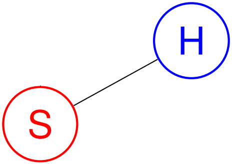
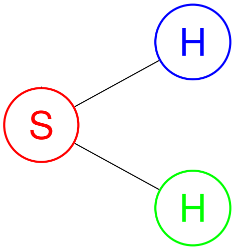
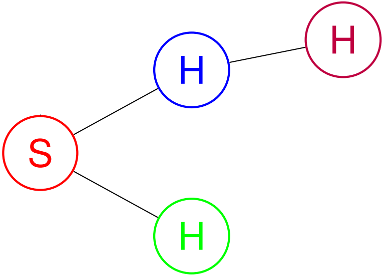
CS 170: Algorithms

CS 170: Algorithms

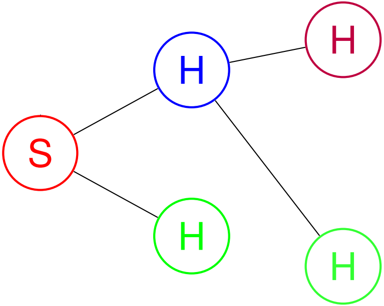

H
S

CS 170: Algorithms

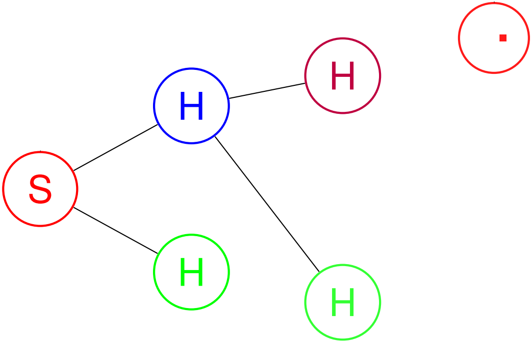



H
H
S
H

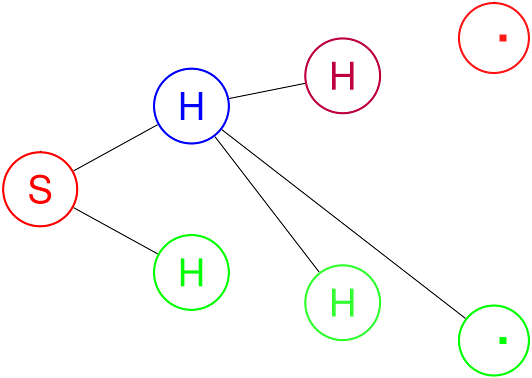
CS 170: Algorithms



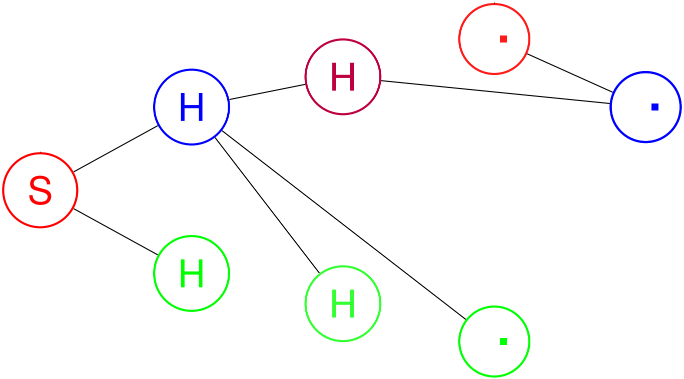
CS 170: Algorithms



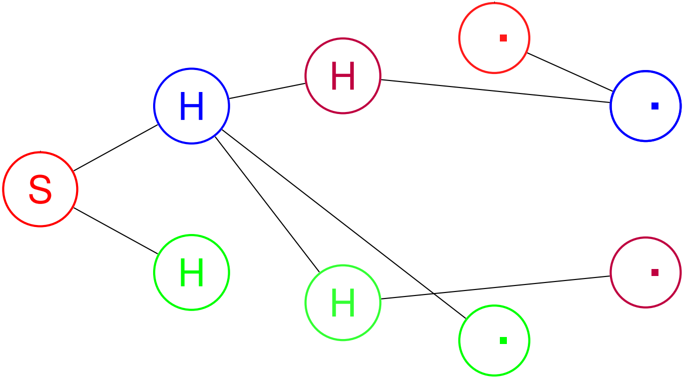
CS 170: Algorithms



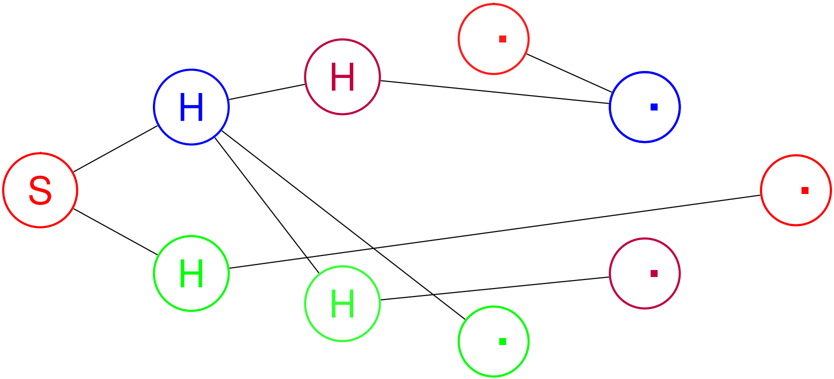
CS 170: Algorithms



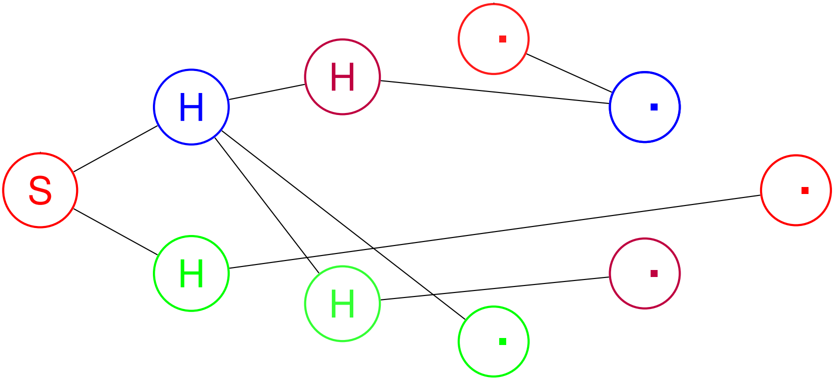
CS 170: Algorithms



CS 170: Algorithms



CS 170: Algorithms



Minimum spanning tree.

Lecture in a minute.

Tree Definitions:

$n-1$ edges and connected.

$n-1$ edges and no cycles.

All pairs of vertices connected by unique path.

Lecture in a minute.

Tree Definitions:

$n-1$ edges and connected.

$n-1$ edges and no cycles.

All pairs of vertices connected by unique path.

Minimum Spanning Tree: $G=(V,E)$, weights $w:E$

Kruskal: Sort edges.

Add edges in this order if no cycle.

Cut property:

Exists MST with minimum weight edge across cut.

Lecture in a minute.

Tree Definitions:

$n-1$ edges and connected.

$n-1$ edges and no cycles.

All pairs of vertices connected by unique path.

Minimum Spanning Tree: $G=(V,E)$, weights $w:E$

Kruskal: Sort edges.

Add edges in this order if no cycle.

Cut property:

Exists MST with minimum weight edge across cut.

Union-Find Data Structure.

Pointer implementation: $\pi(u)$.

makeset(s) – $\pi(u)=u$.

find(x) – returns root of pointer structure.

Lecture in a minute.

Tree Definitions:

$n-1$ edges and connected.

$n-1$ edges and no cycles.

All pairs of vertices connected by unique path.

Minimum Spanning Tree: $G=(V,E)$, weights $w:E$

Kruskal: Sort edges.

Add edges in this order if no cycle.

Cut property:

Exists MST with minimum weight edge across cut.

Union-Find Data Structure.

Pointer implementation: $\pi(u)$.

makeset(s) – $\pi(u)=u$.

find(x) – returns root of pointer structure.

union(x,y) – $\pi(find(x))=\pi(find(y))$.

Lecture in a minute.

Tree Definitions:

$n-1$ edges and connected.
$n-1$ edges and no cycles.
All pairs of vertices connected by unique path.

Minimum Spanning Tree: $G=(V,E)$, weights $w:E$

Kruskal: Sort edges.
Add edges in this order if no cycle.

Cut property:
Exists MST with minimum weight edge across cut.

Union-Find Data Structure.

Pointer implementation: $\pi(u)$.
makeset(s) – $\pi(u)=u$.
find(x) – returns root of pointer structure.
union(x,y) – $\pi(find(x))=\pi(find(y))$.

Lecture in a minute.

Tree Definitions:

$n-1$ edges and connected.

$n-1$ edges and no cycles.

All pairs of vertices connected by unique path.

Minimum Spanning Tree: $G=(V,E)$, weights $w:E$

Kruskal: Sort edges.

Add edges in this order if no cycle.

Cut property:

Exists MST with minimum weight edge across cut.

Union-Find Data Structure.

Pointer implementation: $\pi(u)$.

makeset(s) – $\pi(u)=u$.

find(x) – returns root of pointer structure.

union(x,y) – $\pi(find(x))=\pi(find(y))$.

Union by rank: $O(\log n)$ depth for pointer structure.

Lecture in a minute.

Tree Definitions:

$n-1$ edges and connected.

$n-1$ edges and no cycles.

All pairs of vertices connected by unique path.

Minimum Spanning Tree: $G = (V, E)$, weights $w : E$

Kruskal: Sort edges.

Add edges in this order if no cycle.

Cut property:

Exists MST with minimum weight edge across cut.

Union-Find Data Structure.

Pointer implementation: $\pi(u)$.

makeset(s) – $\pi(u) = u$.

find(x) – returns root of pointer structure.

union(x,y) – $\pi(find(x)) = \pi(find(y))$.

Union by rank: $O(\log n)$ depth for pointer structure.

union(x,y) - point to larger rank root.

Lecture in a minute.

Tree Definitions:

$n-1$ edges and connected.

$n-1$ edges and no cycles.

All pairs of vertices connected by unique path.

Minimum Spanning Tree: $G=(V,E)$, weights $w:E$

Kruskal: Sort edges.

Add edges in this order if no cycle.

Cut property:

Exists MST with minimum weight edge across cut.

Union-Find Data Structure.

Pointer implementation: $\pi(u)$.

makeset(s) – $\pi(u)=u$.

find(x) – returns root of pointer structure.

union(x,y) – $\pi(find(x))=\pi(find(y))$.

Union by rank: $O(\log n)$ depth for pointer structure.

union(x,y) - point to larger rank root.

increase rank if tied.

Lecture in a minute.

Tree Definitions:

$n-1$ edges and connected.

$n-1$ edges and no cycles.

All pairs of vertices connected by unique path.

Minimum Spanning Tree: $G=(V,E)$, weights $w:E$

Kruskal: Sort edges.

Add edges in this order if no cycle.

Cut property:

Exists MST with minimum weight edge across cut.

Union-Find Data Structure.

Pointer implementation: $\pi(u)$.

makeset(s) – $\pi(u)=u$.

find(x) – returns root of pointer structure.

union(x,y) – $\pi(find(x))=\pi(find(y))$.

Union by rank: $O(\log n)$ depth for pointer structure.

union(x,y) - point to larger rank root.

increase rank if tied.

$>2^k$ nodes in rank k root tree.

$O(\log n)$ depth structure.

Trees.

Def: A tree is a connected graph with no cycles.

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.

D

B C

A E

Adding any edge between components

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.

D

B C

A E

Adding any edge between components

$\implies$ reduces number of components by one.

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.

D

B C

A E

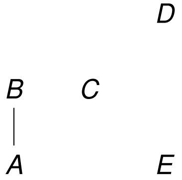
Adding any edge between components
$\implies$ reduces number of components by one.

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.



Adding any edge between components

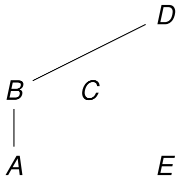
$\implies$ reduces number of components by one.

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.



Adding any edge between components

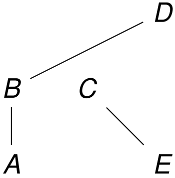
$\implies$ reduces number of components by one.

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.



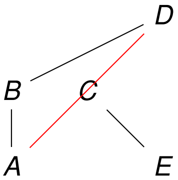
Adding any edge between components
$\implies$ reduces number of components by one.

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.



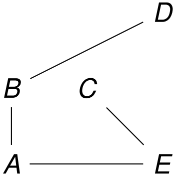
Adding any edge between components
$\implies$ reduces number of components by one.
After $n-1$ additions

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.



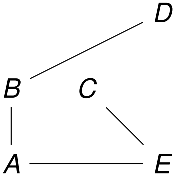
Adding any edge between components
$\Longrightarrow$ reduces number of components by one.
After $n-1$ additions one component!

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.



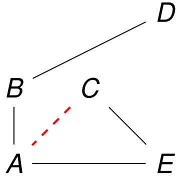
Adding any edge between components
$\implies$ reduces number of components by one.
After $n-1$ additions one component!
(If more additions,

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.



Adding any edge between components

$\implies$ reduces number of components by one.

After $n-1$ additions one component!

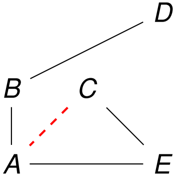
(If more additions, inside component

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.



Adding any edge between components

$\implies$ reduces number of components by one.

After $n-1$ additions one component!

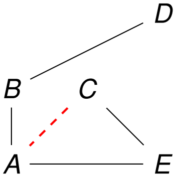
(If more additions, inside component $\implies$ cycle!)

Trees.

Def: A tree is a connected graph with no cycles.

Property: A tree has $n-1$ edges.

Start with empty graph with n components.



Adding any edge between components
$\implies$ reduces number of components by one.
After $n-1$ additions one component!
(If more additions, inside component $\implies$ cycle!) $\square$

Equivalent Definition.

Def: A tree is a connected graph with no cycles.

Equivalent Definition.

Def: A tree is a connected graph with no cycles.

Property: A connected graph with $n-1$ edges is a tree.

Equivalent Definition.

Def: A tree is a connected graph with no cycles.

Property: A connected graph with $n-1$ edges is a tree.

If not,

Equivalent Definition.

Def: A tree is a connected graph with no cycles.

Property: A connected graph with $n-1$ edges is a tree.

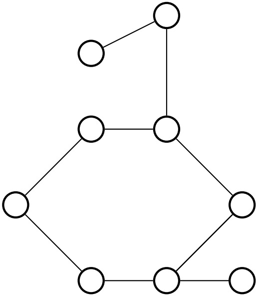
If not, there is $n-1$ edge connected graph with a cycle.

Equivalent Definition.

Def: A tree is a connected graph with no cycles.

Property: A connected graph with $n-1$ edges is a tree.

If not, there is $n-1$ edge connected graph with a cycle.

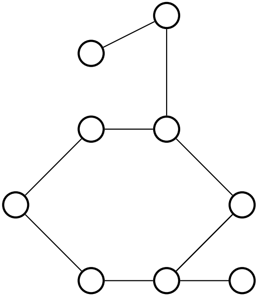


Equivalent Definition.

Def: A tree is a connected graph with no cycles.

Property: A connected graph with $n-1$ edges is a tree.

If not, there is $n-1$ edge connected graph with a cycle.



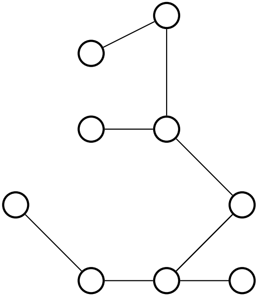
Remove edge on cycle, still connected.

Equivalent Definition.

Def: A tree is a connected graph with no cycles.

Property: A connected graph with $n-1$ edges is a tree.

If not, there is $n-1$ edge connected graph with a cycle.



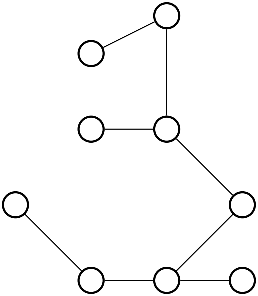
Remove edge on cycle, still connected.   And n-2 edges.

Equivalent Definition.

Def: A tree is a connected graph with no cycles.

Property: A connected graph with $n-1$ edges is a tree.

If not, there is $n-1$ edge connected graph with a cycle.



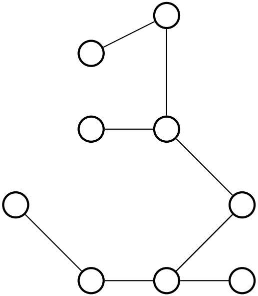
Remove edge on cycle, still connected. And n-2 edges.
Must have at least $n-1$ edges to be connected.

Equivalent Definition.

Def: A tree is a connected graph with no cycles.

Property: A connected graph with $n-1$ edges is a tree.

If not, there is $n-1$ edge connected graph with a cycle.



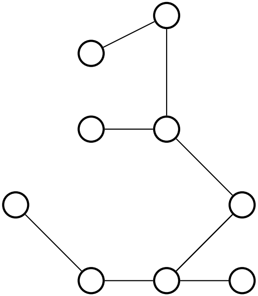
Remove edge on cycle, still connected. And n-2 edges.
Must have at least $n-1$ edges to be connected.
Doh!

Equivalent Definition.

Def: A tree is a connected graph with no cycles.

Property: A connected graph with $n-1$ edges is a tree.

If not, there is $n-1$ edge connected graph with a cycle.



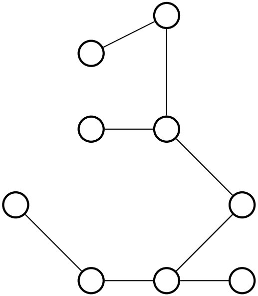
Remove edge on cycle, still connected.   And n-2 edges.
Must have at least $n-1$ edges to be connected.
Doh! $\rightarrow$ no cycle.

Equivalent Definition.

Def: A tree is a connected graph with no cycles.

Property: A connected graph with $n-1$ edges is a tree.

If not, there is $n-1$ edge connected graph with a cycle.



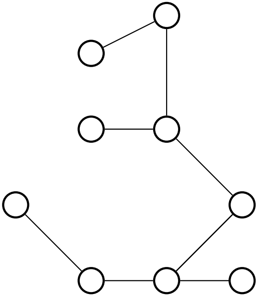
Remove edge on cycle, still connected. And n-2 edges.
Must have at least $n-1$ edges to be connected.
Doh! $\rightarrow$ no cycle. $\square$

Equivalent Definition.

Def: A tree is a connected graph with no cycles.

Property: A connected graph with $n-1$ edges is a tree.

If not, there is $n-1$ edge connected graph with a cycle.



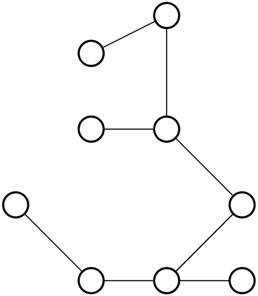
Remove edge on cycle, still connected. And n-2 edges.
Must have at least $n-1$ edges to be connected.
Doh! $\rightarrow$ no cycle. $\square$

Equivalent Definition.

Def: A tree is a connected graph with no cycles.

Property: A connected graph with $n-1$ edges is a tree.

If not, there is $n-1$ edge connected graph with a cycle.



Remove edge on cycle, still connected. And n-2 edges.
Must have at least $n-1$ edges to be connected.
Doh! $\rightarrow$ no cycle. $\square$

Another Equivalent Defition

Def: A tree is a connected graph with no cycles.

Another Equivalent Defition

Def: A tree is a connected graph with no cycles.

Property: A graph is a tree if and only if it has a unique path between every pair of nodes.

Another Equivalent Defition

Def: A tree is a connected graph with no cycles.

Property: A graph is a tree if and only if it has a unique path between every pair of nodes.

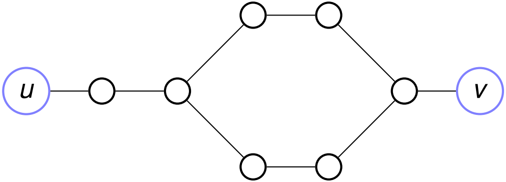
If two paths:

Another Equivalent Defition

Def: A tree is a connected graph with no cycles.

Property: A graph is a tree if and only if it has a unique path between every pair of nodes.

If two paths:

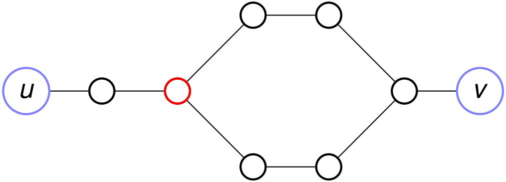


Another Equivalent Defition

Def: A tree is a connected graph with no cycles.

Property: A graph is a tree if and only if it has a unique path between every pair of nodes.

If two paths:



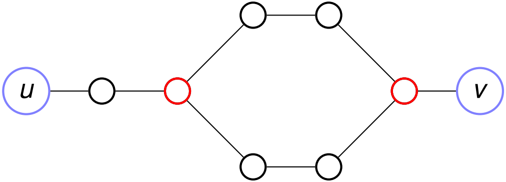
Diverge

Another Equivalent Defition

Def: A tree is a connected graph with no cycles.

Property: A graph is a tree if and only if it has a unique path between every pair of nodes.

If two paths:



Diverge

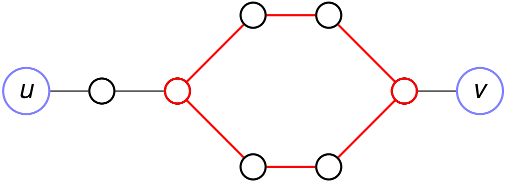
Come back together.

Another Equivalent Defition

Def: A tree is a connected graph with no cycles.

Property: A graph is a tree if and only if it has a unique path between every pair of nodes.

If two paths:



Diverge

Come back together.

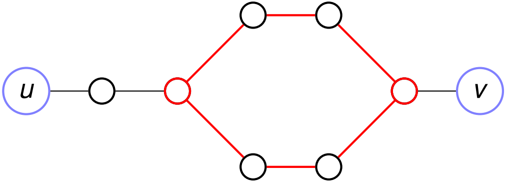
$\Longrightarrow$ cycle!

Another Equivalent Defition

Def: A tree is a connected graph with no cycles.

Property: A graph is a tree if and only if it has a unique path between every pair of nodes.

If two paths:



Diverge

Come back together.

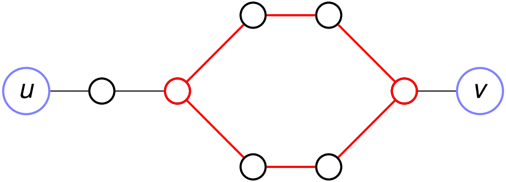
$\Longrightarrow$ cycle! Not Tree!

Another Equivalent Defition

Def: A tree is a connected graph with no cycles.

Property: A graph is a tree if and only if it has a unique path between every pair of nodes.

If two paths:



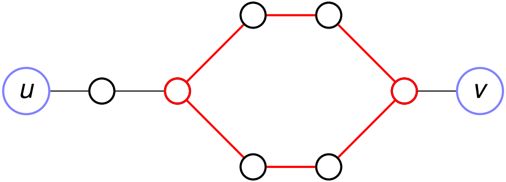
Diverge
  Come back together.
   $\Longrightarrow$ cycle! Not Tree!
If yes, connected and no cycle.

Another Equivalent Defition

Def: A tree is a connected graph with no cycles.

Property: A graph is a tree if and only if it has a unique path between every pair of nodes.

If two paths:



Diverge

Come back together.

$\Longrightarrow$ cycle! Not Tree!

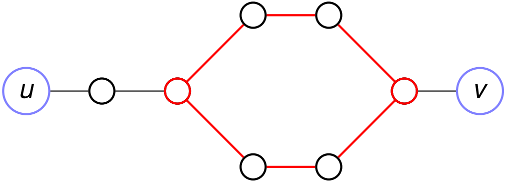
If yes, connected and no cycle. Tree!

Another Equivalent Defition

Def: A tree is a connected graph with no cycles.

Property: A graph is a tree if and only if it has a unique path between every pair of nodes.

If two paths:



Diverge

Come back together.

$\Longrightarrow$ cycle! Not Tree!

If yes, connected and no cycle. Tree! $\square$

Minimum Spanning Tree.

Given a graph, $G = (V, E)$, edge weights w_e, find the cheapest possible connected subgraph.

Minimum Spanning Tree.

Given a graph, $G = (V, E)$, edge weights w_e, find the cheapest possible connected subgraph.

Will it be a tree?

Minimum Spanning Tree.

Given a graph, $G = (V, E)$, edge weights w_e, find the cheapest possible connected subgraph.

Will it be a tree?

Yes?

Minimum Spanning Tree.

Given a graph, $G = (V, E)$, edge weights w_e, find the cheapest possible connected subgraph.

Will it be a tree?

Yes? No?

Minimum Spanning Tree.

Given a graph, $G = (V, E)$, edge weights w_e, find the cheapest possible connected subgraph.

Will it be a tree?

Yes? No?

Yes.

Minimum Spanning Tree.

Given a graph, $G = (V, E)$, edge weights w_e, find the cheapest possible connected subgraph.

Will it be a tree?

Yes? No?

Yes. If edge weights positive.

Minimum Spanning Tree.

Given a graph, $G = (V, E)$, edge weights w_e, find the cheapest possible connected subgraph.

Will it be a tree?

Yes? No?

Yes. If edge weights positive.

If negative edges, then restrict to tree.

Minimum Spanning Tree.

Given a graph, $G=(V,E)$, edge weights w_e, find the cheapest possible connected subgraph.

Will it be a tree?

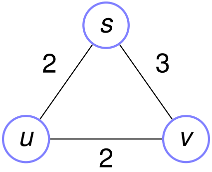
Yes? No?

Yes. If edge weights positive.

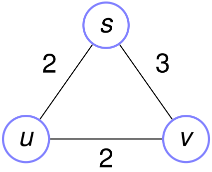
If negative edges, then restrict to tree.

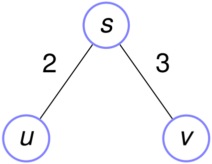
Given a graph, $G=(V,E)$, edge weights w_e, find the lowest weight spanning tree.

To MST or not!

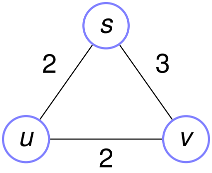


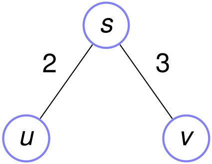
To MST or not!





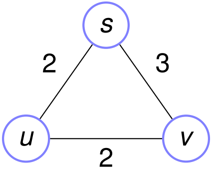
To MST or not!

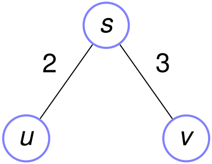




MST?

To MST or not!

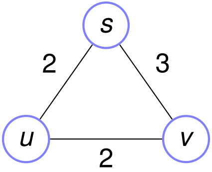


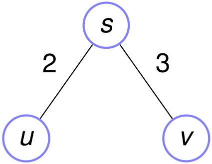


MST?

Shortest Path Tree from *s*!

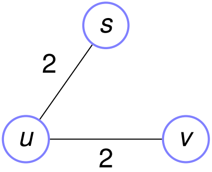
To MST or not!





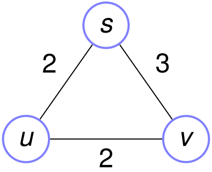
MST?

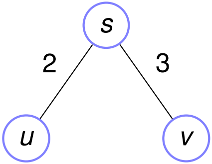
Shortest Path Tree from *s*!



MST?

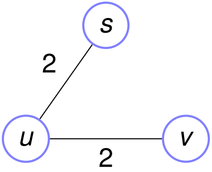
To MST or not!





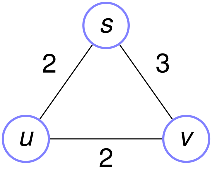
MST?

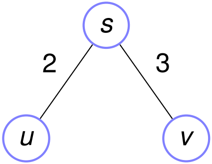
Shortest Path Tree from *s*!



MST? Yes!

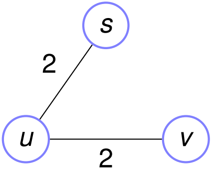
To MST or not!





MST?

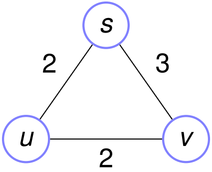
Shortest Path Tree from *s*!

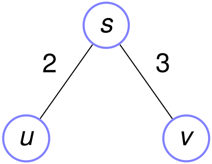


MST? Yes!

Shortest path from *s* to *v* in tree?

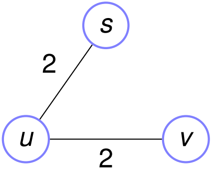
To MST or not!





MST?

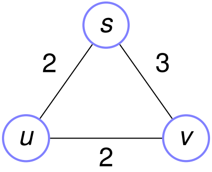
Shortest Path Tree from *s*!



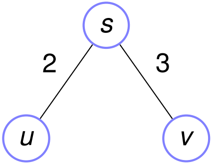
MST? Yes!

Shortest path from *s* to *v* in tree? No!

To MST or not!

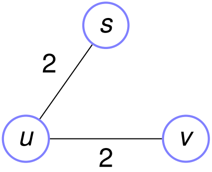


MST



MST?

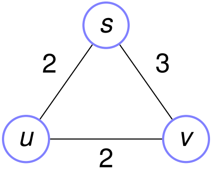
Shortest Path Tree from *s*!

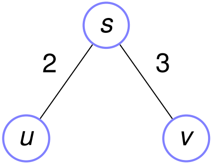


MST? Yes!

Shortest path from *s* to *v* in tree? No!

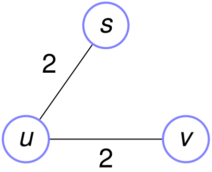
To MST or not!





MST?

Shortest Path Tree from *s*!

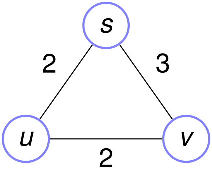


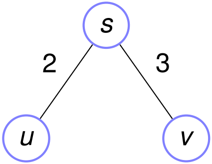
MST? Yes!

Shortest path from *s* to *v* in tree? No!

MST - cheapest spanning tree of graph.

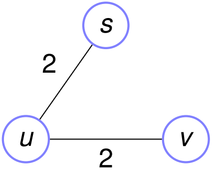
To MST or not!





MST?

Shortest Path Tree from *s*!



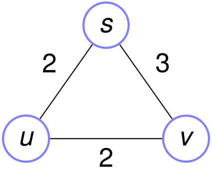
MST? Yes!

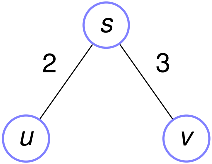
Shortest path from *s* to *v* in tree? No!

MST - cheapest spanning tree of graph.

Shortest path tree

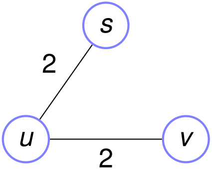
To MST or not!





MST?

Shortest Path Tree from *s*!



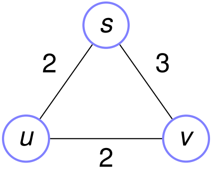
MST? Yes!

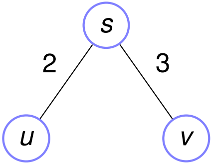
Shortest path from *s* to *v* in tree? No!

MST - cheapest spanning tree of graph.

Shortest path tree
- contains shortest paths from *s* to other nodes.

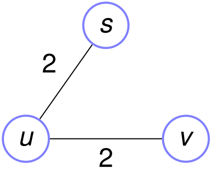
To MST or not!





MST?

Shortest Path Tree from *s*!



MST? Yes!

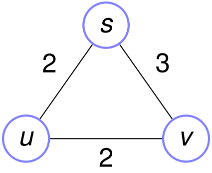
Shortest path from *s* to *v* in tree? No!

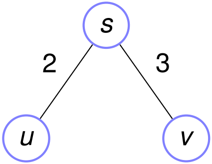
MST - cheapest spanning tree of graph.

Shortest path tree
- contains shortest paths from *s* to other nodes.

MST -

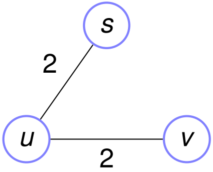
To MST or not!





MST?

Shortest Path Tree from *s*!



MST? Yes!

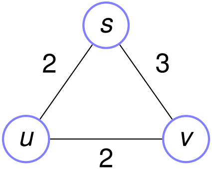
Shortest path from *s* to *v* in tree? No!

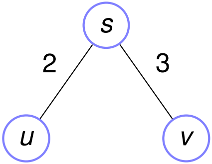
MST - cheapest spanning tree of graph.

Shortest path tree
- contains shortest paths from *s* to other nodes.

MST -
do not care about shortest paths!

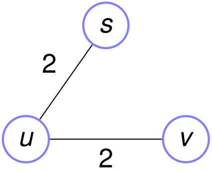
To MST or not!





MST?

Shortest Path Tree from *s*!



MST? Yes!

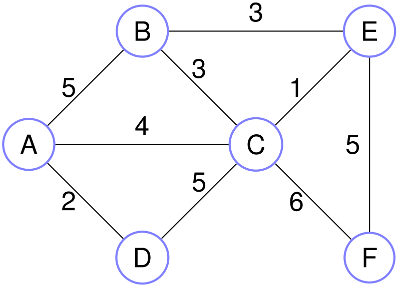
Shortest path from *s* to *v* in tree? No!

MST - cheapest spanning tree of graph.

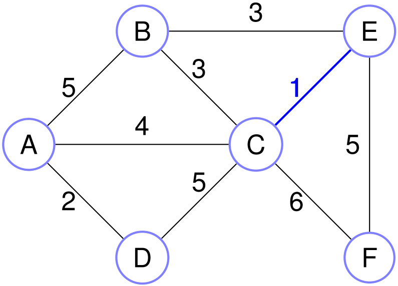
Shortest path tree
- contains shortest paths from *s* to other nodes.

MST -
do not care about shortest paths!
just lowest weight tree.

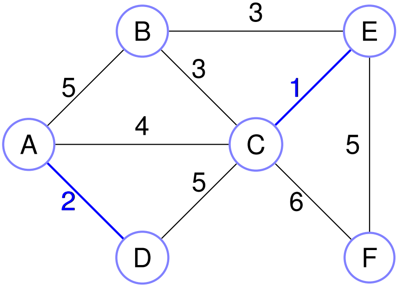
Example and Algorithm



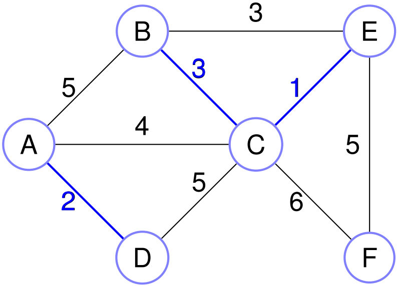
Example and Algorithm



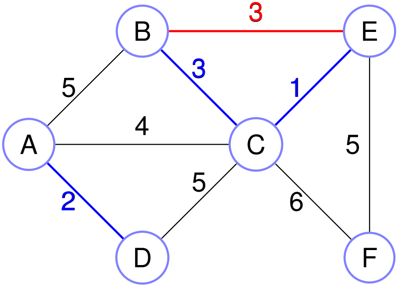
Example and Algorithm



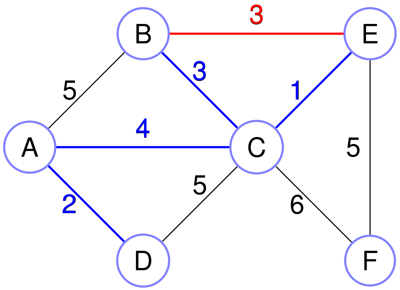
Example and Algorithm



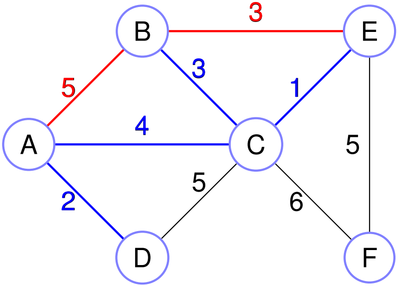
Example and Algorithm



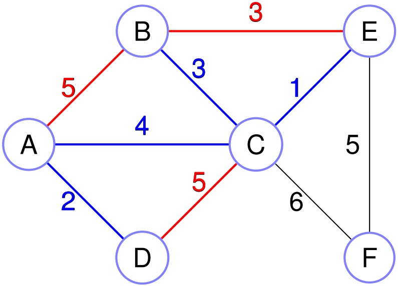
Example and Algorithm



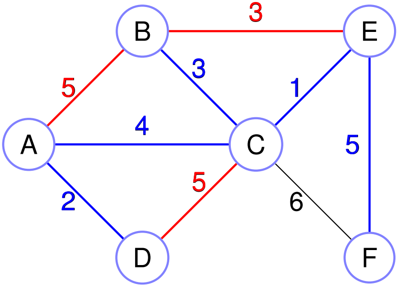
Example and Algorithm



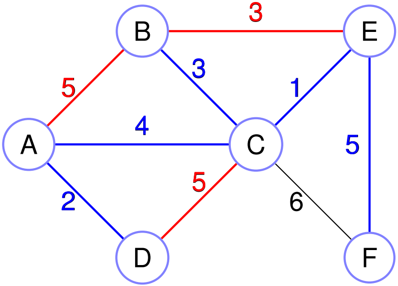
Example and Algorithm



Example and Algorithm



Example and Algorithm



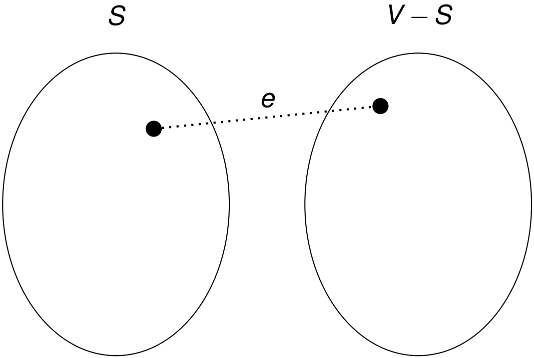
MST: total cost is 2+4+3 +1 + 5 = 15.

Cut property.

Smallest edge across any cut is in some MST.

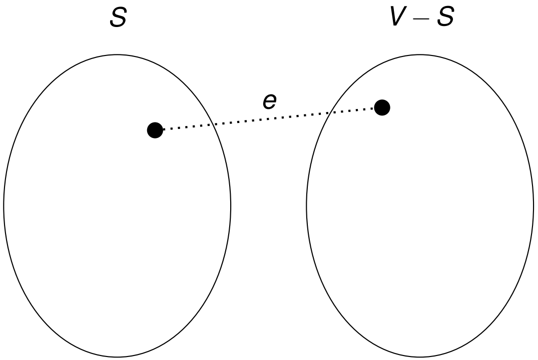
Cut property.

Smallest edge across any cut is in some MST.



Cut property.

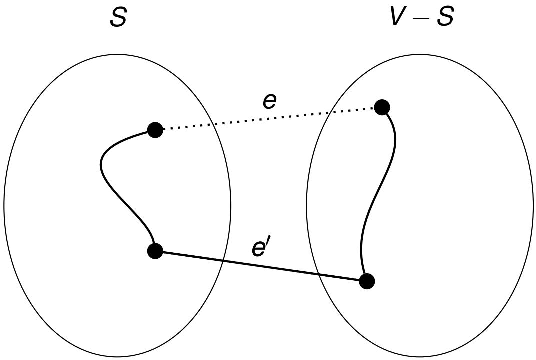
Smallest edge across any cut is in some MST.



Tree Connected $\implies$

Cut property.

Smallest edge across any cut is in some MST.

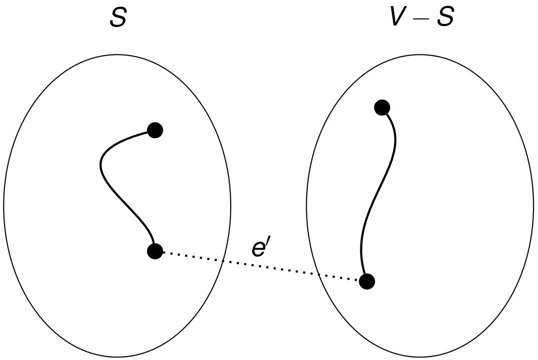


Tree Connected $\implies$

there exists e' across cut! Replace e' with e.

Cut property.

Smallest edge across any cut is in some MST.



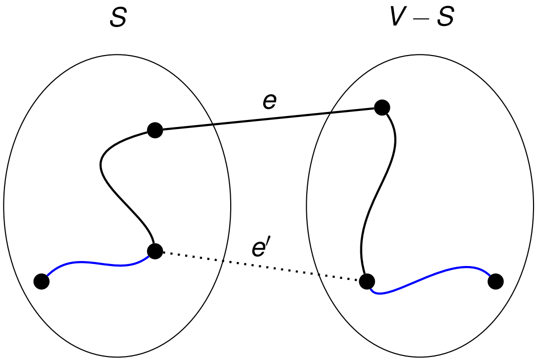
Tree Connected $\implies$

there exists e' across cut! Replace e' with e.

Every pair remains connected.

Cut property.

Smallest edge across any cut is in some MST.



Tree Connected $\implies$

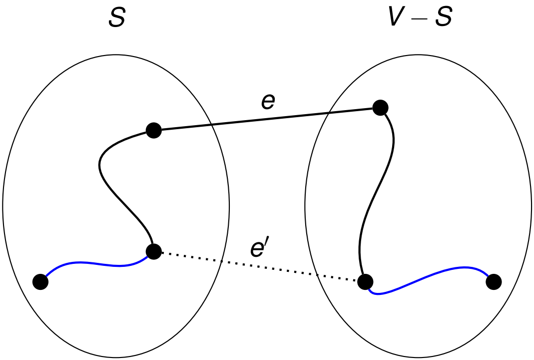
there exists e' across cut! Replace e' with e.

Every pair remains connected.

If used e' can use path through e.

Cut property.

Smallest edge across any cut is in some MST.



Tree Connected $\implies$

there exists e' across cut! Replace e' with e.

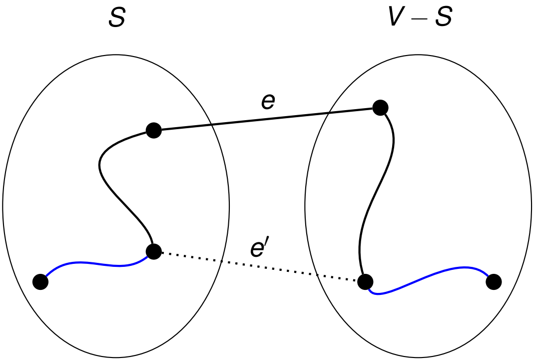
Every pair remains connected.

If used e' can use path through e.

and $n-1$ edges.

Cut property.

Smallest edge across any cut is in some MST.



Tree Connected $\implies$

there exists e' across cut! Replace e' with e.

Every pair remains connected.

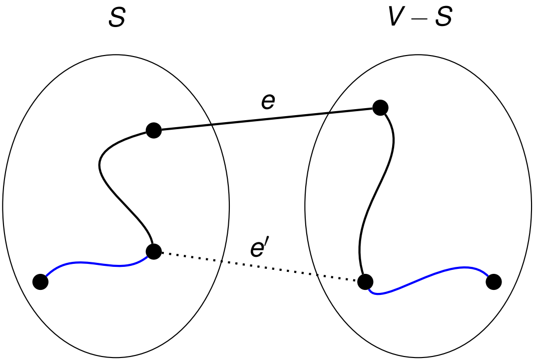
If used e' can use path through e.

and $n-1$ edges.

So still a tree

Cut property.

Smallest edge across any cut is in some MST.



Tree Connected $\implies$

there exists e' across cut! Replace e' with e.

Every pair remains connected.

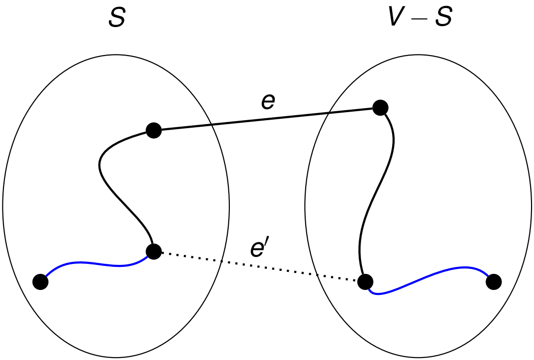
If used e' can use path through e.

and $n-1$ edges.

So still a tree and is no more costly ($w(e) \leq w(e')$.)

Cut property.

Smallest edge across any cut is in some MST.



Tree Connected $\implies$

there exists e' across cut! Replace e' with e.

Every pair remains connected.

If used e' can use path through e.

and $n-1$ edges.

So still a tree and is no more costly ($w(e) \leq w(e')$.)

$\square$

Kruskal

Sort edges.
$F =$. For each edge: e
    If no cycle, $F = F + e$.

Kruskal

Sort edges.
$F =$. For each edge: e
    If no cycle, $F = F + e$.

How to check for cycle for edge (u, v)in F?

Kruskal

Sort edges.
$F =$. For each edge: e
    If no cycle, $F = F + e$.

How to check for cycle for edge (u, v)in F?

Check for path between u and v in F.

Kruskal

Sort edges.
$F =$. For each edge: e
    If no cycle, $F = F + e$.

How to check for cycle for edge (u, v)in F?

Check for path between u and v in F.

Total Running time?

Kruskal

Sort edges.
$F =$. For each edge: e
    If no cycle, $F = F + e$.

How to check for cycle for edge (u, v)in F?

Check for path between u and v in F.

Total Running time?

$O(n)$ time

Kruskal

Sort edges.
$F =$. For each edge: e
    If no cycle, $F = F + e$.

How to check for cycle for edge (u, v)in F?

Check for path between u and v in F.

Total Running time?

$O(n)$ time $\rightarrow$ $O(nm)$ for Kruskals.

Kruskal

Sort edges.
$F =$. For each edge: $e = (u, v)$
    If no cycle in F, add edge.

Kruskal

Sort edges.
$F =$. For each edge: $e = (u, v)$
If no cycle in F, add edge.

Main issue: Check for cycle.

Kruskal

Sort edges.
$F =$. For each edge: $e = (u, v)$
    If no cycle in F, add edge.

Main issue: Check for cycle.

Maintain connected components.

Kruskal

Sort edges.
$F =$. For each edge: $e = (u, v)$
    If no cycle in F, add edge.

Main issue: Check for cycle.

Maintain connected components.

At beginning each node by itself.

Kruskal

Sort edges.
$F =$. For each edge: $e = (u, v)$
If no cycle in F, add edge.

Main issue: Check for cycle.

Maintain connected components.

At beginning each node by itself.
Adding edge, joins component.

Kruskal

Sort edges.
$F =$. For each edge: $e = (u, v)$
    If no cycle in F, add edge.

Main issue: Check for cycle.

Maintain connected components.

At beginning each node by itself.
Adding edge, joins component.
Edge (u, v) in cycle?

Kruskal

Sort edges.
$F =$. For each edge: $e = (u, v)$
  If no cycle in F, add edge.

Main issue: Check for cycle.

Maintain connected components.

At beginning each node by itself.
Adding edge, joins component.
Edge (u, v) in cycle? u and v in same component.

Kruskal

Sort edges.
$F =$. For each edge: $e = (u, v)$
    If no cycle in F, add edge.

Main issue: Check for cycle.

Maintain connected components.

At beginning each node by itself.
Adding edge, joins component.
Edge (u, v) in cycle? u and v in same component.

Disjoint Sets Data Structure.

Kruskal

Sort edges.
$F =$. For each edge: $e = (u, v)$
    If no cycle in F, add edge.

Main issue: Check for cycle.

Maintain connected components.

At beginning each node by itself.
Adding edge, joins component.
Edge (u, v) in cycle? u and v in same component.

Disjoint Sets Data Structure.

makeset(x) - makes singleton set $\{x\}$.

Kruskal

Sort edges.
$F =$. For each edge: $e = (u, v)$
If no cycle in F, add edge.

Main issue: Check for cycle.

Maintain connected components.

At beginning each node by itself.
Adding edge, joins component.
Edge (u, v) in cycle? u and v in same component.

Disjoint Sets Data Structure.

makeset(x) - makes singleton set $\{x\}$.
find(x) - finds set containing x.

Kruskal

Sort edges.
$F =$. For each edge: $e = (u,v)$
    If no cycle in F, add edge.

Main issue: Check for cycle.

Maintain connected components.

At beginning each node by itself.
Adding edge, joins component.
Edge (u,v) in cycle? u and v in same component.

Disjoint Sets Data Structure.

makeset(x) - makes singleton set $\{x\}$.
find(x) - finds set containing x.
union(x,y) - merge sets containing x and y.

Kruskal

Sort edges.
$F =$. For each edge: $e = (u, v)$
    If no cycle in F, add edge.

Main issue: Check for cycle.

Maintain connected components.

At beginning each node by itself.
Adding edge, joins component.
Edge (u, v) in cycle? u and v in same component.

Disjoint Sets Data Structure.

makeset(x) - makes singleton set $\{x\}$.
find(x) - finds set containing x.
union(x,y) - merge sets containing x and y.

“If no cycle”

Kruskal

Sort edges.
$F =$. For each edge: $e = (u, v)$
  If no cycle in F, add edge.

Main issue: Check for cycle.

Maintain connected components.

At beginning each node by itself.
Adding edge, joins component.
Edge (u, v) in cycle? u and v in same component.

Disjoint Sets Data Structure.

makeset(x) - makes singleton set $\{x\}$.
find(x) - finds set containing x.
union(x,y) - merge sets containing x and y.

"If no cycle" $\equiv$ "**find**(u) $\neq$ **find**(v)"

Kruskal

Sort edges.
$F =$. For each edge: $e = (u, v)$
    If no cycle in F, add edge.

Main issue: Check for cycle.

Maintain connected components.

At beginning each node by itself.
Adding edge, joins component.
Edge (u, v) in cycle? u and v in same component.

Disjoint Sets Data Structure.

makeset(x) - makes singleton set $\{x\}$.
find(x) - finds set containing x.
union(x,y) - merge sets containing x and y.

"If no cycle" $\equiv$ "**find**(u) $\neq$ **find**(v)"

"Add edge"

Kruskal

Sort edges.
$F =$. For each edge: $e = (u, v)$
    If no cycle in F, add edge.

Main issue: Check for cycle.

Maintain connected components.

At beginning each node by itself.
Adding edge, joins component.
Edge (u, v) in cycle? u and v in same component.

Disjoint Sets Data Structure.

makeset(x) - makes singleton set $\{x\}$.
find(x) - finds set containing x.
union(x,y) - merge sets containing x and y.

"If no cycle" $\equiv$ "**find**(u) $\neq$ **find**(v)"

"Add edge" $\equiv$ "**union(*u*,*v*)**"

Kruskal

Sort edges.
$F =$. For each edge: $e = (u, v)$
  If no cycle in F, add edge.

Main issue: Check for cycle.

Maintain connected components.

At beginning each node by itself.
Adding edge, joins component.
Edge (u, v) in cycle? u and v in same component.

Disjoint Sets Data Structure.

makeset(x) - makes singleton set $\{x\}$.
find(x) - finds set containing x.
union(x,y) - merge sets containing x and y.

"If no cycle" $\equiv$ "**find**(u) $\neq$ **find**(v)"

"Add edge" $\equiv$ "**union(**u**,**v**)**"

Disjoint Set Data Structure

Maintain pointers: $\pi(x)$ for each x.

Disjoint Set Data Structure

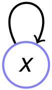
Maintain pointers: $\pi(x)$ for each x.

makeset(x)

Disjoint Set Data Structure

Maintain pointers: $\pi(x)$ for each x.

makeset(x) $\pi(x) = x$.



union(x,y)
$\pi(\text{find}(x)) = \text{find}(y)$

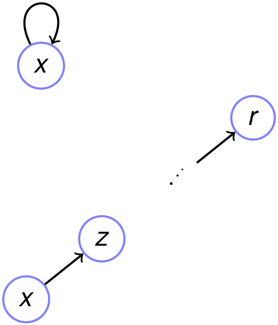
Disjoint Set Data Structure

Maintain pointers: $\pi(x)$ for each x.

makeset(x) $\pi(x) = x$.

union(x,y)
$\pi(\text{find}(x)) = \text{find}(y)$

find(x)
if $\pi(x) == x$
return x
else
find($\pi(x)$)



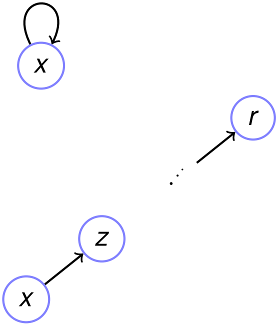
Disjoint Set Data Structure

Maintain pointers: $\pi(x)$ for each x.

makeset(x) $\pi(x) = x$.

union(x,y)
$\pi(\text{find}(x)) = \text{find}(y)$

find(x)
if $\pi(x) == x$
return x
else
find($\pi(x)$)



How long does find take?

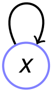
(A) $O(n)$

(B) $O(1)$

Disjoint Set Data Structure

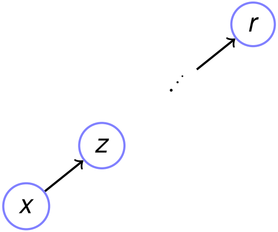
Maintain pointers: $\pi(x)$ for each x.

makeset(x) $\pi(x) = x$.



union(x,y)
$\pi(\text{find}(x)) = \text{find}(y)$

find(x)
if $\pi(x) == x$
return x
else
find($\pi(x)$)



How long does find take?

(A) $O(n)$

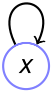
(B) $O(1)$

(C) Depends.

Disjoint Set Data Structure

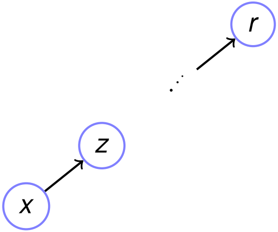
Maintain pointers: $\pi(x)$ for each x.

makeset(x) $\pi(x) = x$.



union(x,y)
$\pi(\text{find}(x)) = \text{find}(y)$

find(x)
if $\pi(x) == x$
return x
else
find($\pi(x)$)



How long does find take?

(A) $O(n)$

(B) $O(1)$

(C) Depends.

Want depth to be small!

Disjoint Set Data Structure

Maintain pointers: $\pi(x)$ for each x.

Disjoint Set Data Structure

Maintain pointers: $\pi(x)$ for each x.

makeset(x)

Disjoint Set Data Structure

Maintain pointers: $\pi(x)$ for each x.

makeset(x) $\pi(x) = x$.

Disjoint Set Data Structure

Maintain pointers: $\pi(x)$ for each x.

makeset(x) $\pi(x) = x$.

find(x)
  if $\pi(x) == x$
    return x
  else
    find($\pi(x)$)

Disjoint Set Data Structure

Maintain pointers: $\pi(x)$ for each x.

makeset(x) $\pi(x) = x$.

find(x)
    if $\pi(x) == x$
        return x
    else
        find($\pi(x)$)

Make a bit less deep: union-by-rank.

Disjoint Set Data Structure

Maintain pointers: $\pi(x)$ for each x.

makeset(x) $\pi(x) = x$.

find(x)
  if $\pi(x) == x$
    return x
  else
    find($\pi(x)$)

Make a bit less deep: union-by-rank.
union(x,y)

Disjoint Set Data Structure

Maintain pointers: $\pi(x)$ for each x.

makeset(x) $\pi(x) = x$.

find(x)
  if $\pi(x) == x$
    return x
  else
    find($\pi(x)$)

Make a bit less deep: union-by-rank.
union(x,y)
Use roots of x and y.

Disjoint Set Data Structure

Maintain pointers: $\pi(x)$ for each x.

makeset(x) $\pi(x) = x$.

find(x)
  if $\pi(x) == x$
    return x
  else
    find($\pi(x)$)

Make a bit less deep: union-by-rank.
union(x,y)
Use roots of x and y.
Which points to which?

Disjoint Set Data Structure

Maintain pointers: $\pi(x)$ for each x.

makeset(x) $\pi(x) = x$.

find(x)
  if $\pi(x) == x$
    return x
  else
    find($\pi(x)$)

Make a bit less deep: union-by-rank.
union(x,y)
Use roots of x and y.
Which points to which?
"smaller" to "larger"

Disjoint Set Data Structure

Maintain pointers: $\pi(x)$ for each x.

makeset(x) $\pi(x) = x$.

find(x)
    if $\pi(x) == x$
        return x
    else
        find($\pi(x)$)

Make a bit less deep: union-by-rank.
union(x,y)
Use roots of x and y.
Which points to which?
“smaller” to “larger” ..sort of.

Union by rank.

Initially: rank(x) $= 0$.

Union by rank.

Initially: rank(x) $= 0$.

union(x,y)

r_x = find(x)
r_y = find(y)

Union by rank.

Initially: rank(x) $= 0$.

union(x,y)

r_x = find(x)
r_y = find(y)
if rank(r_x) $<$ rank(r_y):

Union by rank.

Initially: rank(x) $= 0$.

union(x,y)

r_x = find(x)
r_y = find(y)
if rank(r_x) $<$ rank(r_y):
$\pi(r_x) = r_y$

Union by rank.

Initially: rank(x) $= 0$.

union(x,y)

r_x = find(x)
r_y = find(y)
if rank(r_x) $<$ rank(r_y):
$\pi(r_x) = r_y$
else:
$\pi(r_y) = r_x$

Union by rank.

Initially: rank(x) $= 0$.

union(x,y)

r_x = find(x)
r_y = find(y)
if rank(r_x) $<$ rank(r_y):
$\pi(r_x) = r_y$
else:
$\pi(r_y) = r_x$
if rank(r_x) == rank(r_y):
rank(r_x) $+= 1$

Why rank?

Lemma: Pop's got a higher rank:

Why rank?

Lemma: Pop's got a higher rank:
$\text{rank}(x) < \text{rank}(\pi(x))$

Why rank?

Lemma: Pop's got a higher rank:
$\text{rank}(x) < \text{rank}(\pi(x))$

Why rank?

Lemma: Pop's got a higher rank:

$$\text{rank}(x) < \text{rank}(\pi(x))$$

Duh!

Why rank?

Lemma: Pop's got a higher rank:

$$\text{rank}(x) < \text{rank}(\pi(x))$$

Duh!
Code enforces it.
union(x,y):

$\vdots$

if rank(r_x) $<$ rank(r_y):
    $\pi(r_x) = r_y$
else:
    $\pi(r_y) = r_x$
    if rank(r_x) $==$ rank(r_y):
        rank(r_x) $+= 1$

Why rank?

Lemma: Pop's got a higher rank:

$$\text{rank}(x) < \text{rank}(\pi(x))$$

if $x \neq \pi(x)$.

Duh!

Code enforces it.

union(x,y):

$\vdots$

if rank(r_x) $<$ rank(r_y):

$\pi(r_x) = r_y$

else:

$\pi(r_y) = r_x$

if rank(r_x) $==$ rank(r_y):

rank(r_x) $+= 1$

Initially?

Big rank is a big dog!

Big rank is a big dog!

union(x,y):

$\vdots$

if rank(r_x) $<$ rank(r_y):
  $\pi(r_x) = r_y$
else:
  $\pi(r_y) = r_x$
  if rank(r_x) == rank(r_y):
    rank(r_x) $+=1$

Lemma: Any rank k root node has $\geq 2^k$ nodes in its tree.

Big rank is a big dog!

```
union(x,y):
   .
   .
   .
  if rank(r_x) < rank(r_y):
      π(r_x) = r_y
  else:
      π(r_y) = r_x
      if rank(r_x) == rank(r_y):
          rank(r_x) += 1
```

Lemma: Any rank k root node has $\geq 2^k$ nodes in its tree.
Induction:

Big rank is a big dog!

```
union(x,y):
   .
   .
   .
  if rank(r_x) < rank(r_y):
      π(r_x) = r_y
  else:
      π(r_y) = r_x
      if rank(r_x) == rank(r_y):
          rank(r_x) += 1
```

Lemma: Any rank k root node has $\geq 2^k$ nodes in its tree.

Induction:

Base Case

Big rank is a big dog!

```
union(x,y):
    .
    .
    .
  if rank(r_x) < rank(r_y):
      π(r_x) = r_y
  else:
      π(r_y) = r_x
      if rank(r_x) == rank(r_y):
          rank(r_x) += 1
```

Lemma: Any rank k root node has $\geq 2^k$ nodes in its tree.
Induction:
Base Case ?

Big rank is a big dog!

```
union(x,y):
  .
  .
  .
  if rank(r_x) < rank(r_y):
      π(r_x) = r_y
  else:
      π(r_y) = r_x
      if rank(r_x) == rank(r_y):
          rank(r_x) += 1
```

Lemma: Any rank k root node has $\geq 2^k$ nodes in its tree.
Induction:
Base Case ?

(A) $2^0 \geq 1$

(B) $2^1 \geq 1$

Big rank is a big dog!

```
union(x,y):
    ⋮
  if rank(r_x) < rank(r_y):
      π(r_x) = r_y
  else:
      π(r_y) = r_x
      if rank(r_x) == rank(r_y):
          rank(r_x) += 1
```

Lemma: Any rank k root node has $\geq 2^k$ nodes in its tree.

Induction:

Base Case ?

(A) $2^0 \geq 1$

(B) $2^1 \geq 1$

A.

Big rank is a big dog!

```
union(x,y):
    ⋮
  if rank(r_x) < rank(r_y):
        π(r_x) = r_y
  else:
        π(r_y) = r_x
        if rank(r_x) == rank(r_y):
              rank(r_x) += 1
```

Lemma: Any rank k root node has $\geq 2^k$ nodes in its tree.
Induction:
Base Case ?

(A) $2^0 \geq 1$

(B) $2^1 \geq 1$

A. Initially $rank(x) = 0$, 1 node in tree.

Big rank is a big dog!

```
union(x,y):
  .
  .
  .
  if rank(r_x) < rank(r_y):
      π(r_x) = r_y
  else:
      π(r_y) = r_x
      if rank(r_x) == rank(r_y):
          rank(r_x) += 1
```

Lemma: Any rank k root node has $\geq 2^k$ nodes in its tree.
Induction:
Base Case ?

(A) $2^0 \geq 1$

(B) $2^1 \geq 1$

A. Initially $rank(x) = 0$, 1 node in tree.

Induction step:

Big rank is a big dog!

```
union(x,y):
  .
  .
  .
  if rank(r_x) < rank(r_y):
      π(r_x) = r_y
  else:
      π(r_y) = r_x
      if rank(r_x) == rank(r_y):
          rank(r_x) += 1
```

Lemma: Any rank k root node has $\geq 2^k$ nodes in its tree.
Induction:
Base Case ?

(A) $2^0 \geq 1$

(B) $2^1 \geq 1$

A. Initially $rank(x) = 0$, 1 node in tree.

Induction step:
When rank(x) goes up to k.

Big rank is a big dog!

```
union(x,y):
   .
   .
   .
  if rank(r_x) < rank(r_y):
        π(r_x) = r_y
  else:
        π(r_y) = r_x
        if rank(r_x) == rank(r_y):
             rank(r_x) += 1
```

Lemma: Any rank k root node has $\geq 2^k$ nodes in its tree.
Induction:
Base Case ?

(A) $2^0 \geq 1$

(B) $2^1 \geq 1$

A. Initially $rank(x) = 0$, 1 node in tree.

Induction step:
When rank(x) goes up to k.
  rank(x) was $k - 1$

Big rank is a big dog!

union(x,y):

$\vdots$

if rank(r_x) $<$ rank(r_y):
  $\pi(r_x) = r_y$
else:
  $\pi(r_y) = r_x$
  if rank(r_x) == rank(r_y):
    rank(r_x) $+= 1$

Lemma: Any rank k root node has $\geq 2^k$ nodes in its tree.
Induction:
Base Case ?

(A) $2^0 \geq 1$

(B) $2^1 \geq 1$

A. Initially $rank(x) = 0$, 1 node in tree.

Induction step:
When rank(x) goes up to k.
rank(x) was $k-1$ so has $\geq 2^{k-1}$ nodes.

Big rank is a big dog!

```
union(x,y):
  .
  .
  .
  if rank(r_x) < rank(r_y):
      π(r_x) = r_y
  else:
      π(r_y) = r_x
      if rank(r_x) == rank(r_y):
          rank(r_x) += 1
```

Lemma: Any rank k root node has $\geq 2^k$ nodes in its tree.
Induction:
Base Case ?

(A) $2^0 \geq 1$

(B) $2^1 \geq 1$

A. Initially $rank(x) = 0$, 1 node in tree.

Induction step:
When rank(x) goes up to k.
  rank(x) was $k-1$ so has $\geq 2^{k-1}$ nodes. by ind. hyp.

Big rank is a big dog!

```
union(x,y):
  .
  .
  .
  if rank(r_x) < rank(r_y):
      π(r_x) = r_y
  else:
      π(r_y) = r_x
      if rank(r_x) == rank(r_y):
          rank(r_x) += 1
```

Lemma: Any rank k root node has $\geq 2^k$ nodes in its tree.
Induction:
Base Case ?

(A) $2^0 \geq 1$

(B) $2^1 \geq 1$

A. Initially $rank(x) = 0$, 1 node in tree.

Induction step:
When rank(x) goes up to k.
rank(x) was $k-1$ so has $\geq 2^{k-1}$ nodes. by ind. hyp.
gains nodes from rank $k-1$ node

Big rank is a big dog!

```
union(x,y):
    .
    .
    .
  if rank(r_x) < rank(r_y):
      π(r_x) = r_y
  else:
      π(r_y) = r_x
      if rank(r_x) == rank(r_y):
          rank(r_x) += 1
```

Lemma: Any rank k root node has $\geq 2^k$ nodes in its tree.
Induction:
Base Case ?

(A) $2^0 \geq 1$

(B) $2^1 \geq 1$

A. Initially $rank(x) = 0$, 1 node in tree.

Induction step:
When rank(x) goes up to k.
rank(x) was $k-1$ so has $\geq 2^{k-1}$ nodes. by ind. hyp.
gains nodes from rank $k-1$ node with $\geq 2^{k-1}$ nodes

Big rank is a big dog!

union(x,y):

$\vdots$

if rank(r_x) $<$ rank(r_y):
  $\pi(r_x) = r_y$
else:
  $\pi(r_y) = r_x$
  if rank(r_x) == rank(r_y):
    rank(r_x) $+= 1$

Lemma: Any rank k root node has $\geq 2^k$ nodes in its tree.
Induction:
Base Case ?

(A) $2^0 \geq 1$

(B) $2^1 \geq 1$

A. Initially $rank(x) = 0$, 1 node in tree.

Induction step:
When rank(x) goes up to k.
  rank(x) was $k-1$ so has $\geq 2^{k-1}$ nodes. by ind. hyp.
  gains nodes from rank $k-1$ node with $\geq 2^{k-1}$ nodes
$\implies \geq 2^{k-1} + 2^{k-1} = 2^k$ nodes.

Big rank is a big dog!

union(x,y):

$\vdots$

if rank(r_x) $<$ rank(r_y):

  $\pi(r_x) = r_y$

else:

  $\pi(r_y) = r_x$

  if rank(r_x) == rank(r_y):

    rank(r_x) $+= 1$

Lemma: Any rank k root node has $\geq 2^k$ nodes in its tree.
Induction:
Base Case ?

(A) $2^0 \geq 1$

(B) $2^1 \geq 1$

A. Initially $rank(x) = 0$, 1 node in tree.

Induction step:
When rank(x) goes up to k.
rank(x) was $k-1$ so has $\geq 2^{k-1}$ nodes. by ind. hyp.
gains nodes from rank $k-1$ node with $\geq 2^{k-1}$ nodes
$\implies \geq 2^{k-1} + 2^{k-1} = 2^k$ nodes. $\square$

Check your understanding?

Exactly 2^k nodes in tree of rank k?

Check your understanding?

Exactly 2^k nodes in tree of rank k? Yes?

Check your understanding?

Exactly 2^k nodes in tree of rank k? Yes? No?

Check your understanding?

Exactly 2^k nodes in tree of rank k? Yes? No?

No.

Check your understanding?

Exactly 2^k nodes in tree of rank k? Yes? No?

No.

$\vdots$

if $\text{rank}(r_x) < \text{rank}(r_y)$:

$\pi(r_x) = r_y$

$\vdots$

Check your understanding?

Exactly 2^k nodes in tree of rank k? Yes? No?

No.

$\vdots$

if rank(r_x) $<$ rank(r_y):

$\quad\quad \pi(r_x) = r_y$

$\vdots$

Gains nodes without gaining rank!

Back to complexity.

Find(x) is

(A) $O(\log n)$ time.

(B) $O(1)$ time

(C) $O(n)$ time.

Back to complexity.

Find(x) is

(A) $O(\log n)$ time.

(B) $O(1)$ time

(C) $O(n)$ time.

A.

Back to complexity.

Find(x) is

(A) $O(\log n)$ time.

(B) $O(1)$ time

(C) $O(n)$ time.

A. (and (C)).

Back to complexity.

Find(x) is

(A) $O(\log n)$ time.

(B) $O(1)$ time

(C) $O(n)$ time.

A. (and (C)).

Rank k root node has $\geq 2^k$ nodes.

Back to complexity.

Find(x) is

(A) $O(\log n)$ time.

(B) $O(1)$ time

(C) $O(n)$ time.

A. (and (C)).

Rank k root node has $\geq 2^k$ nodes.

Only n nodes in any set.

Back to complexity.

Find(x) is

(A) $O(\log n)$ time.

(B) $O(1)$ time

(C) $O(n)$ time.

A. (and (C)).

Rank k root node has $\geq 2^k$ nodes.

Only n nodes in any set.

Every rank at most $\log n$,

Back to complexity.

Find(x) is

(A) $O(\log n)$ time.

(B) $O(1)$ time

(C) $O(n)$ time.

A. (and (C)).

Rank k root node has $\geq 2^k$ nodes.

Only n nodes in any set.

Every rank at most $\log n$, (otherwise, $> 2^{\log n} = n$ nodes.)

Back to complexity.

Find(x) is

(A) $O(\log n)$ time.

(B) $O(1)$ time

(C) $O(n)$ time.

A. (and (C)).

Rank k root node has $\geq 2^k$ nodes.

Only n nodes in any set.

Every rank at most $\log n$, (otherwise, $> 2^{\log n} = n$ nodes.)

Parent has higher rank.

Back to complexity.

Find(x) is

(A) $O(\log n)$ time.

(B) $O(1)$ time

(C) $O(n)$ time.

A. (and (C)).

Rank k root node has $\geq 2^k$ nodes.

Only n nodes in any set.

Every rank at most $\log n$, (otherwise, $> 2^{\log n} = n$ nodes.)

Parent has higher rank. Code enforces it.

Back to complexity.

Find(x) is

(A) $O(\log n)$ time.

(B) $O(1)$ time

(C) $O(n)$ time.

A. (and (C)).

Rank k root node has $\geq 2^k$ nodes.

Only n nodes in any set.

Every rank at most $\log n$, (otherwise, $> 2^{\log n} = n$ nodes.)

Parent has higher rank. Code enforces it.

Only k steps in find.

Back to complexity.

Find(x) is

(A) $O(\log n)$ time.

(B) $O(1)$ time

(C) $O(n)$ time.

A. (and (C)).

Rank k root node has $\geq 2^k$ nodes.

Only n nodes in any set.

Every rank at most $\log n$, (otherwise, $> 2^{\log n} = n$ nodes.)

Parent has higher rank. Code enforces it.

Only k steps in find.

$O(k)$

Back to complexity.

Find(x) is

(A) $O(\log n)$ time.

(B) $O(1)$ time

(C) $O(n)$ time.

A. (and (C)).

Rank k root node has $\geq 2^k$ nodes.

Only n nodes in any set.

Every rank at most $\log n$, (otherwise, $> 2^{\log n} = n$ nodes.)

Parent has higher rank. Code enforces it.

Only k steps in find.

$O(k) = O(\log n)$ time.

Back to complexity.

Find(x) is

(A) $O(\log n)$ time.

(B) $O(1)$ time

(C) $O(n)$ time.

A. (and (C)).

Rank k root node has $\geq 2^k$ nodes.

Only n nodes in any set.

Every rank at most $\log n$, (otherwise, $> 2^{\log n} = n$ nodes.)

Parent has higher rank. Code enforces it.

Only k steps in find.

$O(k) = O(\log n)$ time.

Yay!

Back to complexity.

Find(x) is

(A) $O(\log n)$ time.

(B) $O(1)$ time

(C) $O(n)$ time.

A. (and (C)).

Rank k root node has $\geq 2^k$ nodes.

Only n nodes in any set.

Every rank at most $\log n$, (otherwise, $> 2^{\log n} = n$ nodes.)

Parent has higher rank. Code enforces it.

Only k steps in find.

$O(k) = O(\log n)$ time.

Yay!

Can we do better?

Back to complexity.

Find(x) is

(A) $O(\log n)$ time.

(B) $O(1)$ time

(C) $O(n)$ time.

A. (and (C)).

Rank k root node has $\geq 2^k$ nodes.

Only n nodes in any set.

Every rank at most $\log n$, (otherwise, $> 2^{\log n} = n$ nodes.)

Parent has higher rank. Code enforces it.

Only k steps in find.

$O(k) = O(\log n)$ time.

Yay!

Can we do better? Yes. We will see better.

Back to complexity.

Find(x) is

(A) $O(\log n)$ time.

(B) $O(1)$ time

(C) $O(n)$ time.

A. (and (C)).

Rank k root node has $\geq 2^k$ nodes.

Only n nodes in any set.

Every rank at most $\log n$, (otherwise, $> 2^{\log n} = n$ nodes.)

Parent has higher rank. Code enforces it.

Only k steps in find.

$O(k) = O(\log n)$ time.

Yay!

Can we do better? Yes. We will see better.

Kruskal Implementation.

$|V|$ unions. $|E|$ finds.

Kruskal Implementation.

$|V|$ unions. $|E|$ finds.

$O(|E| \log n)$ time!

Kruskal Implementation.

$|V|$ unions. $|E|$ finds.

$O(|E| \log n)$ time!

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

Lecture in a minute.

Tree Definitions:

$n-1$ edges and connected.

$n-1$ edges and no cycles.

All pairs of vertices connected by unique path.

Lecture in a minute.

Tree Definitions:

$n-1$ edges and connected.

$n-1$ edges and no cycles.

All pairs of vertices connected by unique path.

Minimum Spanning Tree: $G=(V,E)$, weights $w:E$

Kruskal: Sort edges.

Add edges in this order if no cycle.

Cut property:

Exists MST with minimum weight edge across cut.

Lecture in a minute.

Tree Definitions:

$n-1$ edges and connected.

$n-1$ edges and no cycles.

All pairs of vertices connected by unique path.

Minimum Spanning Tree: $G=(V,E)$, weights $w:E$

Kruskal: Sort edges.

Add edges in this order if no cycle.

Cut property:

Exists MST with minimum weight edge across cut.

Union-Find Data Structure.

Pointer implementation: $\pi(u)$.

makeset(s) – $\pi(u)=u$.

find(x) – returns root of pointer structure.

Lecture in a minute.

Tree Definitions:

$n-1$ edges and connected.

$n-1$ edges and no cycles.

All pairs of vertices connected by unique path.

Minimum Spanning Tree: $G = (V, E)$, weights $w : E$

Kruskal: Sort edges.

Add edges in this order if no cycle.

Cut property:

Exists MST with minimum weight edge across cut.

Union-Find Data Structure.

Pointer implementation: $\pi(u)$.

makeset(s) – $\pi(u) = u$.

find(x) – returns root of pointer structure.

union(x,y) – $\pi(find(x)) = \pi(find(y))$.

Lecture in a minute.

Tree Definitions:

$n-1$ edges and connected.

$n-1$ edges and no cycles.

All pairs of vertices connected by unique path.

Minimum Spanning Tree: $G = (V, E)$, weights $w : E$

Kruskal: Sort edges.

Add edges in this order if no cycle.

Cut property:

Exists MST with minimum weight edge across cut.

Union-Find Data Structure.

Pointer implementation: $\pi(u)$.

makeset(s) – $\pi(u) = u$.

find(x) – returns root of pointer structure.

union(x,y) – $\pi(find(x)) = \pi(find(y))$.

Lecture in a minute.

Tree Definitions:

$n-1$ edges and connected.

$n-1$ edges and no cycles.

All pairs of vertices connected by unique path.

Minimum Spanning Tree: $G = (V, E)$, weights $w : E$

Kruskal: Sort edges.

Add edges in this order if no cycle.

Cut property:

Exists MST with minimum weight edge across cut.

Union-Find Data Structure.

Pointer implementation: $\pi(u)$.

makeset(s) – $\pi(u) = u$.

find(x) – returns root of pointer structure.

union(x,y) – $\pi(find(x)) = \pi(find(y))$.

Union by rank: $O(\log n)$ depth for pointer structure.

Lecture in a minute.

Tree Definitions:

$n-1$ edges and connected.

$n-1$ edges and no cycles.

All pairs of vertices connected by unique path.

Minimum Spanning Tree: $G = (V, E)$, weights $w : E$

Kruskal: Sort edges.

Add edges in this order if no cycle.

Cut property:

Exists MST with minimum weight edge across cut.

Union-Find Data Structure.

Pointer implementation: $\pi(u)$.

makeset(s) – $\pi(u) = u$.

find(x) – returns root of pointer structure.

union(x,y) – $\pi(find(x)) = \pi(find(y))$.

Union by rank: $O(\log n)$ depth for pointer structure.

union(x,y) - point to larger rank root.

Lecture in a minute.

Tree Definitions:

$n-1$ edges and connected.

$n-1$ edges and no cycles.

All pairs of vertices connected by unique path.

Minimum Spanning Tree: $G=(V,E)$, weights $w:E$

Kruskal: Sort edges.

Add edges in this order if no cycle.

Cut property:

Exists MST with minimum weight edge across cut.

Union-Find Data Structure.

Pointer implementation: $\pi(u)$.

makeset(s) – $\pi(u)=u$.

find(x) – returns root of pointer structure.

union(x,y) – $\pi(find(x))=\pi(find(y))$.

Union by rank: $O(\log n)$ depth for pointer structure.

union(x,y) - point to larger rank root.

increase rank if tied.

Lecture in a minute.

Tree Definitions:
- $n-1$ edges and connected.
- $n-1$ edges and no cycles.
- All pairs of vertices connected by unique path.

Minimum Spanning Tree: $G=(V,E)$, weights $w:E$
- Kruskal: Sort edges.
  - Add edges in this order if no cycle.
- Cut property:
  - Exists MST with minimum weight edge across cut.

Union-Find Data Structure.
- Pointer implementation: $\pi(u)$.
- makeset(s) – $\pi(u)=u$.
- find(x) – returns root of pointer structure.
- union(x,y) – $\pi(find(x))=\pi(find(y))$.

Union by rank: $O(\log n)$ depth for pointer structure.
- union(x,y) - point to larger rank root.
  - increase rank if tied.
- $>2^k$ nodes in rank k root tree.
- $O(\log n)$ depth structure.

See you on Wednesday!