Kruskal's Minimum Spanning Tree Algorithm & Union-Find Data Structures

Slides by Carl Kingsford

Jan. 21, 2013

Reading: AD 4.5–4.6

Greedy minimum spanning tree rules

All of these greedy rules work:

1. Starting with any root node, add the frontier edge with the smallest weight. (**Prim's Algorithm**)

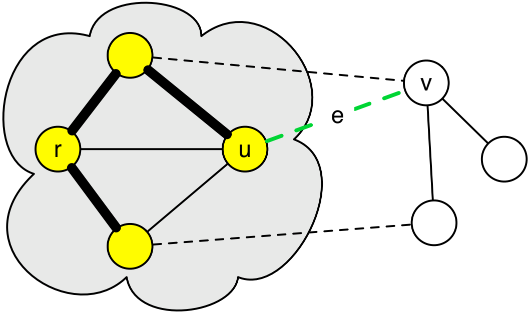
2. Add edges in increasing weight, skipping those whose addition would create a cycle. (**Kruskal's Algorithm**)

3. Start with all edges, remove them in decreasing order of weight, skipping those whose removal would disconnect the graph. (**"Reverse-Delete" Algorithm**)

Prim's Algorithm

Prim's Algorithm: Starting with any root node, add the frontier edge with the smallest weight.

Theorem. *Prim's algorithm produces a minimum spanning tree.*



S = set of nodes already in the tree when e is added

Cycle Property

Theorem (Cycle Property). *Let C be a cycle in G. Let $e = (u, v)$ be the edge with maximum weight on C. Then e is not in any MST of G.*

Suppose the theorem is false. Let T be a MST that contains e.

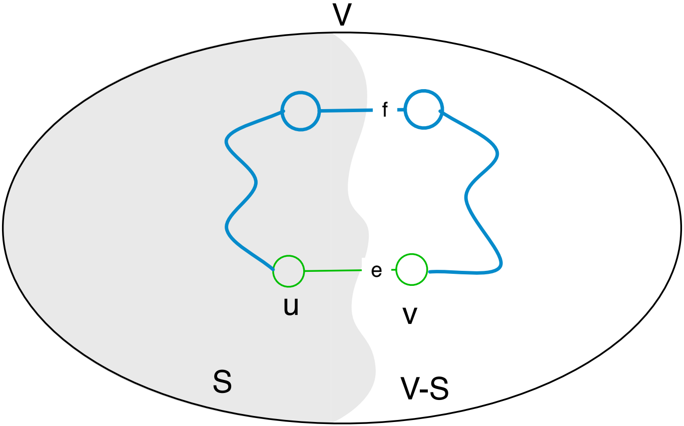
Deleting e from T partitions vertices into 2 sets:

$$S \text{ (that contains } u\text{) and } V - S \text{ (that contains } v\text{).}$$

Cycle C must have some *other* edge f that goes from S and $V - S$.

Replacing e by f produces a lower cost tree, contradicting that T is an MST.

Cycle Property, Picture



MST Property Summary

1. Cut Property: The smallest edge crossing any cut must be in all MSTs.

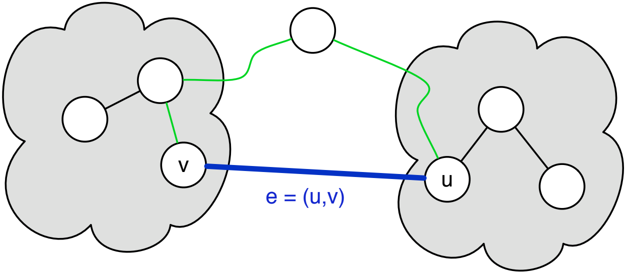
2. Cycle Property: The largest edge on any cycle is never in any MST.

Reverse-Delete Algorithm

Reverse-Delete Algorithm: Remove edges in decreasing order of weight, skipping those whose removal would disconnect the graph.

Theorem. *Reverse-Delete algorithm produces a minimum spanning tree.*

Because removing e won't disconnect the graph,
there must be another path between u and v



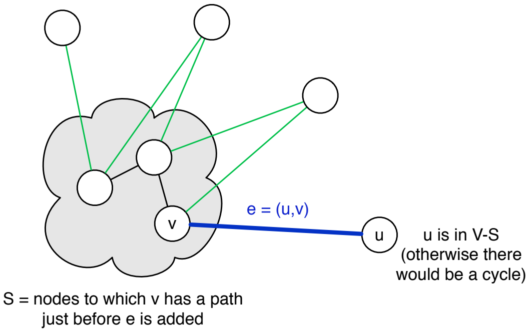
Because we're removing in order of decreasing weight,
e must be the largest edge on that cycle.

Kruskal's Algorithm

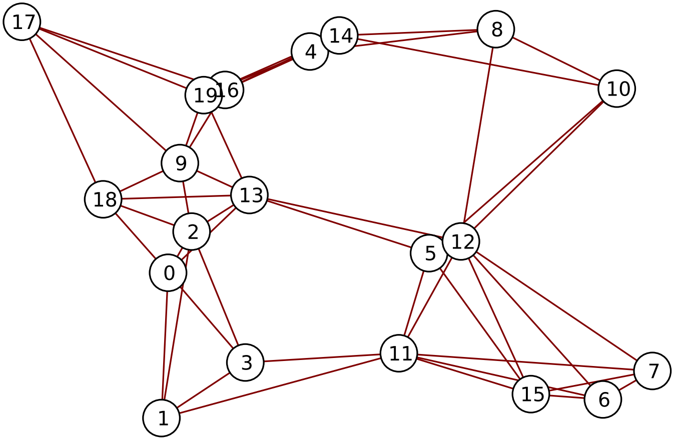
Kruskal's Algorithm: Add edges in increasing weight, skipping those whose addition would create a cycle.

Theorem. *Kruskal's algorithm produces a minimum spanning tree.*

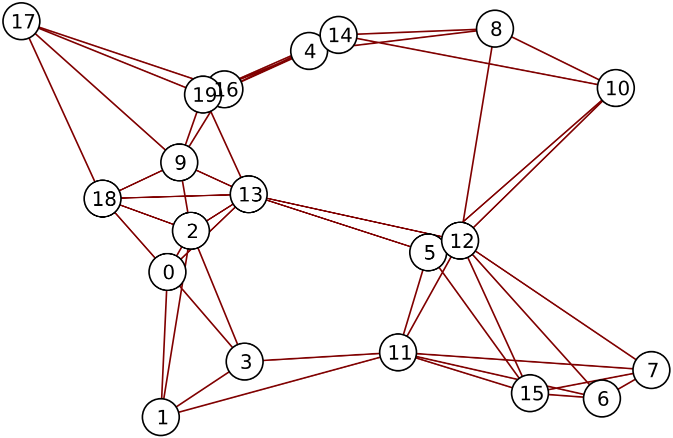
Proof. Consider the point when edge $e = (u, v)$ is added:



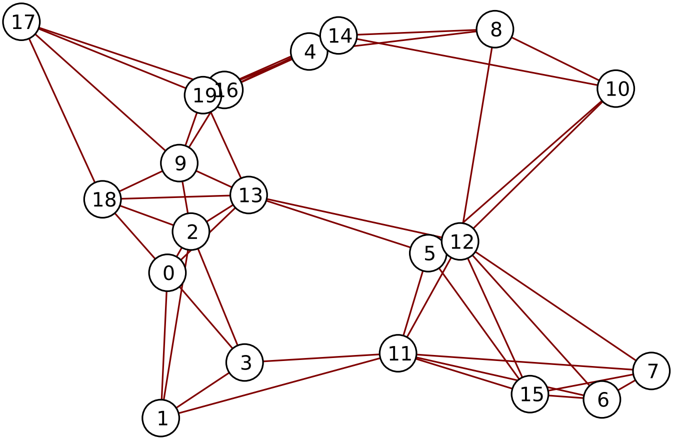
Example run of Kruskal's



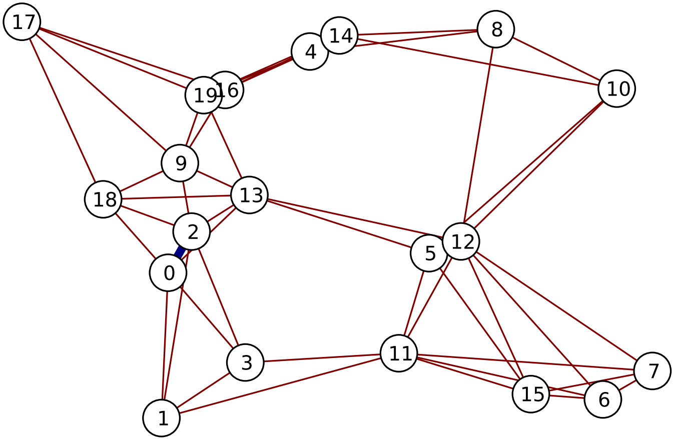
Example run of Kruskal's



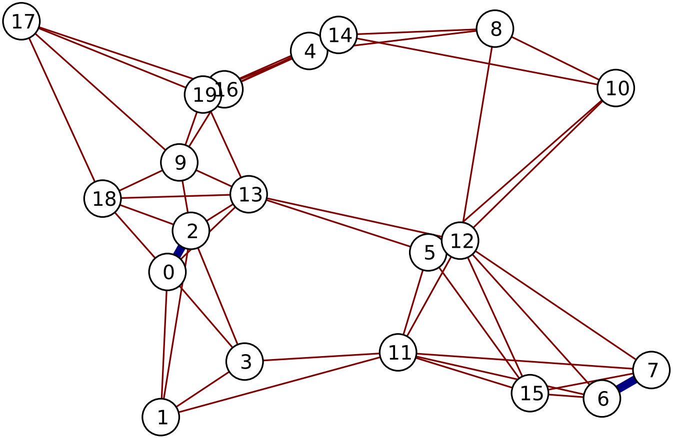
Example run of Kruskal's



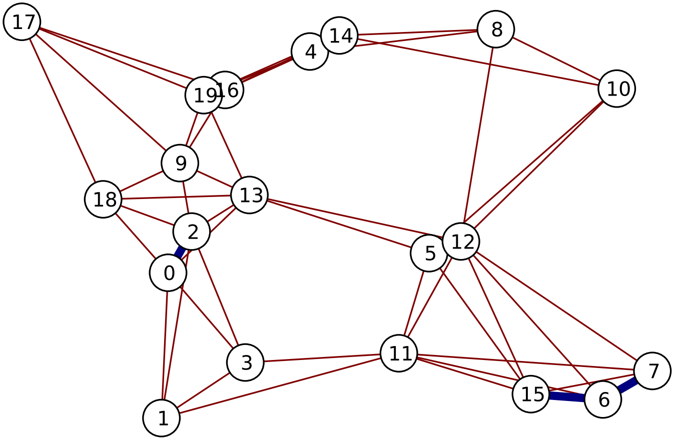
Example run of Kruskal's



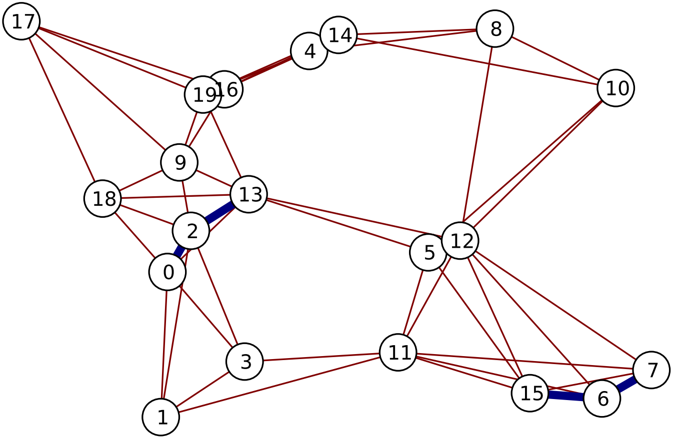
Example run of Kruskal's



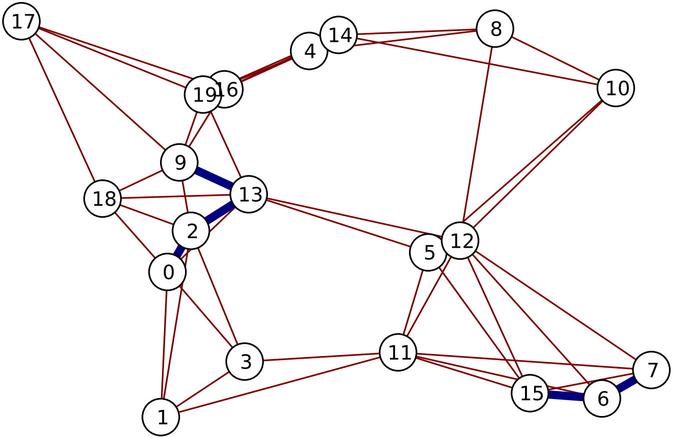
Example run of Kruskal's



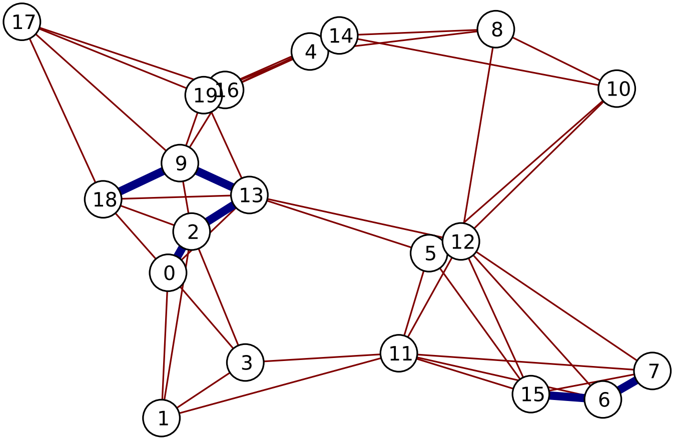
Example run of Kruskal's



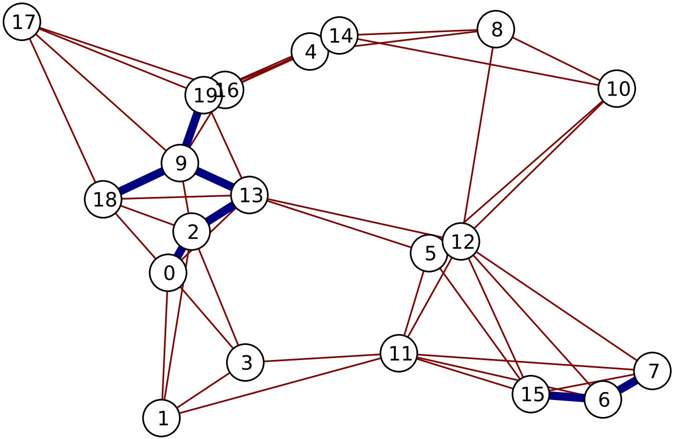
Example run of Kruskal's



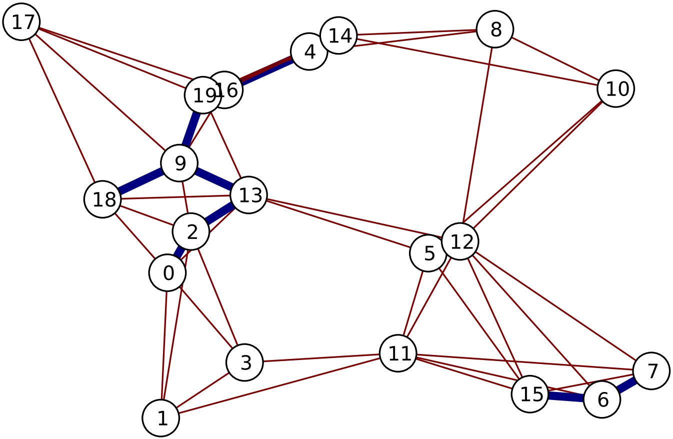
Example run of Kruskal's



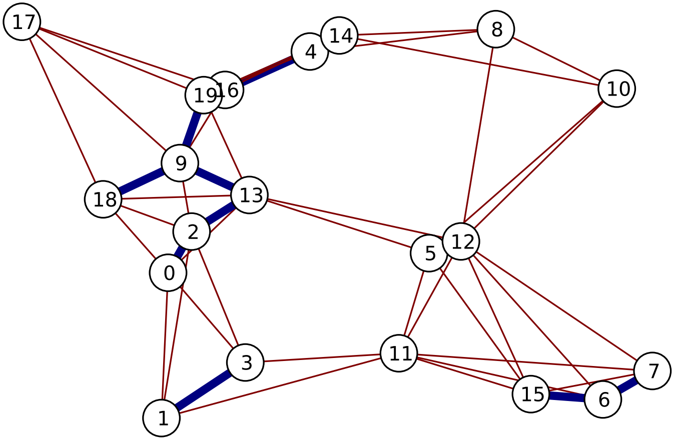
Example run of Kruskal's



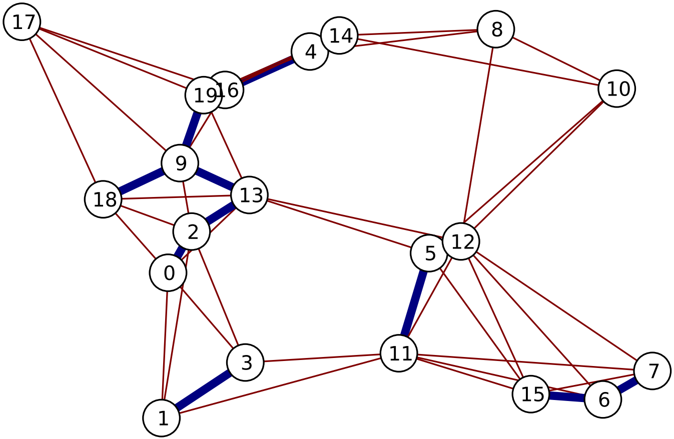
Example run of Kruskal's



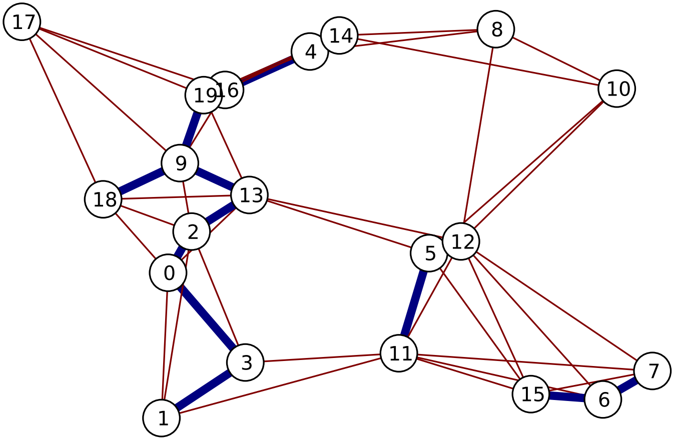
Example run of Kruskal's



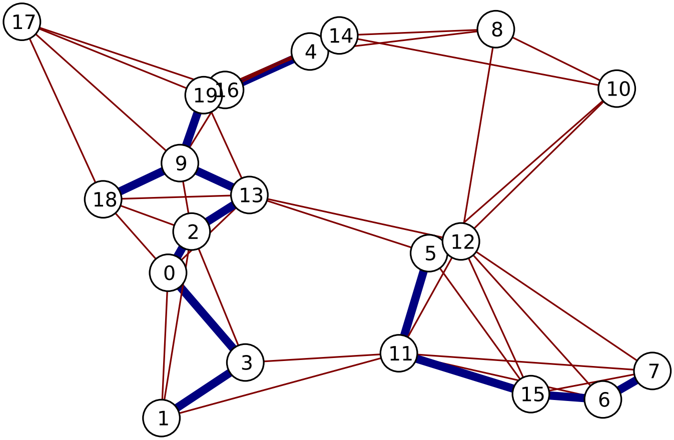
Example run of Kruskal's



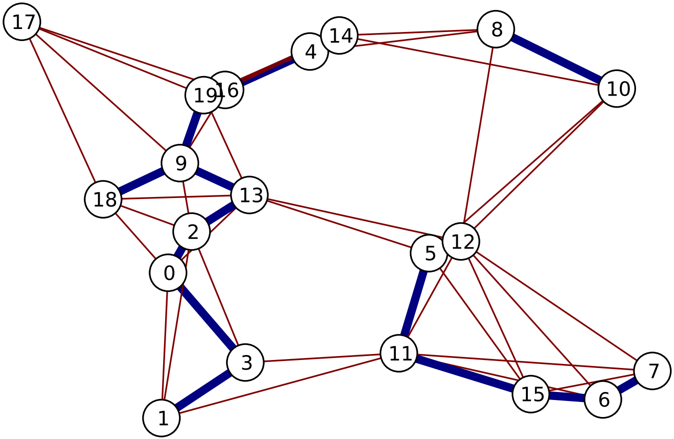
Example run of Kruskal's



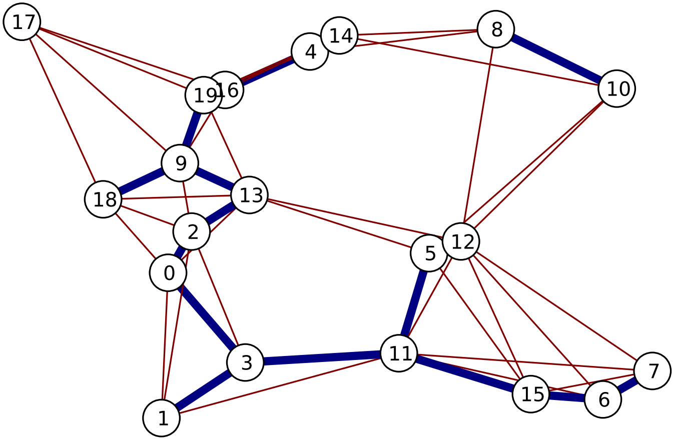
Example run of Kruskal's



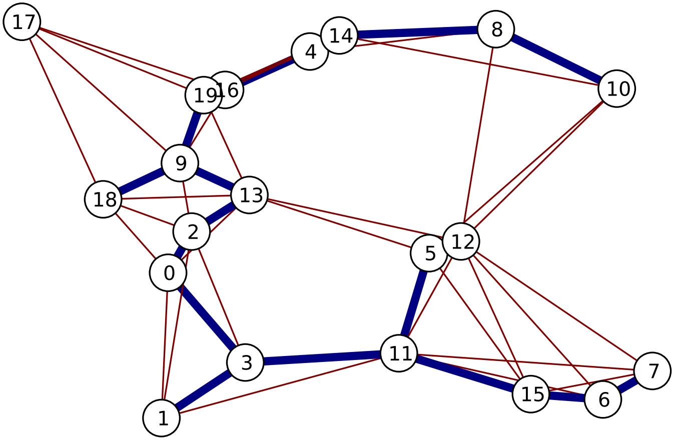
Example run of Kruskal's



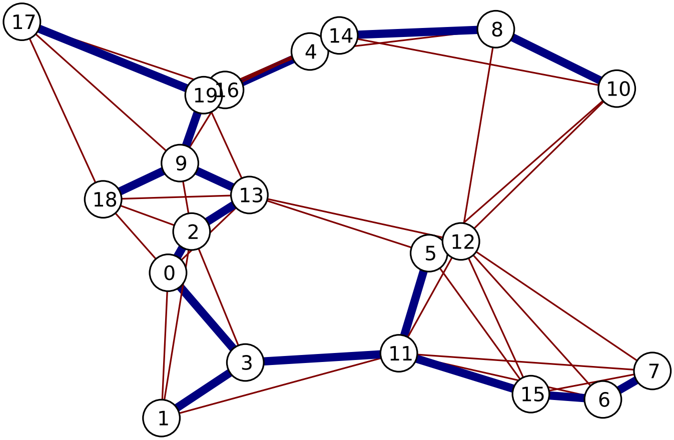
Example run of Kruskal's



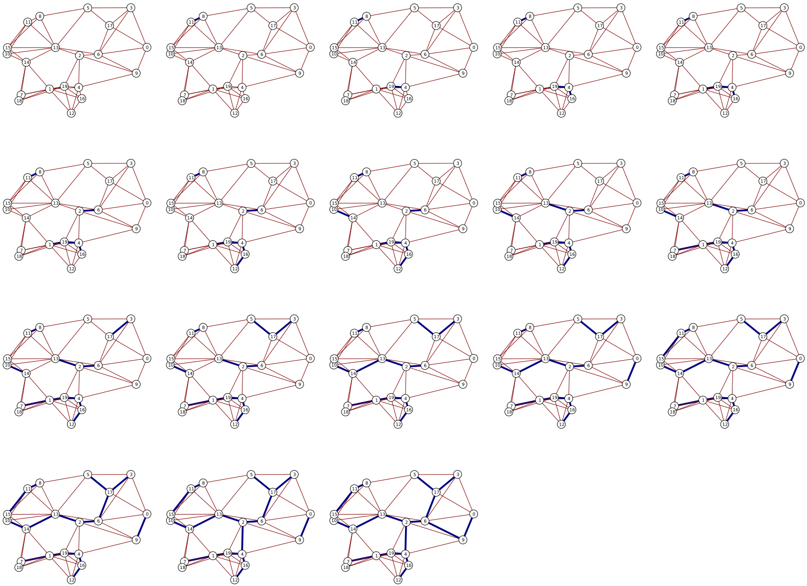
Example run of Kruskal's



Example run of Kruskal's



Another example



Data Structure for Kruskal's Algorithm

Kruskal's Algorithm: Add edges in increasing weight, skipping those whose addition would create a cycle.

How would we check if adding an edge $\{u, v\}$ would create a cycle?

Data Structure for Kruskal's Algorithm

Kruskal's Algorithm: Add edges in increasing weight, skipping those whose addition would create a cycle.

How would we check if adding an edge $\{u, v\}$ would create a cycle?

- Would create a cycle if u and v are already in the same component.

Data Structure for Kruskal's Algorithm

Kruskal's Algorithm: Add edges in increasing weight, skipping those whose addition would create a cycle.

How would we check if adding an edge $\{u, v\}$ would create a cycle?

- Would create a cycle if u and v are already in the same component.
- We start with a component for each node.

Data Structure for Kruskal's Algorithm

Kruskal's Algorithm: Add edges in increasing weight, skipping those whose addition would create a cycle.

How would we check if adding an edge $\{u, v\}$ would create a cycle?

- Would create a cycle if u and v are already in the same component.
- We start with a component for each node.
- Components merge when we add an edge.

Data Structure for Kruskal's Algorithm

Kruskal's Algorithm: Add edges in increasing weight, skipping those whose addition would create a cycle.

How would we check if adding an edge $\{u, v\}$ would create a cycle?

- Would create a cycle if u and v are already in the same component.
- We start with a component for each node.
- Components merge when we add an edge.
- Need a way to: check if u and v are in same component and to merge two components into one.

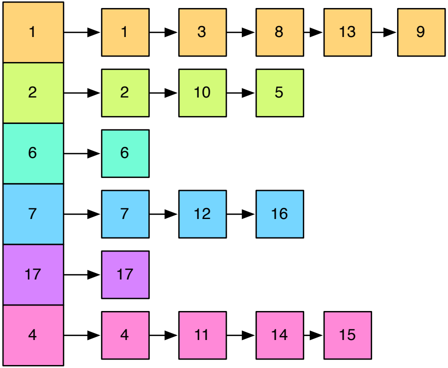
Union-Find Abstract Data Type

The Union-Find abstract data type supports the following operations:

- UF.create(S) — create the data structure containing $|S|$ sets, each containing one item from S.
- UF.find(i) — return the "name" of the set containing item i.
- UF.union(a,b) — merge the sets with names a and b into a single set.

A Union-Find Data Structure

UF Items:



UF Sizes:

1	5
2	3
6	1
7	3
17	1
4	4

UF Sets Array:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	2	1	4	2	6	7	1	1	2	4	7	1	4	4	7	17

Implementing the union & find operations

make_union_find(S) Create data structures on previous slide.
Takes time proportional to the size of S.

find(i) Return UF.sets[i].
Takes a constant amount of time.

union(x,y) Use the "size" array to decide which set is smaller.
Assume x is smaller.
Walk down elements i in set x, setting sets[i] $=$ y.
Set size[y] $=$ size[y] $+$ size[x].

Runtime of array-based Union-Find

Theorem. *Any sequence of k union operations on a collection of n items takes time at most proportional to $k \log k$.*

Proof. After k unions, at most $2k$ items have been involved in a union. (Each union can touch at most 2 new items).

We upper bound the number of times set[v] changes for any v:

- Every time set[v] changes, the size of the set that v is in at least doubles. why?
- So, set[v] can have changed at most $\log_2(2k)$ times.

At most $2k$ items have been modified at all, and each updated at most $\log_2(2k)$ times $\implies 2k \log_2(2k)$ work. $\square$

Running time of Kruskal's algorithm

Sorting the edges: $\approx m \log m$ for m edges.

$$m \leq n^2, \text{ so } \log m < \log n^2 = 2 \log n$$

Therefore sorting takes $\approx m \log n$ time.

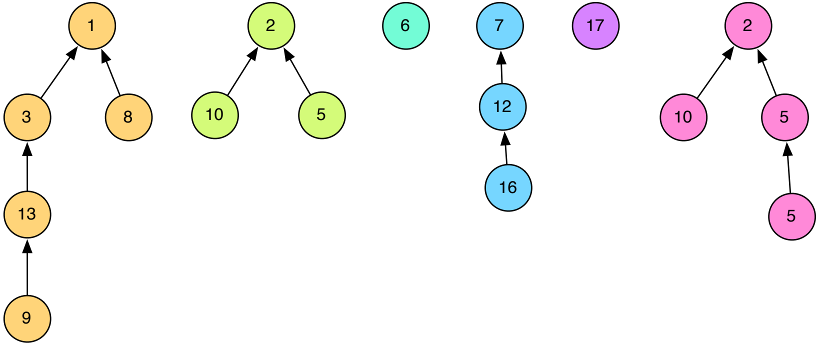
At most $2m$ "find" operations: $\approx 2m$ time.

At most $n - 1$ union operations: $\approx n \log n$ time.

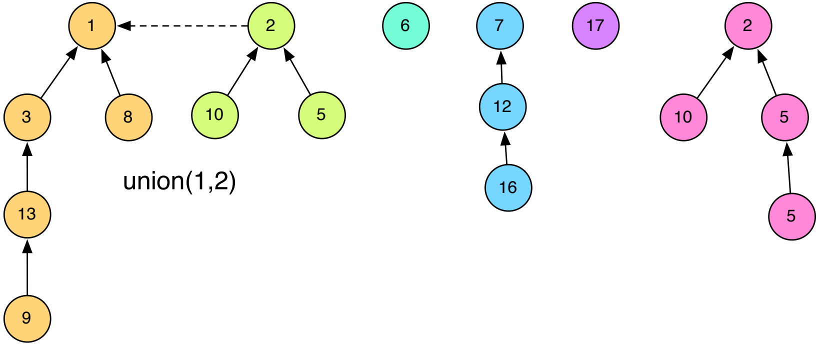
$\implies$ Total running time of $\approx m \log n + 2m + n \log n$.

The biggest term is $m \log n$ since $m > n$ if the graph is connected.

Another way to implement Union-Find



Another way to implement Union-Find



Tree-based Union-Find

make_union_find(S) Create $|S|$ trees each containing a single item and size 1. Takes time proportional to the size of S.

find(i) Follow the pointer from i to the root of its tree.

union(x,y) If the size of set x is $<$ that of y, make y point to x. Takes constant time.

Runtime of tree-based Find

Theorem. *find(i) takes time $\approx \log n$ in a tree-based union-find data structure containing n items.*

Proof. The depth of an item equals the number of times the set it was in was renamed.

The size of the set containing v at least doubles every time the name of the set containing v is changed.

The largest number of times the size can double is $\log_2 n$. □

Running time of Kruskal's algorithm using tree-based union-find

Same running time as using the array-based union-find:

- Sorting the edges: $\approx m \log n$ for m edges.
- At most $2m$ "find" operations: $\approx \log n$ time each.
- At most $n - 1$ union operations: $\approx n$ time.

$\Longrightarrow$ Total running time of $\approx m \log n + 2m \log n + n$.

The biggest term is $m \log n$ since $m > n$ if the graph is connected.