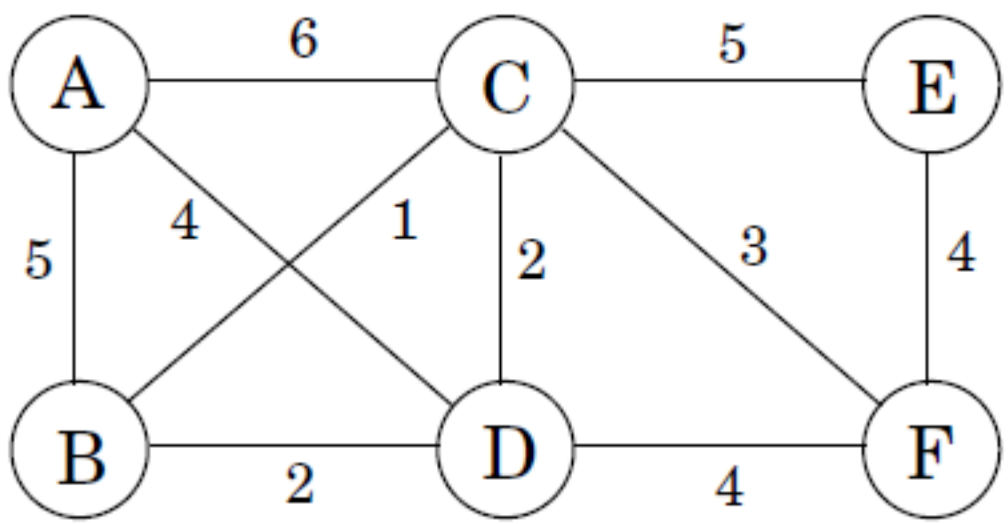
```
X = { } (edges picked so far)
repeat until |X| = |V| − 1:
  pick a set S ⊂ V for which X has no edges between S and V − S
  let e ∈ E be the minimum-weight edge between S and V − S
  X = X ∪ {e}
```

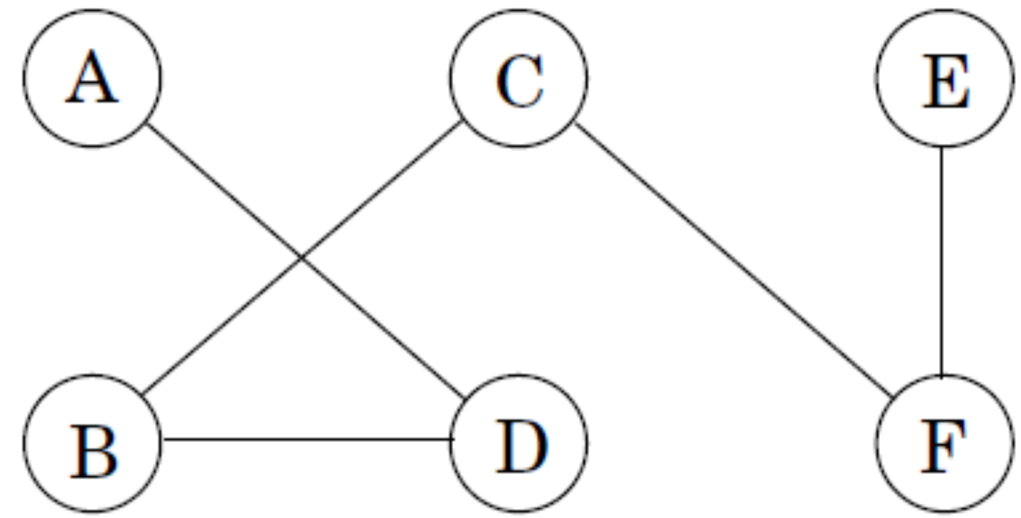


```
procedure prim(G, w)
Input:    A connected undirected graph G = (V, E) with edge weights w_e
Output:   A minimum spanning tree defined by the array prev

for all u ∈ V:
   cost(u) = ∞
   prev(u) = nil
Pick any initial node u_0
cost(u_0) = 0

H = makequeue(V)    (priority queue, using cost-values as keys)
while H is not empty:
   v = deletemin(H)
   for each {v, z} ∈ E:
      if cost(z) > w(v, z):
         cost(z) = w(v, z)
         prev(z) = v
         decreasekey(H, z)
```





Set S	A	B	C	D	E	F
$\{\}$	0/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
A		$5/A$	$6/A$	$4/A$	∞/nil	∞/nil
A, D		$2/D$	$2/D$		∞/nil	$4/D$
A, D, B			$1/B$		∞/nil	$4/D$
A, D, B, C					$5/C$	$3/C$
A, D, B, C, F					$4/F$	