Kruskal’s MST Algorithm and Find-Union Data Structure

CS 4102: Algorithms

Spring 2022

Robbie Hott and Tom Horton

Topics in this slide-deck:

- Motivating Problem: Minimum Spanning Trees
  - This is a graph problem, and you've seen it
- One solution
  - Kruskal's Algorithm (Uses a find-union structure)
- Define and design the find-union to support Kruskal's Algorithm
  - Will require some clever implementation details

Minimum Spanning Trees

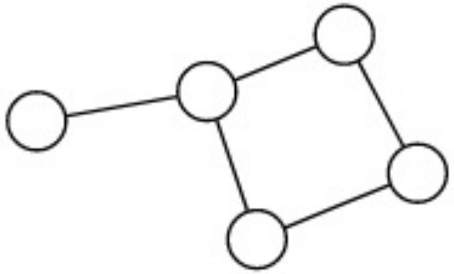
Readings: CLRS 23
(but not 23.1 and only first part of 23.2)

Spanning Tree

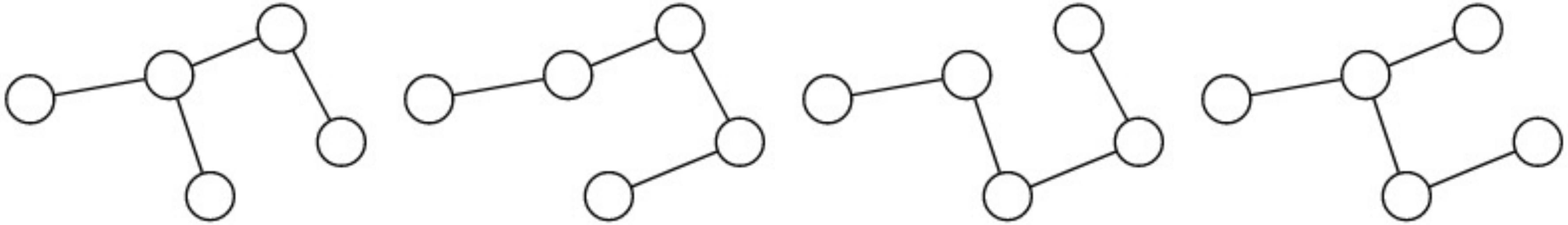
- A ***spanning tree*** of a graph G is a subgraph of G that contains every vertex in G and is also a ***tree*** (i.e., it has no cycles)
  – All connected graphs have spanning tree(s)
  – All spanning trees have the same number of nodes (all of them)
  – You can construct a spanning tree by arbitrarily remove edges from cycles

Spanning Tree: Example

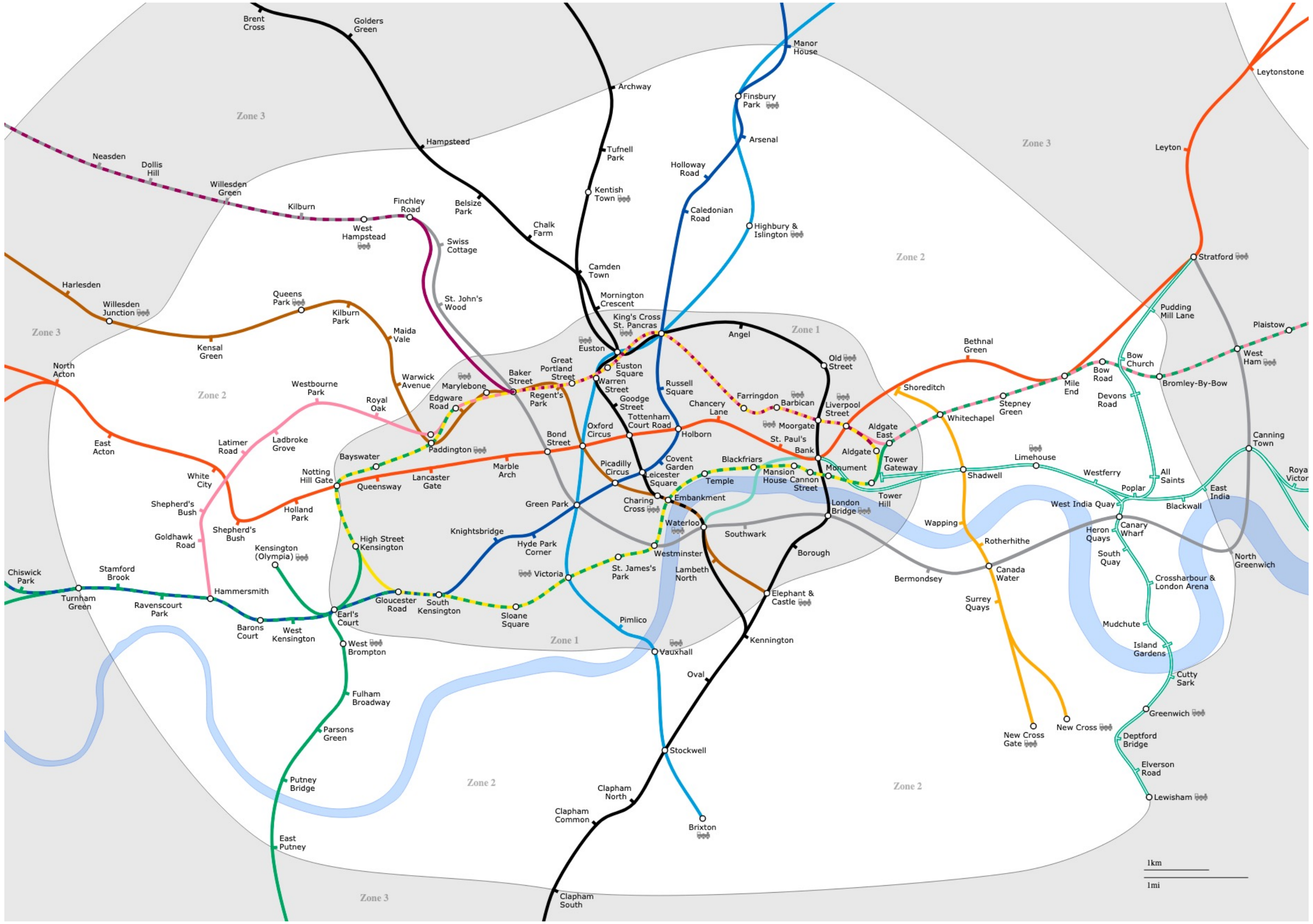
- Original Graph:



- Possible spanning trees:



Spanning Tree: Example (almost)



Minimum Spanning Tree

- Just constructing any spanning tree is simple

- Suppose edges have costs!
  - Cost of building tracks between two stations
  - Length of wire between boxes in a house

- Each spanning tree has a different total cost (sum of edges included in tree)

- The ***Minimum Spanning Tree*** is the spanning tree with lowest overall cost

Minimum Spanning Tree

- Given a connected and undirected graph G=(V, E)

- Find a graph G’ = (V, E’) such that:
  - E’ is a subset of E
  - |E’| = |V| - 1
  - G’ is connected (assuming G was connected)
  - Sum of cost of edges in E’ is minimum

- G’ is then the minimum spanning tree

Kruskal’s MST Algorithm

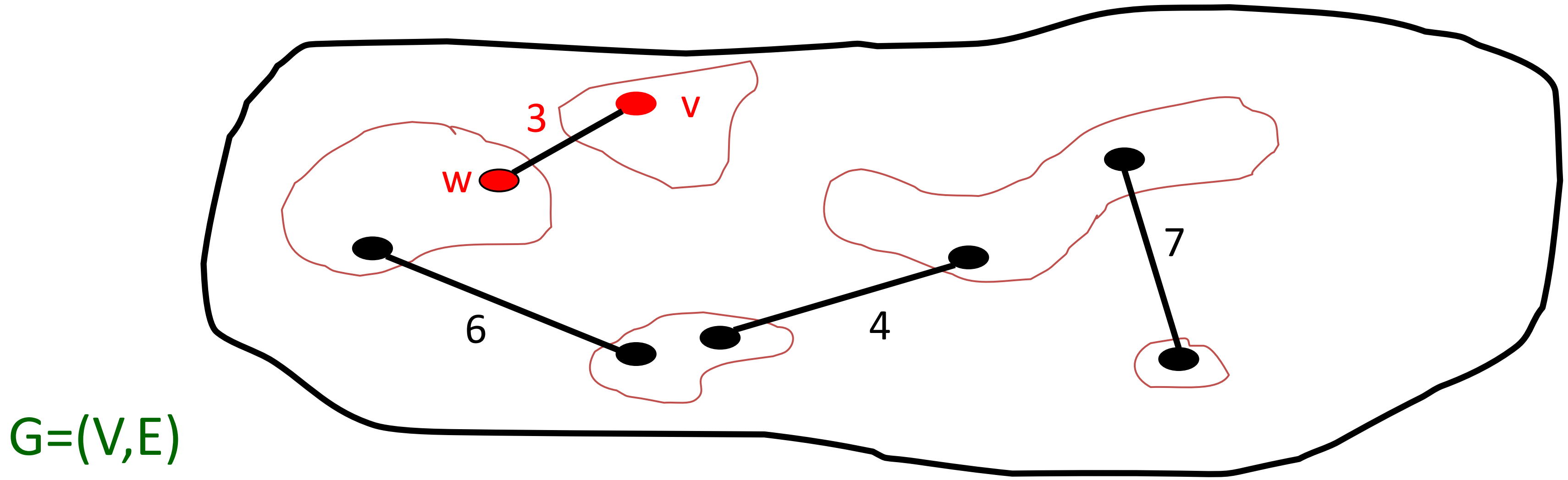
Readings: CLRS first part of 23.2

Kruskal's MST Algorithm

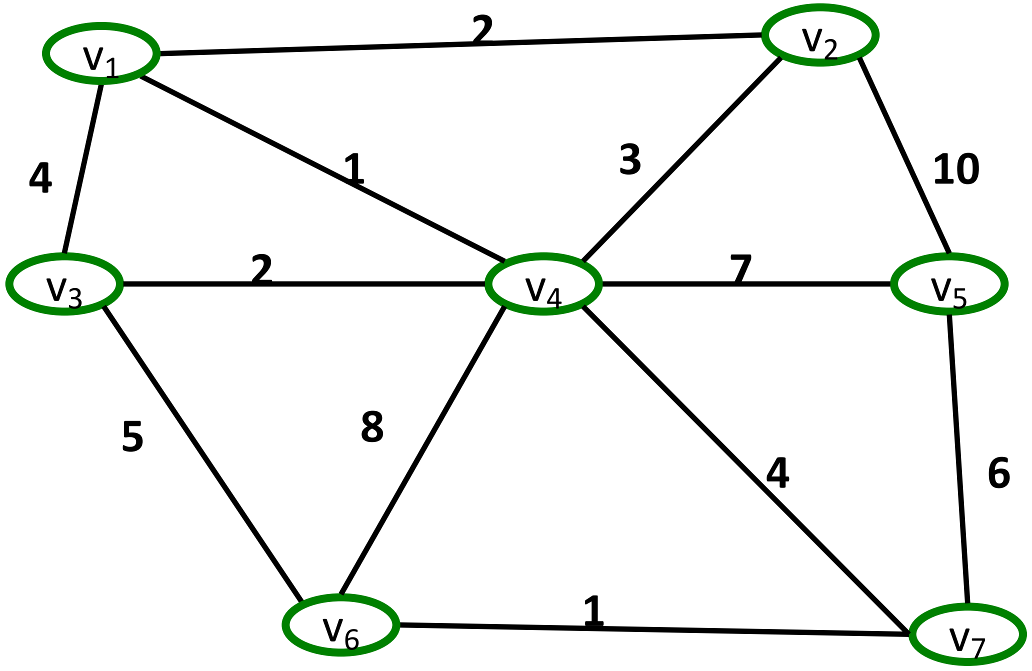
- Prim's approach:
  - Build one tree. Make the one tree bigger and as good as it can be.
- Kruskal's approach
  - Choose the best edge possible: smallest weight
  - Not one tree – maintain a forest!
  - Each edge added will connect two trees.
    Can't form a cycle in a tree!
  - After adding n-1 edges, you have one tree, the MST

Kruskal's MST Algorithm

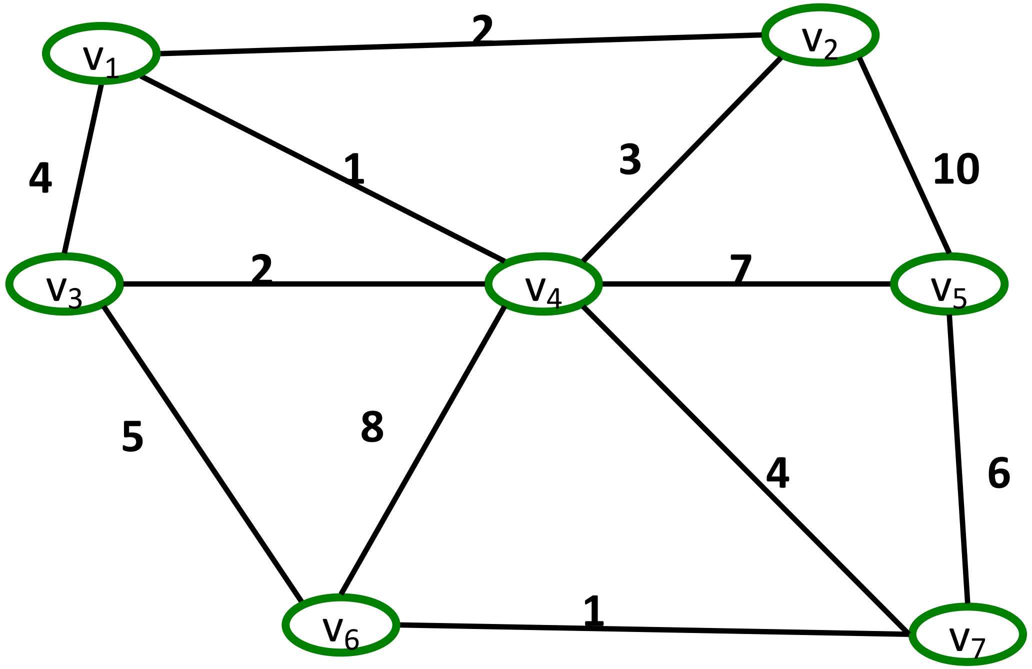
- Idea: Have a forest (set of trees) that eventually shrinks into one tree
  - At each step, add an edge that joins two trees (no cycles!)
  - Choose the one (v,w) that has the smallest weight of possible connecting edges
  - Continue until you have one tree, which will be a MST

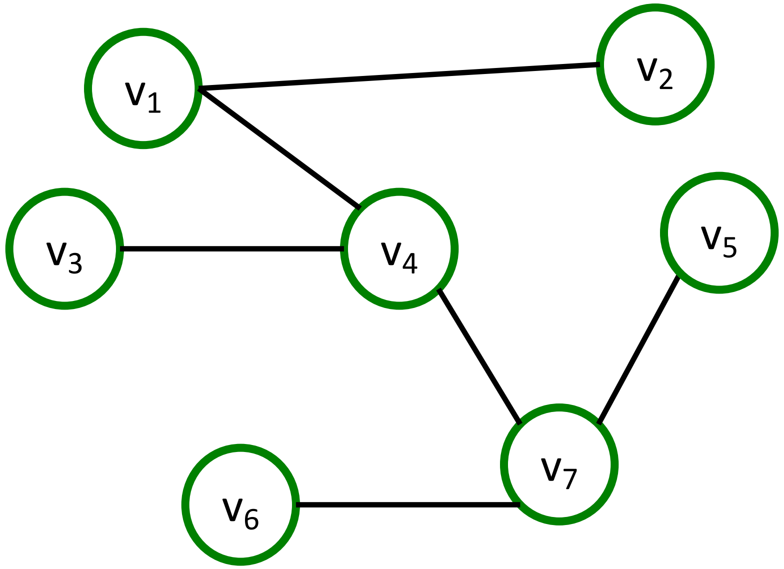


MST Example



MST and Kruskal's Example





Cost(MST) = 16

Kruskal code

```
void Graph::kruskal(){
  int edgesAccepted = 0;
  DisjSet s(NUM_VERTICES);

  while (edgesAccepted < NUM_VERTICES – 1){
    e = smallest weight edge not deleted yet;
    // edge e = (u, v)
    uset = s.find(u);
    vset = s.find(v);
    if (uset != vset){
      edgesAccepted++;
      s.unionSets(uset, vset);
    }
  }
}
```

Assumes we've created a heap. (Could we sort?)

Initialize DisjSet object so all items in separate set

|E| heap ops

2|E| finds

|V| unions

Runtime of Kruskal's

- Every edge is placed on priority queue once and removed once
  - $\Theta(E * \log(E)) = \Theta(E * \log(V))$

- For each edge you do 2 set finds and one set union.
  - Let f(V) be time of find, and u(V) be time of union.
  - $\Theta\left(E * \left(2f(V) + u(V)\right)\right)$
  - If find and union are linear time, then $\Theta\left(E * (2V + V)\right) = \Theta(E * V) = O(V^3)$

- Overall: $\Theta(E * \log(V) + E * V) = \Theta(E * V) = \boldsymbol{O\left(V^3\right)}$ *//Assumes find and union linear time*

Strategy for Kruskal's

- EL = sorted set of edges ascending by weight
  - (For this discussion, we're sorting here, not using a heap)
- Foreach edge e in EL
  - T1 = tree for head(e)
  - T2 = tree for tail(e)
  - If (T1 != T2)
    - add e to the output (the MST)
    - Combine trees T1 and T2

- Seems simple, no?
  - But, how do you keep track of what trees a vertex is in?
  - Trees are sets of vertices. Need to findset(v) and "union" two sets

Disjoint Sets and Find/Union Algorithms

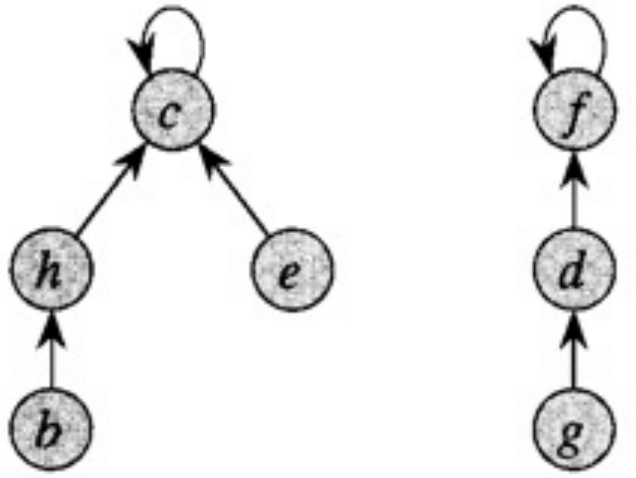
Readings: CLRS 21.3

Union/Find and Disjoint Sets

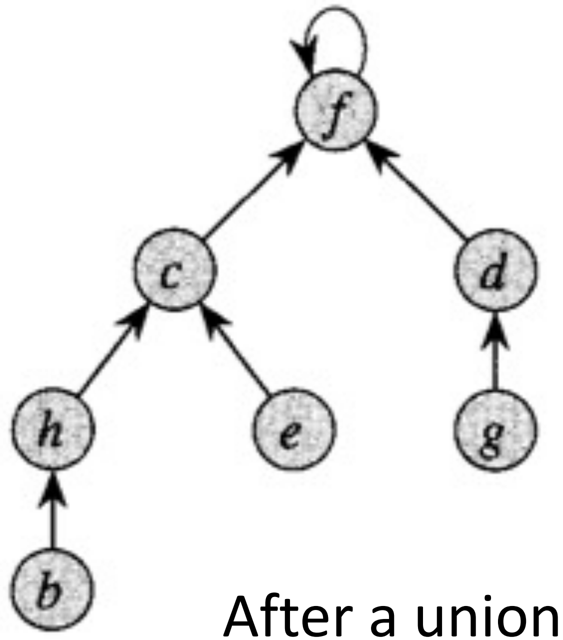
- An Abstract Data Type (ADT) for a collection of sets of any kind of item, where an item can only belong to one of the sets
  – We'll assume each item is identified by a unique integer value
- Need to support the following operations
  – void makeSet(int n) // construct n independent sets
  – int findSet(int i) // given i, which set does i belong to?
  – void union(int i, int j) // merge sets containing i and j

Represent Sets As Trees

- In our implementation, we'll represent each set as a tree
- Identify set by its root node's ID (its "label")
  - findSet() means tracing up to root
  - union() makes one root child of the other root



Two sets



After a union

Union/Find and Disjoint Sets

- Needs to support the following operations
  - void makeSet(int n)   //construct n independent sets

- Solution:
  - Store as array of size n. Each location stores label for that set.

0	1	2	3	4	5	6	7
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*

Union/Find and Disjoint Sets

- Needs to support the following operations
  - int findSet(int i)   //given i, which set does i belong to?

- Solution: Trace around array until we find place where index and contents match
  - Start at index i and repeat:
    - If a[i] == i then return i
    - Else set i = a[i]

0	1	2	3	4	5	6	7
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*

Union/Find and Disjoint Sets

- Needs to support the following operations
  - void union(int i, int j)   //merge sets i and j

- Solution: find label for each set (call find() method), then set one label to point to other
  - Label1 = find(i); Label2 = find(j)
  - a[Label1] = Label2 //OR a[Label2] = Label1

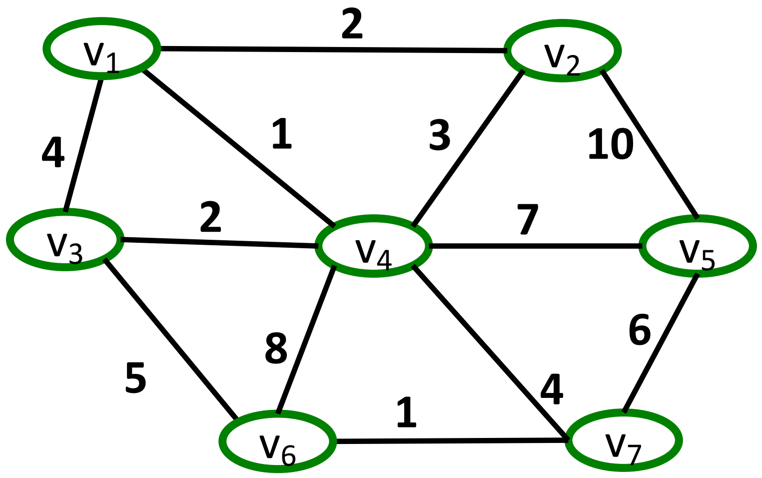
0	1	2	3	4	5	6	7
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*

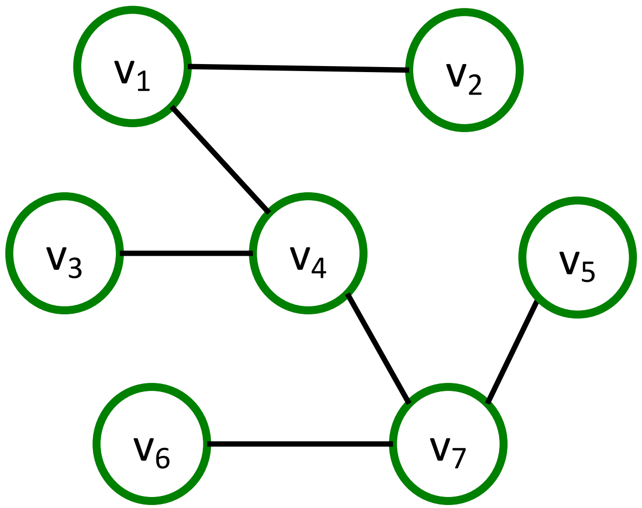
Union/Find and Disjoint Sets

- Example:
  - union(4,5)
  - union(6,7)
  - union(1,2)
  - union(5,6)
  - find(1); find(4); find(6)

0	1	2	3	4	5	6	7
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*

Example Using MST Example





Union/Find and Disjoint Sets

- Time-complexity, where n is size of array?

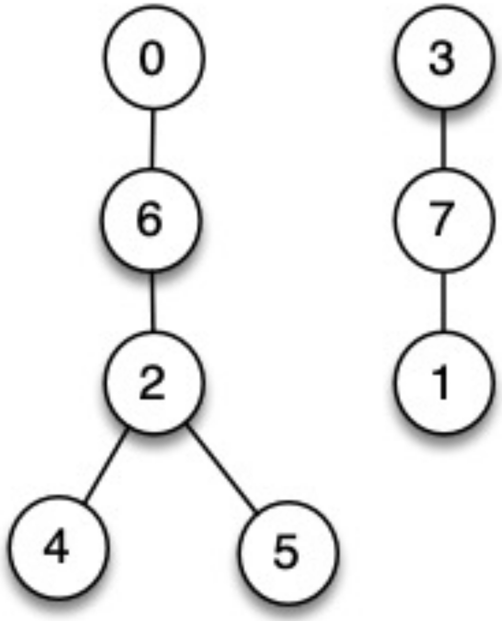
- makeSet()
  - Linear: just create array and fill it with values

- find()
  - Linear if have to trace a long way to get to label
  - Constant if lucky and input is the label (root note) or near it

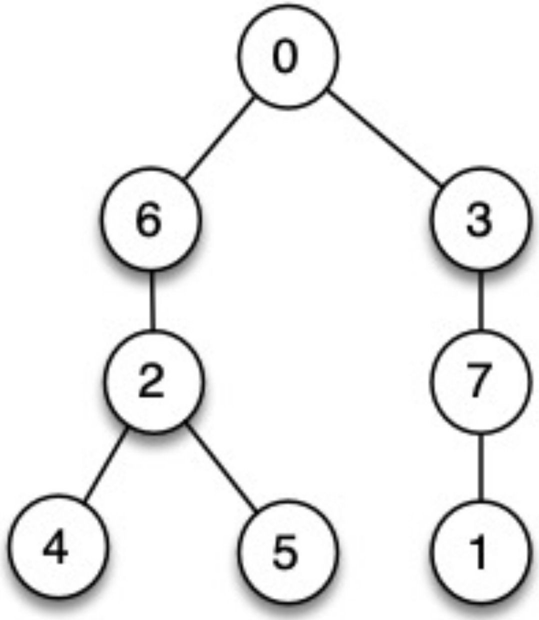
- union()
  - Constant to change the label BUT…
  - Could be linear to find the two labels first.

Optimization 1: Union by rank

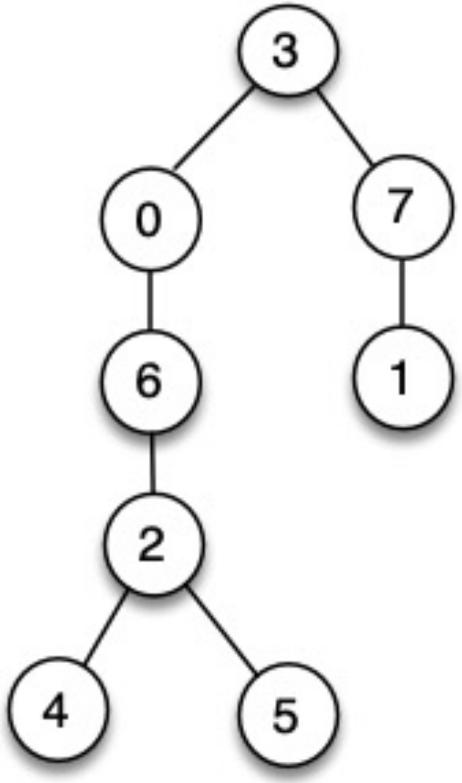
Two Sets:



Union’d under 0:



Union’d under 3:



Optimization 1: Union by rank

- Easy to implement!!
- What's "rank" here?
  - Upper bound on height of a node in our set's tree
- Union by rank:
  - Make the root with smaller rank point to the root with larger rank

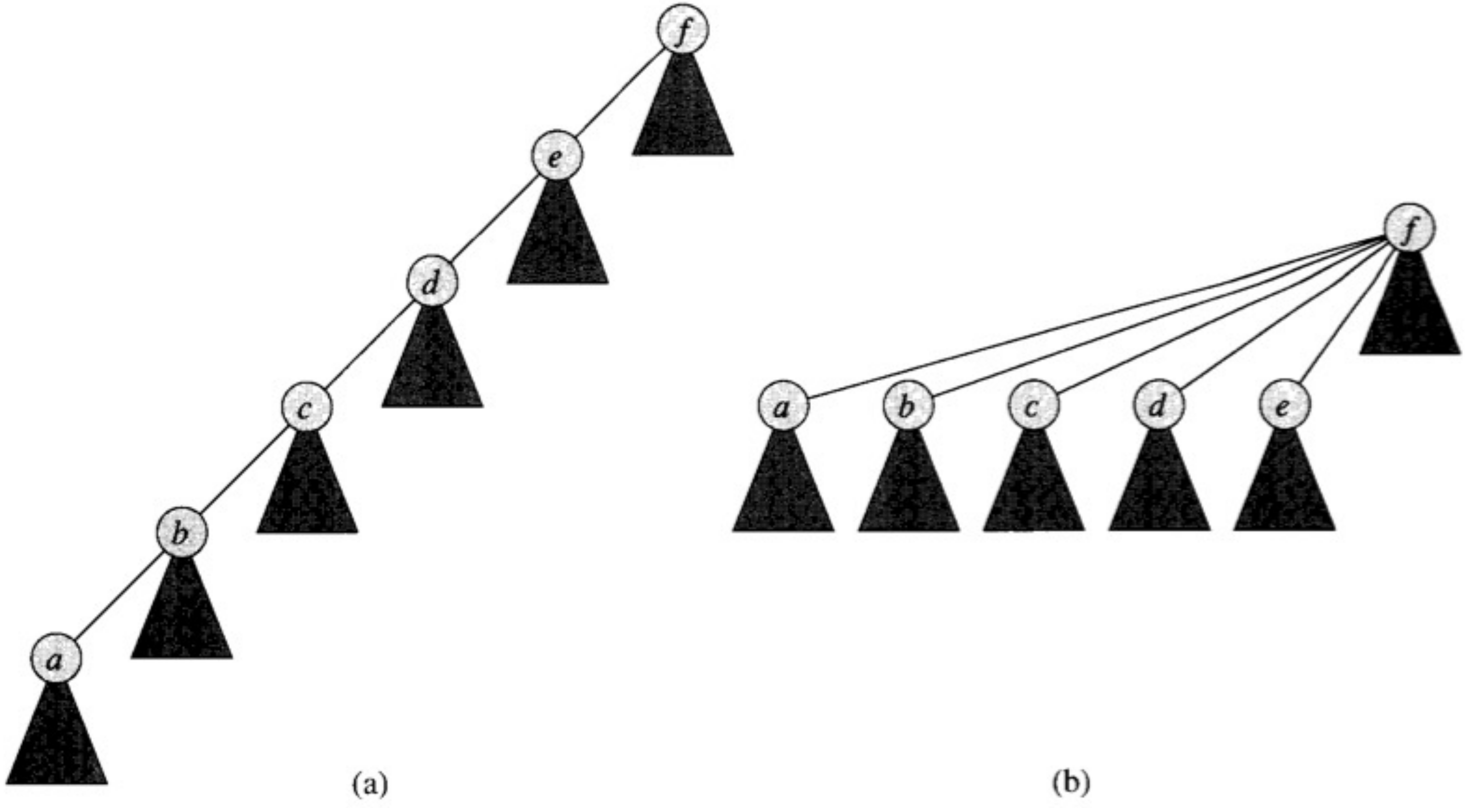
```
MAKE-SET(x)
    x.p = x
    x.rank = 0

UNION(x, y)
    LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
    if x.rank > y.rank
        y.p = x
    else x.p = y
        if x.rank == y.rank
            y.rank = y.rank + 1
```

Optimization 2: Path Compression

- Nothing special about tree's structure, as long as we can trace back to root
- **Idea:** as we do a find, each node we visit gets updated to point directly to root
- <u>Later</u> finds will be faster



Optimization 2: Path Compression

- Also easy to implement
  - CLRS code uses recursion →
  - Or would loop and keep a list

```
def find_set(x):
    path = []
    while x != x.p:
        path.append(x)
        x = x.p
    for n in path:
        n.p = x.p
    return x.p
```

FIND-SET(x)

if $x \neq x.p$
    $x.p = $ FIND-SET($x.p$)
return $x.p$

Complexity for Kruskal's

- Union-by-rank and path compression yields m operations in $\Theta\left(m * \alpha(n)\right)$
  - where $\alpha(n)$ a VERY slowly growing function. (See textbook for details)
  - m is the number of times you run the operation. So constant time, for each operation
- So overall Kruskal's with path compression:

  $\Theta(E * \log(V) + E * 1) = \Theta(E * \log(V))$ //now the heap is slowest part
- Originally:

  $\Theta(E * \log(V) + E * V) = \Theta(E * V) = \boldsymbol{O}\left(\boldsymbol{V^3}\right)$ *//Assumed find and union linear time*
  - (Time complexity if we'd sorted edges and not used a heap?)

Summary

What did we learn?

- Minimum Spanning Trees
  - Review!
- Kruskal’s Algorithm
  - Review again!
- Find-union
  - How to implement
  - How to optimize
  - How it affects runtime of Kruskal’s algorithm.