Design and analysis of algorithms
Lecture 18 & 19

Edyta Szymańska

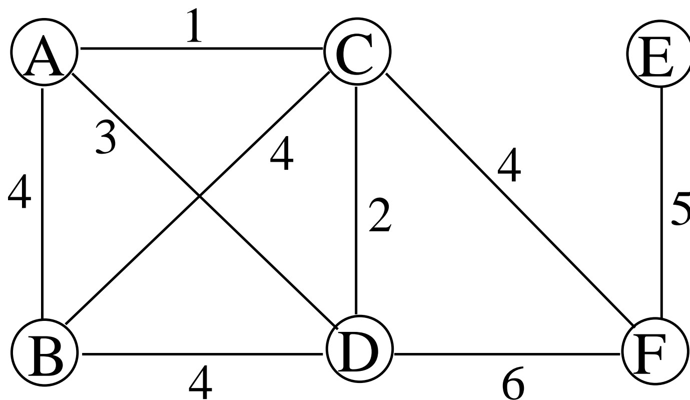
edyta@cc.gatech.edu

Minimum spanning tree (MST)

Problem: given a set of computers and a maintenance cost associated with every potential link, build a communication network which is connected and has the cheapest cost.

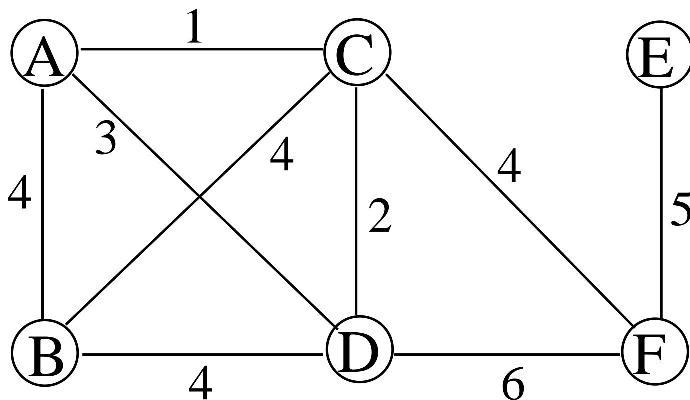
Minimum spanning tree (MST)

Problem: given a set of computers and a maintenance cost associated with every potential link, build a communication network which is connected and has the cheapest cost.



Minimum spanning tree (MST)

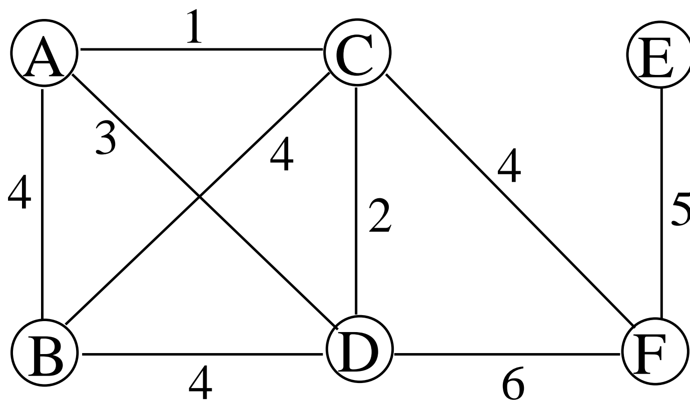
Problem: given a set of computers and a maintenance cost associated with every potential link, build a communication network which is connected and has the cheapest cost.



Observation: the optimal set of edges cannot contain a cycle.

Minimum spanning tree (MST)

Problem: given a set of computers and a maintenance cost associated with every potential link, build a communication network which is connected and has the cheapest cost.



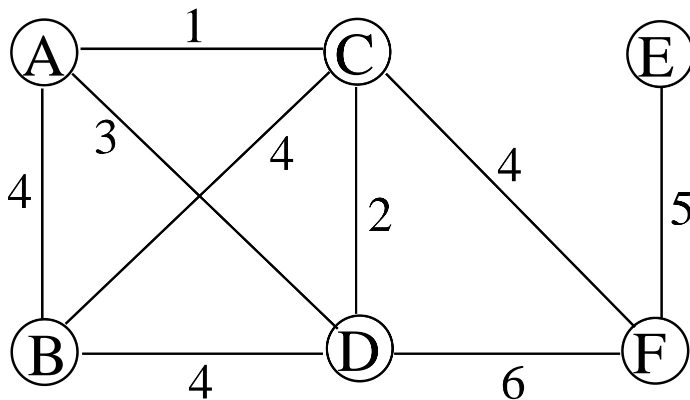
Observation: the optimal set of edges cannot contain a cycle.
Thus, the solution must be connected and acyclic - `TREE`.

Minimum spanning tree (MST)

INPUT: an undirected, connected graph $G = (V, E)$ *with edge weights* $w(e)$.

OUTPUT: a tree $T = (V, E')$ *spanning all the vertices of* G *and minimizing*

$$weight(T) = \sum_{e \in E'} w(e)$$

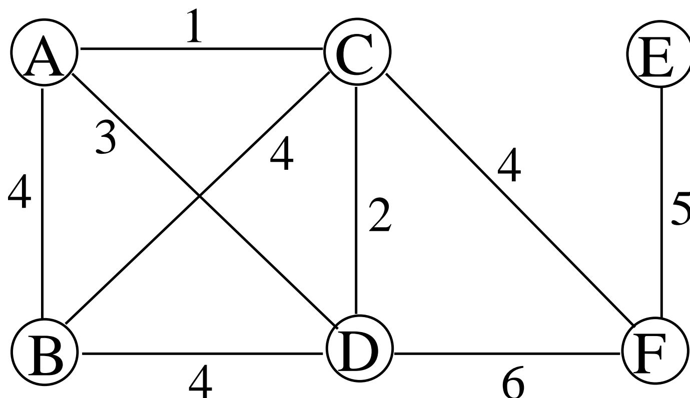


Minimum spanning tree (MST)

INPUT: an undirected, connected graph $G = (V, E)$ *with edge weights* $w(e)$.

OUTPUT: a tree $T = (V, E')$ *spanning all the vertices of* G *and minimizing*

$$weight(T) = \sum_{e \in E'} w(e)$$



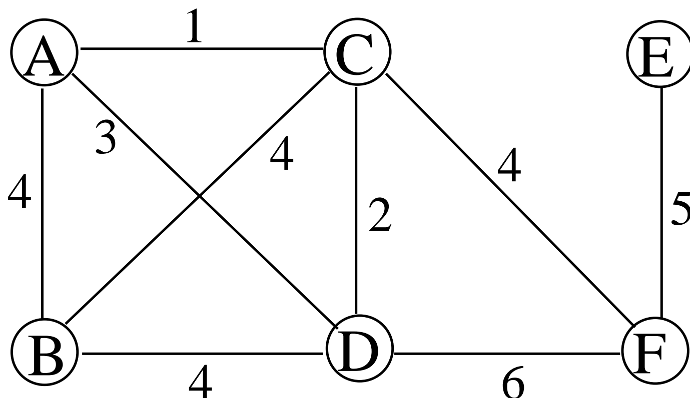
MST cost in the example above is 16.

Minimum spanning tree (MST)

INPUT: an undirected, connected graph $G = (V, E)$ *with edge weights* $w(e)$.

OUTPUT: a tree $T = (V, E')$ *spanning all the vertices of* G *and minimizing*

$$weight(T) = \sum_{e \in E'} w(e)$$



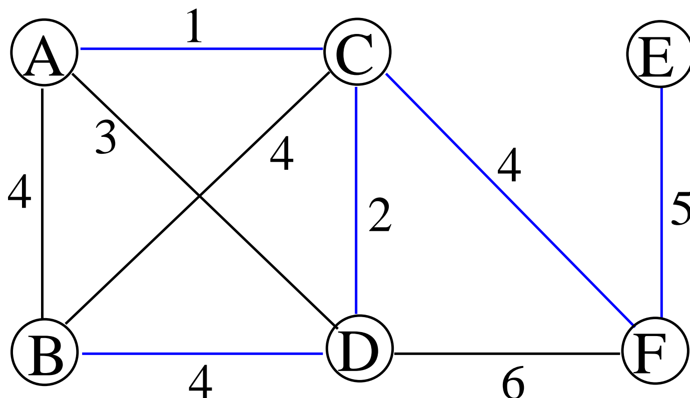
MST cost in the example above is 16.

Minimum spanning tree (MST)

INPUT: an undirected, connected graph $G = (V, E)$ *with edge weights* $w(e)$.

OUTPUT: a tree $T = (V, E')$ *spanning all the vertices of* G *and minimizing*

$$weight(T) = \sum_{e \in E'} w(e)$$



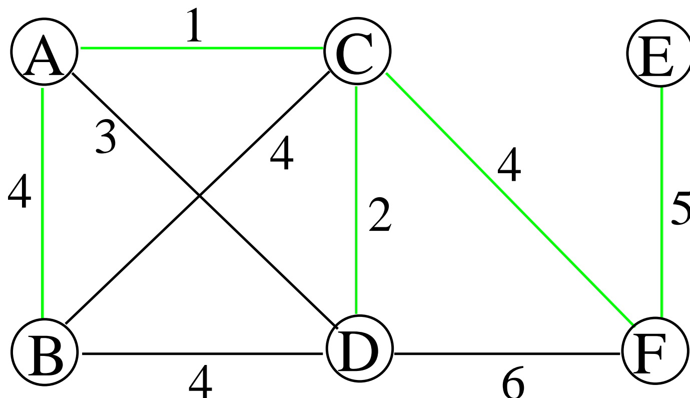
MST cost in the example above is 16. Is it unique ?

Minimum spanning tree (MST)

INPUT: an undirected, connected graph $G = (V, E)$ *with edge weights* $w(e)$.

OUTPUT: a tree $T = (V, E')$ *spanning all the vertices of* G *and minimizing*

$$weight(T) = \sum_{e \in E'} w(e)$$



MST cost in the example above is 16. Is it unique ?

Greedy approach

Algorithms ?

Greedy approach

Algorithms ?

"Greed ... is good. Greed is right. Greed works."

Greedy approach

Algorithms ?

"Greed ... is good. Greed is right. Greed works."

Kruskal's greedy algorithm:

build a spanning tree by successively adding edges in order of increasing weights and in such a way that they do not form a cycle with those edges already chosen.

Greedy approach

Algorithms ?

"Greed ... is good. Greed is right. Greed works."

Kruskal's greedy algorithm:

build a spanning tree by successively adding edges in order of increasing weights and in such a way that they do not form a cycle with those edges already chosen.

See example on the board.

Greedy approach

Algorithms ?

"Greed ... is good. Greed is right. Greed works."

Kruskal's greedy algorithm:

build a spanning tree by successively adding edges in order of increasing weights and in such a way that they do not form a cycle with those edges already chosen.

See example on the board.

This a **greedy** method: makes a decision at each step myopically to optimize the underlying criterion.

Why is Kruskal's algorithm correct ?

We will proof a more general property first:

Why is Kruskal's algorithm correct ?

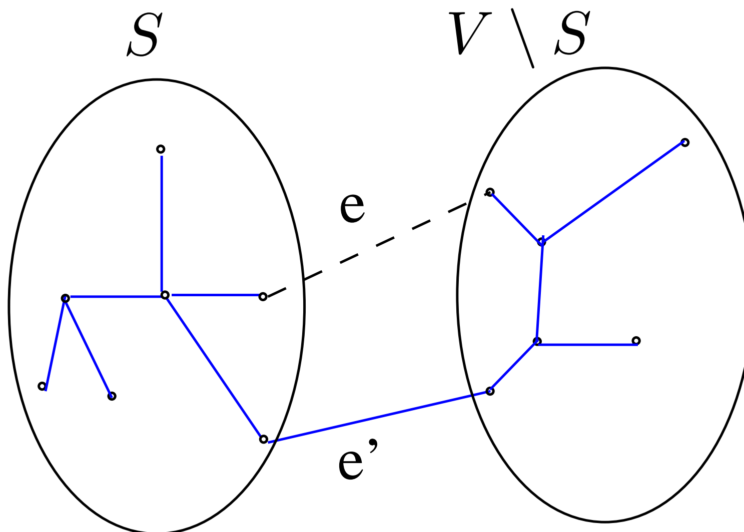
Cut property: *Let* $X \subseteq E$ *be part of some minimum spanning tree* T *of* $G = (V, E)$. *Pick any set of nodes* $S \subset V$ *such that there is no edge in* X *which connects a node in* S *to one in* $V \setminus S$. *If* $e \in E$ *is the minimum-weight edge between* S *and* $V - S$, *then* $X \cup \{e\}$ *is also part of some MST.*

Why is Kruskal's algorithm correct ?

Proof: Assume $e \notin T$. Then $T \cup \{e\}$ contains a cycle C.
Now C contains another edge e' across the cut $(S, V \setminus S)$.

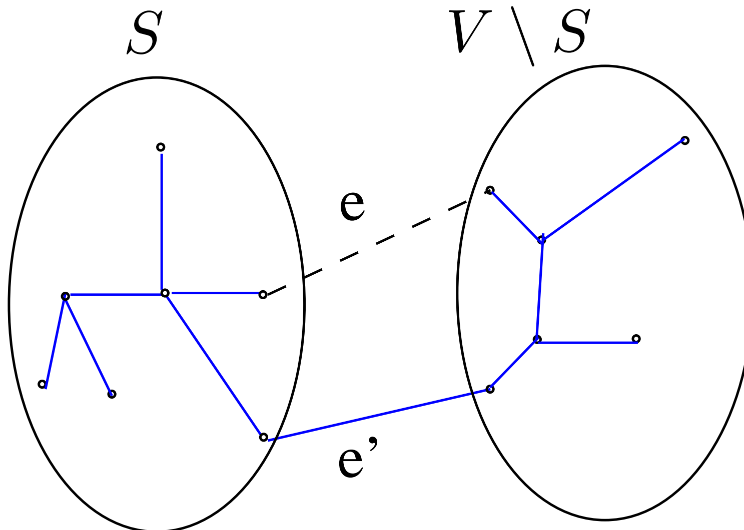
Why is Kruskal's algorithm correct ?

Proof: Assume $e \notin T$. Then $T \cup \{e\}$ contains a cycle C. Now C contains another edge e' across the cut $(S, V \setminus S)$.



Why is Kruskal's algorithm correct ?

Proof: Assume $e \notin T$. Then $T \cup \{e\}$ contains a cycle C. Now C contains another edge e' across the cut $(S, V \setminus S)$.



Consider $T' = T \cup \{e\} - \{e''\}$.It is a tree with $w(T') = w(T) + w(e) - w(e') \leq w(T)$, because $w(e) \leq w(e')$. Since T is MST, so is T'.

Why is Kruskal's algorithm correct ?

Cut property: *Let* $X \subseteq E$ *be part of some minimum spanning tree* T *of* $G = (V, E)$. *Pick any set of nodes* $S \subset V$ *such that there is no edge in* X *which connects a node in* S *to one in* $V \setminus S$. *If* $e \in E$ *is the minimum-weight edge between* S *and* $V - S$, *then* $X \cup \{e\}$ *is also part of some MST.*

Why is Kruskal's algorithm correct ?

Cut property: *Let* $X \subseteq E$ *be part of some minimum spanning tree* T *of* $G = (V, E)$. *Pick any set of nodes* $S \subset V$ *such that there is no edge in* X *which connects a node in* S *to one in* $V \setminus S$. *If* $e \in E$ *is the minimum-weight edge between* S *and* $V - S$, *then* $X \cup \{e\}$ *is also part of some MST.*

In Kruskal's alg.: when we pick an edge connecting two components C and C', it is the lightest edge in the cut $(C, V - S)$ and thus the final tree is optimal, MST.

Implementation of Kruskal's algorithm

Avoiding cycles: every time we want to add edge (u, v) to MST we need to check whether its endpoints u and v are in different connected components.

Implementation of Kruskal's algorithm

Avoiding cycles: every time we want to add edge (u, v) to MST we need to check whether its endpoints u and v are in different connected components.
How to do this efficiently?

Implementation of Kruskal's algorithm

Avoiding cycles: every time we want to add edge (u, v) to MST we need to check whether its endpoints u and v are in different connected components.
How to do this efficiently?
We will maintain a collection of (disjoint) sets, each of them containing nodes of a particular connected component.

Implementation of Kruskal's algorithm

Avoiding cycles: every time we want to add edge (u, v) to MST we need to check whether its endpoints u and v are in different connected components.
How to do this efficiently?
We will maintain a collection of (disjoint) sets, each of them containing nodes of a particular connected component.
Set operations:

- MAKESET(x): creates set $\{x\}$
- FIND(x): to which set does x belong?
- UNION(x, y): merge sets containing x and y.

Implementation of Kruskal's algorithm

Avoiding cycles: every time we want to add edge (u, v) to MST we need to check whether its endpoints u and v are in different connected components.
How to do this efficiently?
We will maintain a collection of (disjoint) sets, each of them containing nodes of a particular connected component.
Set operations:

- MAKESET(x): creates set $\{x\}$
- FIND(x): to which set does x belong?
- UNION(x, y): merge sets containing x and y.

Kruskal's Algorithm

KRUSKAL(G, w)
for all $v \in V$ MAKESET(v)
$X := \emptyset$
Sort the edges of E by weight
for each edge (u, v) in nondecreasing order of weight
do if FIND(u) $\neq$FIND(v) then

- $X := X \cup \{(u, v)\}$
- UNION(u, v)

Kruskal's Algorithm

KRUSKAL(G, w)
for all $v \in V$ MAKESET(v)
$X := \emptyset$
Sort the edges of E by weight
for each edge (u, v) in nondecreasing order of weight
do if FIND$(u) \neq$FIND(v) then

- $X := X \cup \{(u, v)\}$
- UNION(u, v)

Running time :

Kruskal's Algorithm

KRUSKAL(G, w)

for all $v \in V$ MAKESET(v) $n\times$ MakeSet

$X := \emptyset$

Sort the edges of E by weight $O(m \log m)$

for each edge (u, v) in nondecreasing order of weight

do if FIND$(u) \neq$ FIND(v) then $2m\times$ Find

- $X := X \cup \{(u, v)\}$
- UNION(u, v) $(n-1)\times$Union

Running time :

Kruskal's Algorithm

KRUSKAL(G, w)
for all $v \in V$ MAKESET(v) $n\times$ MakeSet
$X := \emptyset$
Sort the edges of E by weight $O(m \log m)$
for each edge (u, v) in nondecreasing order of weight
do if FIND$(u) \neq$FIND(v) then $2m\times$ Find

- $X := X \cup \{(u, v)\}$
- UNION(u, v) $(n-1)\times$Union

Running time :
Total: $O(m \log n)$

Kruskal's Algorithm

KRUSKAL(G, w)

for all $v \in V$ MAKESET(v) — $n\times$ MakeSet

$X := \emptyset$

Sort the edges of E by weight — $O(m \log m)$

for each edge (u, v) in nondecreasing order of weight

do if FIND$(u) \neq$FIND(v) then — $2m\times$ Find

- $X := X \cup \{(u, v)\}$
- UNION(u, v) — $(n-1)\times$Union

Running time :

Total: $O(m \log n)$

b/c all disjoint-set operations take $O(m \log^* n)$ *time, where* $\log^* n$ *is much smaller than* $\log n$.

Prim's greedy choice

builds an MST by growing a single connected component complying with the cut property, r-initial node

Prim's greedy choice

PRIM(G, w, r)
for all $v \in V$ do $cost(v) := \infty$; $prev(v) := nil$
$cost(r) := 0$
$Q := BuilHeap(V)$ {keys = cost values}
while $Q \neq \emptyset$ do

- $u :=$EXTRACT_MIN(Q)
- for each $v \in Adj[u]$ do
  if $v \in Q$ and $cost(v) > w(u, v)$ then
  $cost(v) := w(u, v)$
  $prev(v) := u$

Prim's greedy choice

PRIM(G, w, r)
for all $v \in V$ do $cost(v) := \infty$; $prev(v) := nil$
$cost(r) := 0$
$Q := BuilHeap(V)$ {keys = cost values}
while $Q \neq \emptyset$ do

- $u :=$EXTRACT_MIN(Q)
- for each $v \in Adj[u]$ do
  if $v \in Q$ and $cost(v) > w(u, v)$ then
  $cost(v) := w(u, v)$
  $prev(v) := u$

$\{v, prev(v)\}$ form MST

Prim's greedy choice

PRIM(G, w, r)
for all $v \in V$ do $cost(v) := \infty$; $prev(v) := nil$
$cost(r) := 0$
$Q := BuilHeap(V)$ {keys = cost values}
while $Q \neq \emptyset$ do

- $u :=$EXTRACT_MIN(Q)
- for each $v \in Adj[u]$ do
  if $v \in Q$ and $cost(v) > w(u, v)$ then
  $cost(v) := w(u, v)$
  $prev(v) := u$

$\{v, prev(v)\}$ form MST *See example on the board.*

Analysis of Prim's algorithm

Almost the same as Dijkstra's algorithm!

Analysis of Prim's algorithm

Almost the same as Dijkstra's algorithm!
Only the key values in Q are different: here only the cost of joining the current tree.

Analysis of Prim's algorithm

Almost the same as Dijkstra's algorithm!
Only the key values in Q are different: here only the cost of joining the current tree.
Running time depends on the implementation of the priority queue.

Analysis of Prim's algorithm

Almost the same as Dijkstra's algorithm!
Only the key values in Q are different: here only the cost of joining the current tree.
Running time depends on the implementation of the priority queue.

Q implementation	Extract_MIN	Insert	Total
array	$O(n)$	$O(1)$	$O(n^2)$
binary heap	$O(\log n)$	$O(\log n)$	$O(m \log n)$
Fibonacci heap	$O(\log n)$	$O(1)$	$O(n \log n + m)$

Analysis of Prim's algorithm

Almost the same as Dijkstra's algorithm!
Only the key values in Q are different: here only the cost of joining the current tree.
Running time depends on the implementation of the priority queue.

Q implementation	Extract_MIN	Insert	Total
array	$O(n)$	$O(1)$	$O(n^2)$
binary heap	$O(\log n)$	$O(\log n)$	$O(m \log n)$
Fibonacci heap	$O(\log n)$	$O(1)$	$O(n \log n + m)$

Correctness follows from the **Cut Property**.

Disjoint-sets data structure

Set operations:

- MAKESET(x): creates set $\{x\}$
- FIND(x): to which set does x belong?
- UNION(x, y): merge sets containing x and y.

Disjoint-sets data structure

- An array:
  `Component[s]:=the name of the set containing` s
  Cost:

  MakeSet(x): $O(1)$
  Find(x): $O(1)$
  Union(x, y): $O(n)$

Disjoint-sets data structure

- An array:
  `Component[s]:=the name of the set containing` s
  Cost:

  MakeSet(x): $O(1)$
  Find(x): $O(1)$
  Union(x, y): $O(n)$

- A directed tree (disjoint-sets forest):
  Represent every set by a directed tree, nodes in an arbitrary order, with root as a *representative*

Disjoint-sets data structure

- An array:
  `Component[s]:=the name of the set containing` s
  Cost:

  MakeSet(x): $O(1)$
  Find(x): $O(1)$
  Union(x, y): $O(n)$

- A directed tree (disjoint-sets forest):
  Represent every set by a directed tree, nodes in an arbitrary order, with root as a *representative*

Example:

Disjoint-sets data structure

- A directed tree:
  for all $x:\ \pi(x)$-parent pointer, $rank(x)$-the height of the subtree rooted at x

  MAKESET(x):

  $\pi(x) := x$

  $rank(x) := 0$

  FIND(x):

  while $x \neq \pi(x)$ do $x = \pi(x)$

  return x

  UNION(x, y): ??

Disjoint-sets data structure

- A directed tree:
  for all x : $\pi(x)$-parent pointer, $rank(x)$-the height of the subtree rooted at x

  MAKESET(x):

  $\pi(x) := x$

  $rank(x) := 0$ $\quad O(1)$

  FIND(x):

  while $x \neq \pi(x)$ do $x = \pi(x)$

  return x $\quad O(height(Tree))$

  UNION(x, y): ??