CS 3100
Data Structures and Algorithms 2
Lecture 13: Minimum Spanning Tree Algorithms

Co-instructors: Robbie Hott and Ray Pettit

Spring 2024

Readings in CLRS 4th edition:

- Chapter 21

Announcements

- PS5 due Tomorrow
- PA3 coming soon!
- Office hours
    - Prof Hott Office Hours: Mondays 11a-12p, Fridays 10-11a and 2-3p
    - Prof Pettit Office Hours: Mondays and Fridays 2:30-4:00p
    - TA office hours posted on our website
    - Office hours are not for "checking solutions"

Reminders about Greedy Algorithms

Reminder: Some Terminology

Optimization problems: terminology

- A solution must meet certain constraints:
  A solution is *feasible*

  Example: A possible shortest path must meet these criteria:
  All edges must be in the graph and form a simple path.
- Solutions judged on some criteria:

  Objective function

  Example: The sum of edge weights in path is minimum
- One (or more) feasible solutions that scores highest (by the objective function) is called the *optimal solution(s)*

The **greedy approach** is often a good choice for optimization problems

- So is **dynamic programming** (coming later in the course)

Reminder: Greedy Strategy: An Overview

Greedy strategy:

- Build solution by stages, adding one item to the partial solution we've found before this stage
- At each stage, make *locally optimal choice* based on the ***greedy choice*** (sometimes called the *greedy rule* or the *selection function)*
  - Locally optimal, i.e. best given what info we have now
- Irrevocable: a choice can't be un-done
- Sequence of locally optimal choices leads to globally optimal solution (hopefully)
  - Must prove this for a given problem!

Reminder: We’ve Seen Greedy Graph Algorithms

Dijkstra’s Shortest Path is greedy!

Build solution by adding item to partial solution

- Dijkstra’s: add edge to connect *k*th vertex, where the edges for the *k-1* already selected show the shortest paths to those *k-1* vertices

Greedy choice

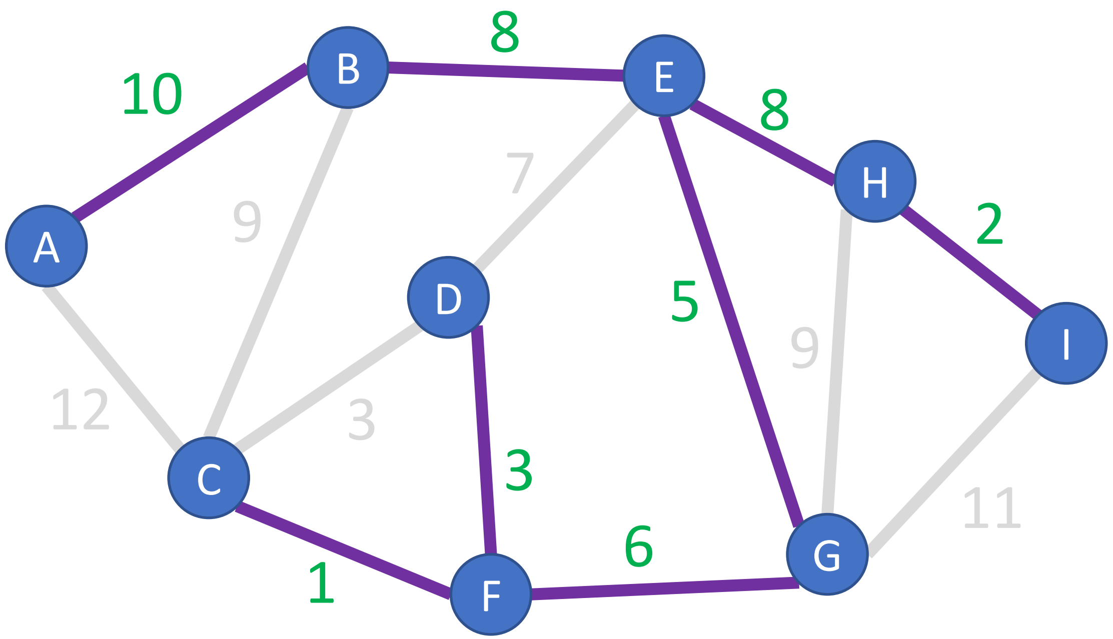
- Dijkstra’s: for all vertices connected to one of the *k-1* vertices already processed, choose *w* where *dist(s,w)* is the minimum

We did have to prove that this sequence of locally optimal choices leads to globally optimal solution

Minimum Spanning Trees

Readings: CLRS 21
(but not 21.1)

Spanning Tree



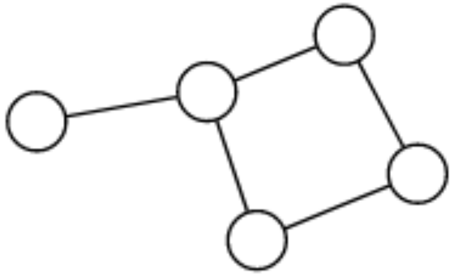
- All connected graphs have spanning tree(s)
- All spanning trees have the same number of nodes (all of them)
- You can construct a spanning tree by arbitrarily remove edges from cycles

How many edges does T have?

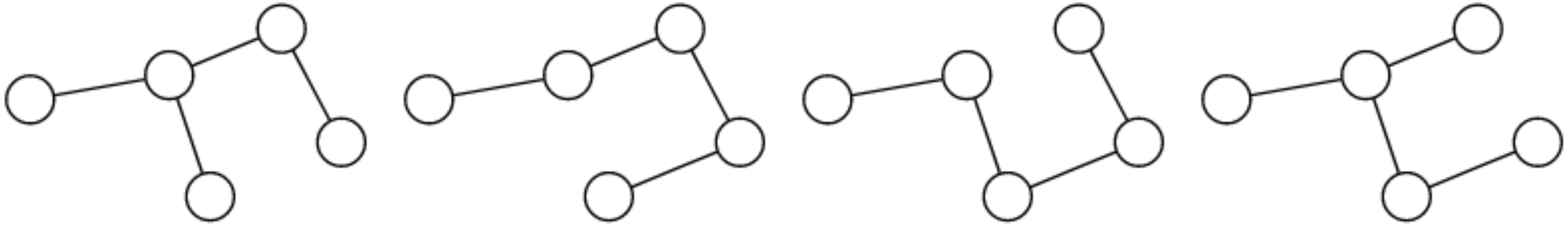
A tree $T = (V_T, E_T)$ is a **spanning tree** for an undirected graph $G = (V, E)$ if $V_T = V$, $E_T \subseteq E$ (namely, T connects or "spans" all the nodes in G)

Spanning Tree: Example

Original Graph:



Possible spanning trees:



Minimum Spanning Tree

Just constructing any spanning tree is simple

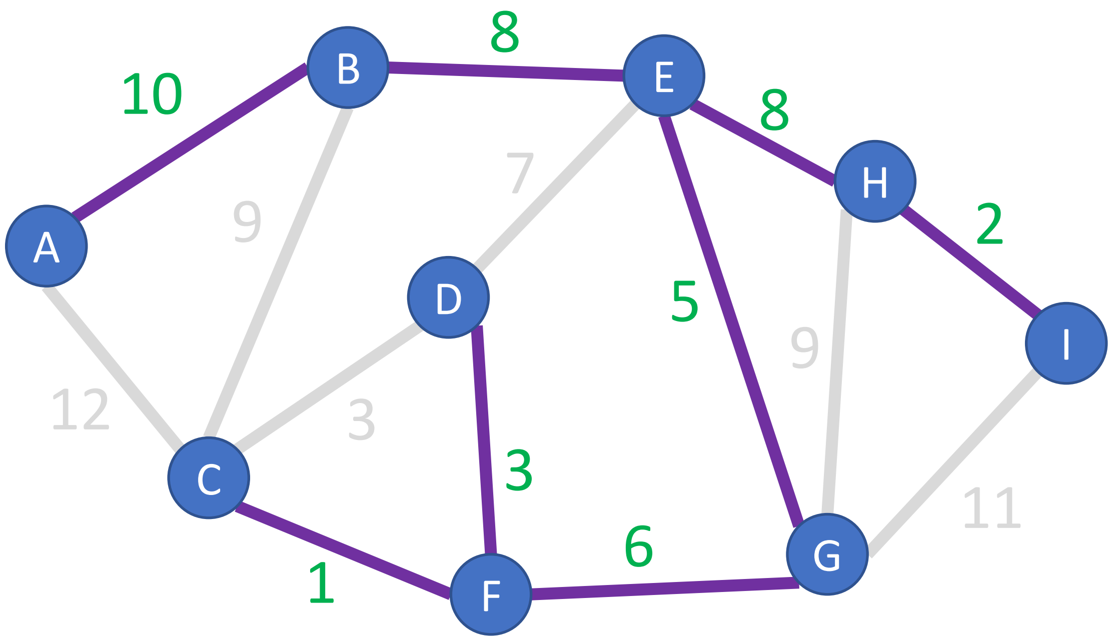
Suppose edges have weights

- Cost of building tracks between two stations
- Length of wire between boxes in a house
- Cheapest way to connect all nodes in some kind of network

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



$$\text{Cost}(T) = \sum_{e \in E_T} w(e)$$

How many edges does T have?

A tree $T = (V_T, E_T)$ is a **minimum spanning tree** for an undirected graph $G = (V, E)$ if T is a spanning tree of minimal cost

MST Algorithms

We’ll see two greedy algorithms to find a graph’s MST

- Prim’s algorithm
  - Very similar to Dijkstra’s SP algorithm
  - Builds a single tree, adding one edge to grow the tree
- Kruskal’s algorithm
  - In a *forest* of trees, add an edge at each step to grow one tree or to connect two trees (don’t make a cycle)
  - Utilizes an interesting data structure for manipulating sets

Prim’s Algorithm

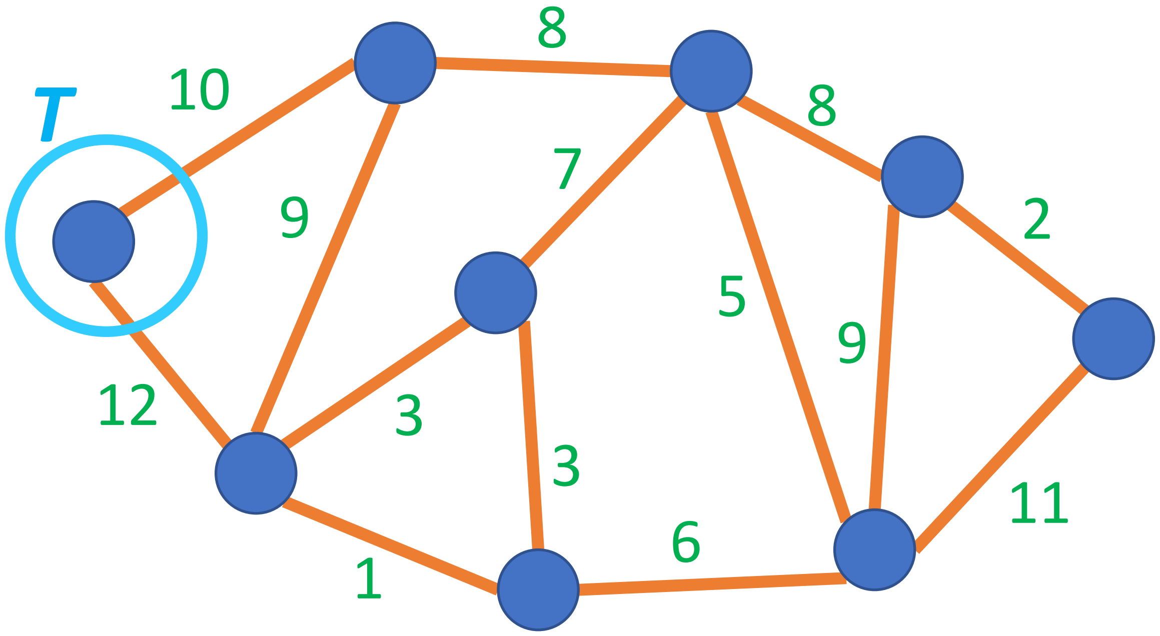
CLRS in 21.2

Reminder: Dijkstra’s SP Algorithm

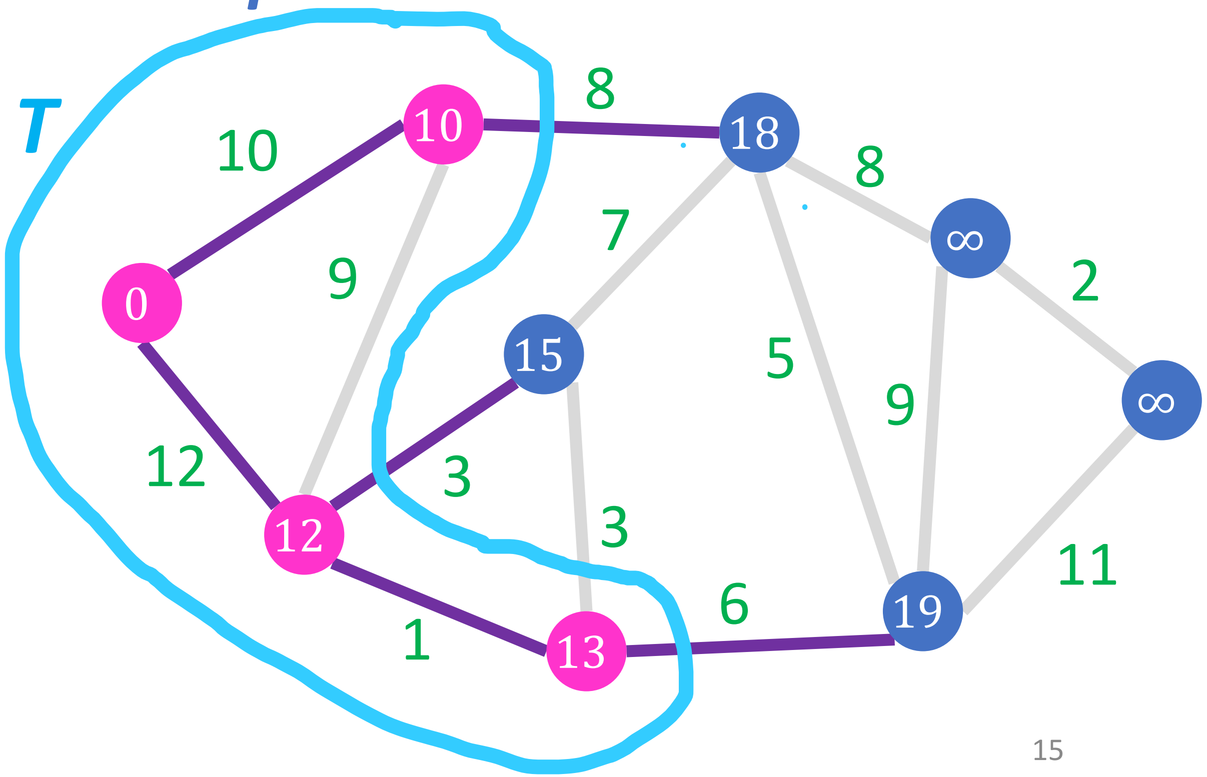
1. Start with an empty tree T and add the source to T
2. Repeat $|V| - 1$ times:
   - At each step, add the node **“nearest” to the source into tree T**

Greedy Choice Property!

Initially:



At some point later:

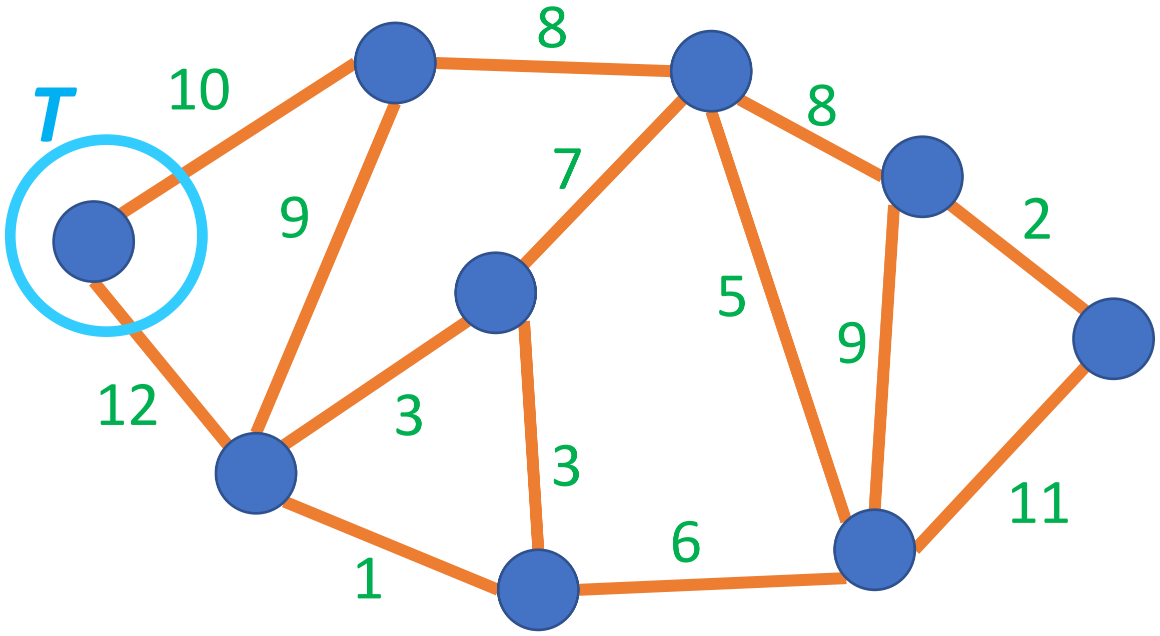


Prim's MST Algorithm

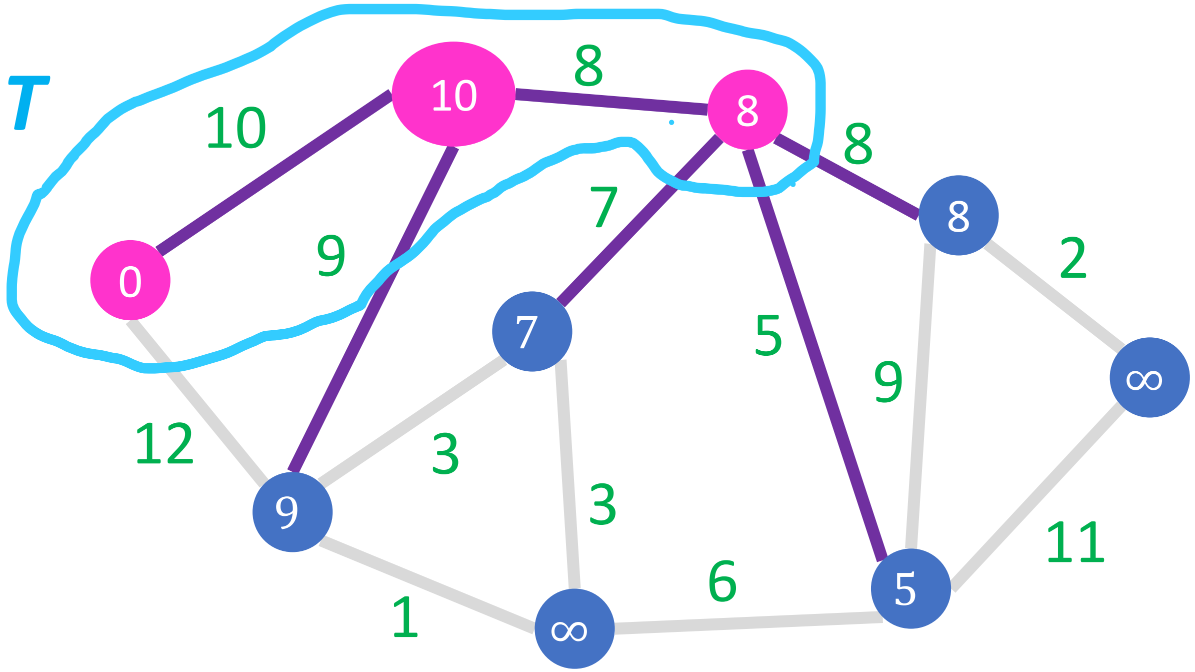
The *Greedy Choice*! Same strategy, but different greedy choice to solve a different problem

1. Start with an empty tree T and add the source to T
2. Repeat $|V| - 1$ times:
   - At each step, add the node with **minimum connecting edge to a node in T**

Initially:

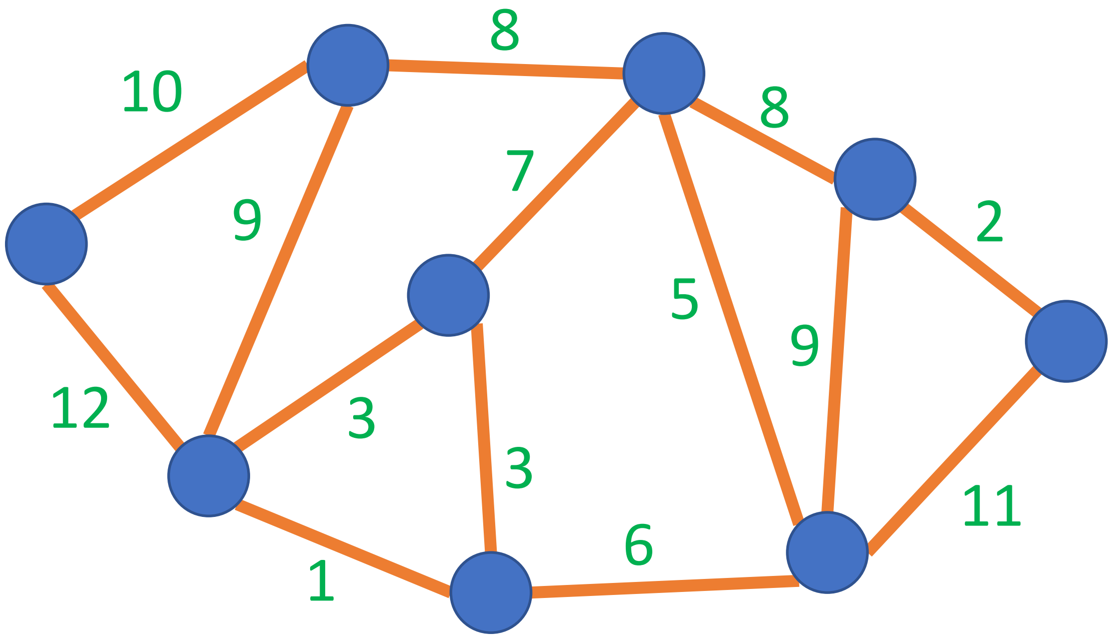


At some point later:



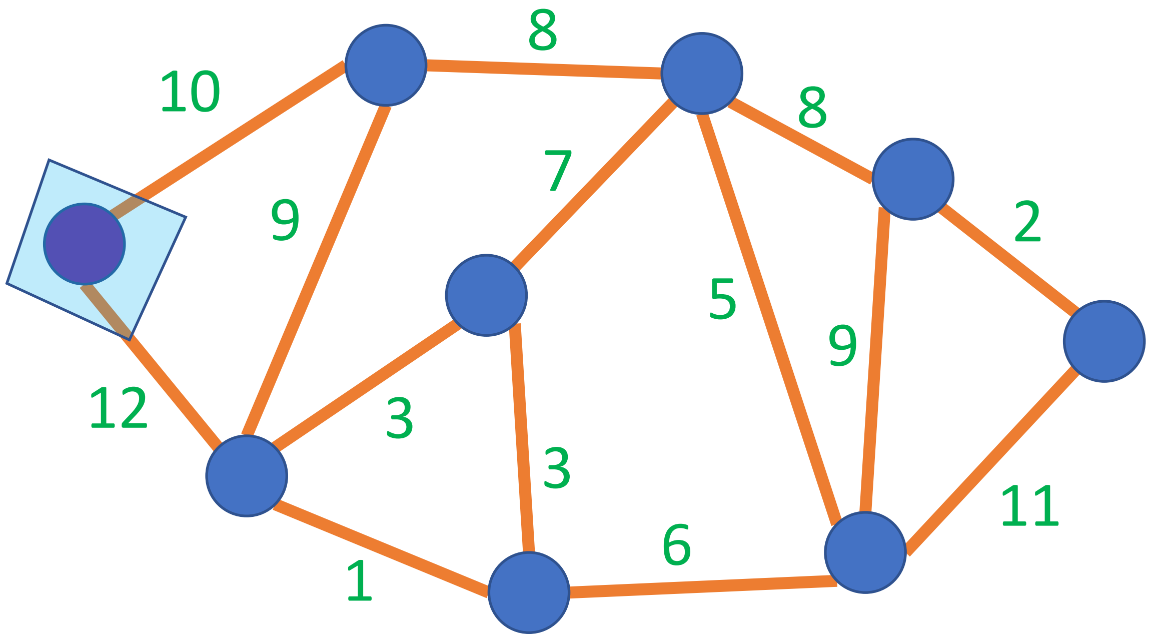
Prim's Algorithm

1. Start with an empty tree T and pick a start node and add it to T
2. Repeat $|V| - 1$ times:
   - Add the min-weight edge which connects a node in T with a node not in T



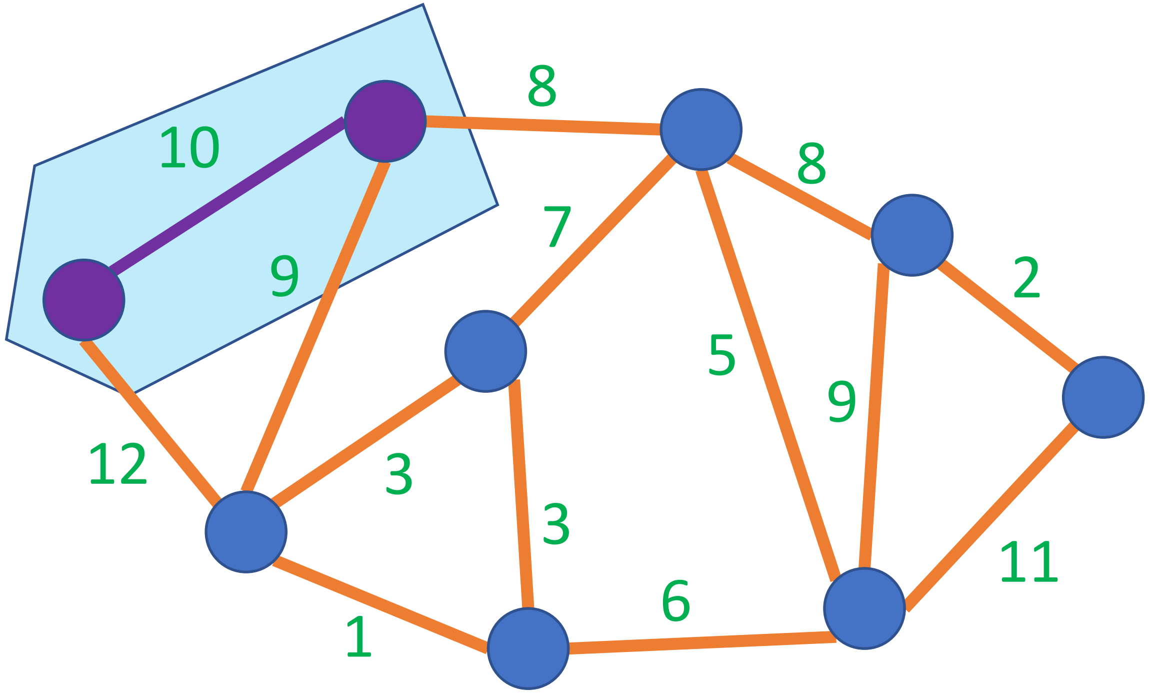
Prim’s Algorithm

1. Start with an empty tree T and pick a start node and add it to T
2. Repeat $|V| - 1$ times:
   - Add the min-weight edge which connects a node in T with a node not in T



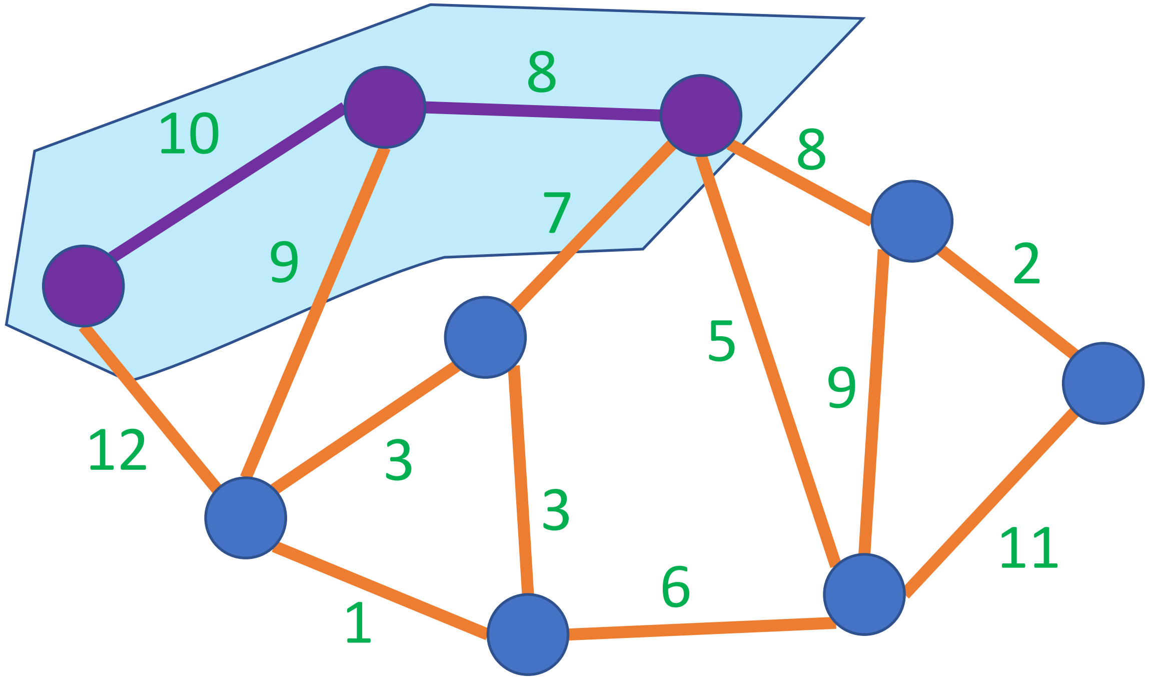
Prim’s Algorithm

1. Start with an empty tree T and pick a start node and add it to T
2. Repeat $|V| - 1$ times:
   - Add the min-weight edge which connects a node in T with a node not in T



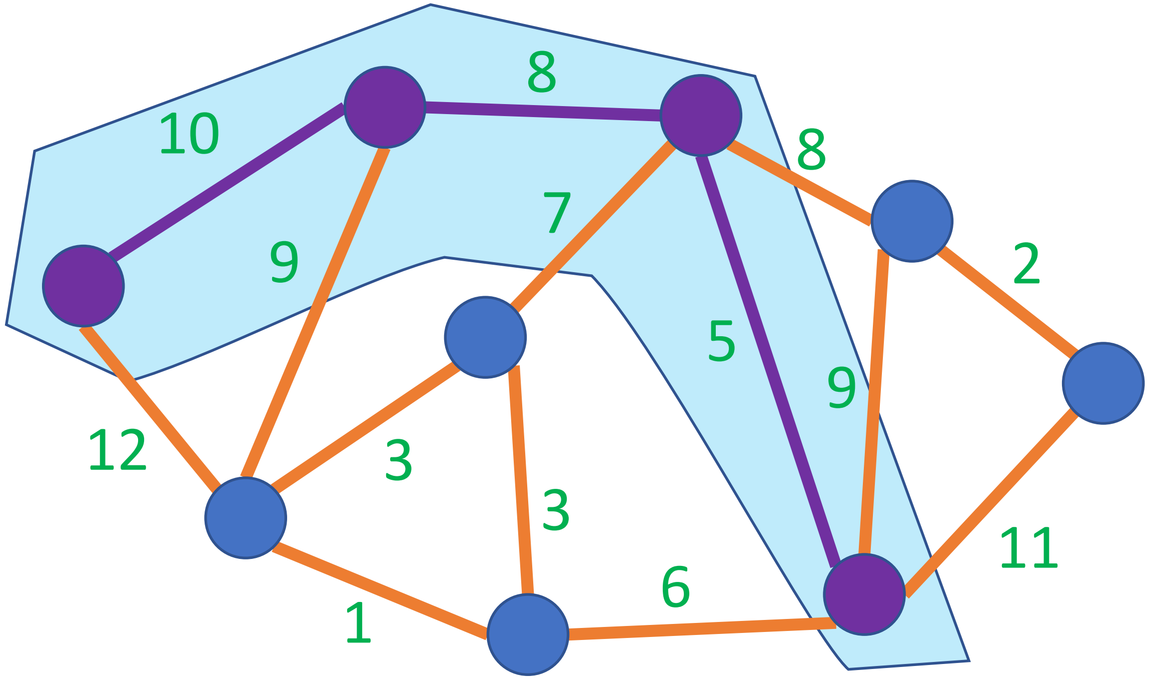
Prim’s Algorithm

1. Start with an empty tree T and pick a start node and add it to T
2. Repeat $|V| - 1$ times:
   - Add the min-weight edge which connects a node in T with a node not in T



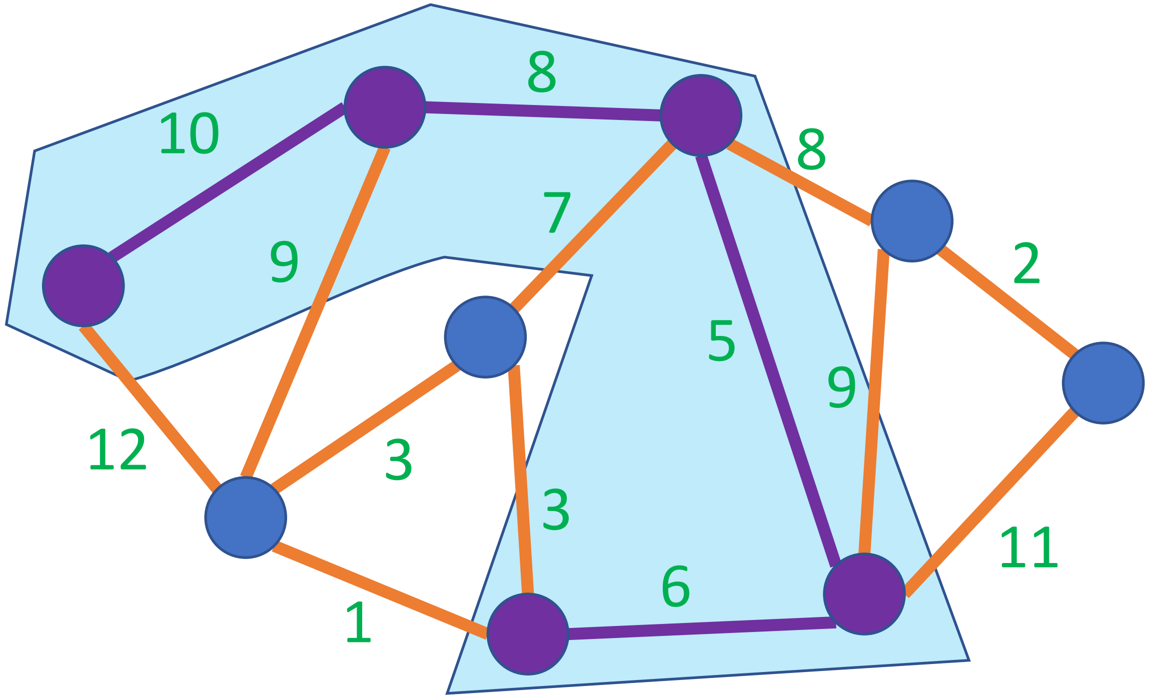
Prim's Algorithm

1. Start with an empty tree T and pick a start node and add it to T
2. Repeat $|V| - 1$ times:
    - Add the min-weight edge which connects a node in T with a node not in T



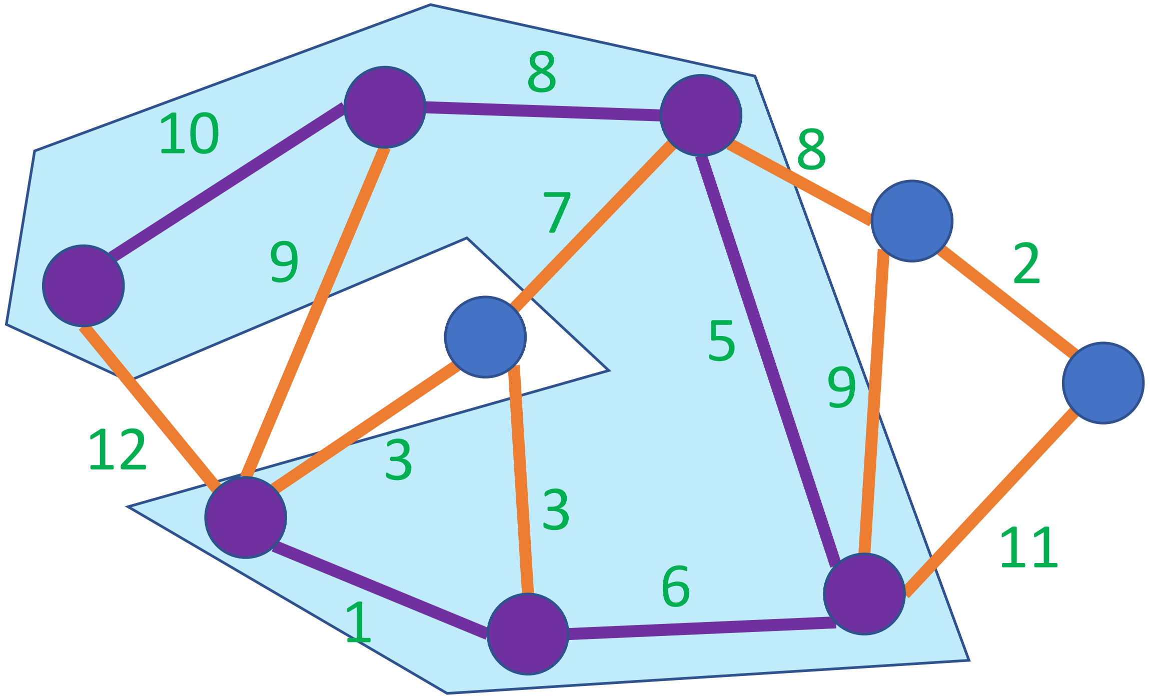
Prim’s Algorithm

1. Start with an empty tree T and pick a start node and add it to T
2. Repeat $|V| - 1$ times:
    - Add the min-weight edge which connects a node in T with a node not in T



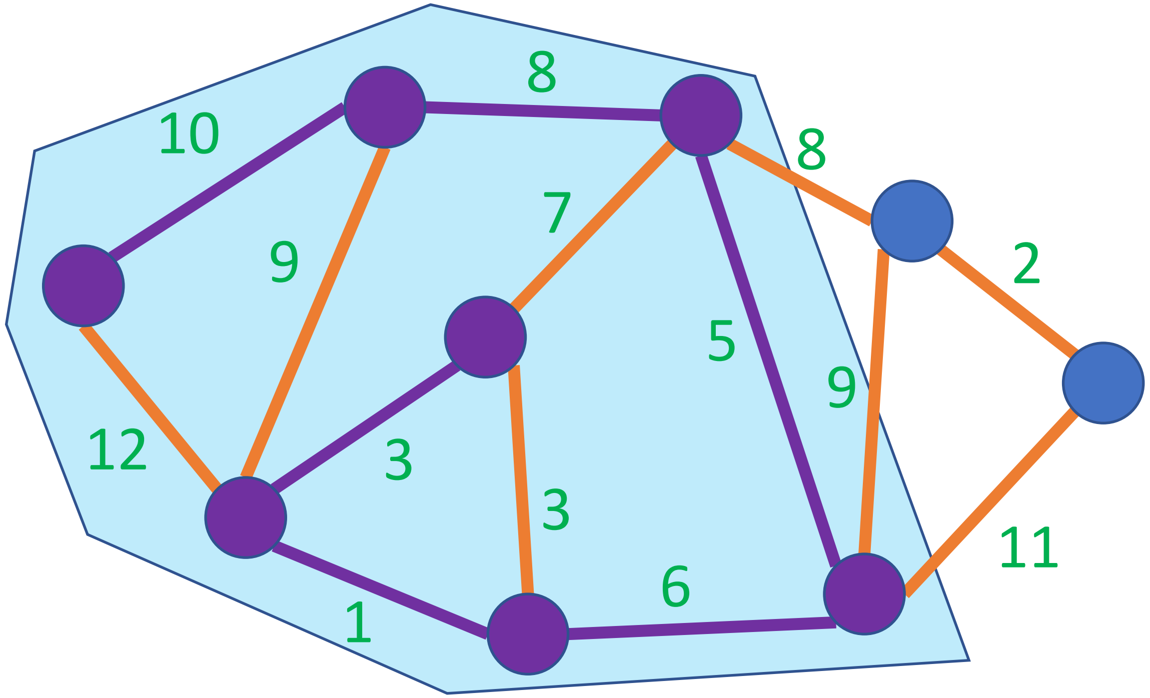
Prim’s Algorithm

1. Start with an empty tree T and pick a start node and add it to T
2. Repeat $|V| - 1$ times:
   - Add the min-weight edge which connects a node in T with a node not in T



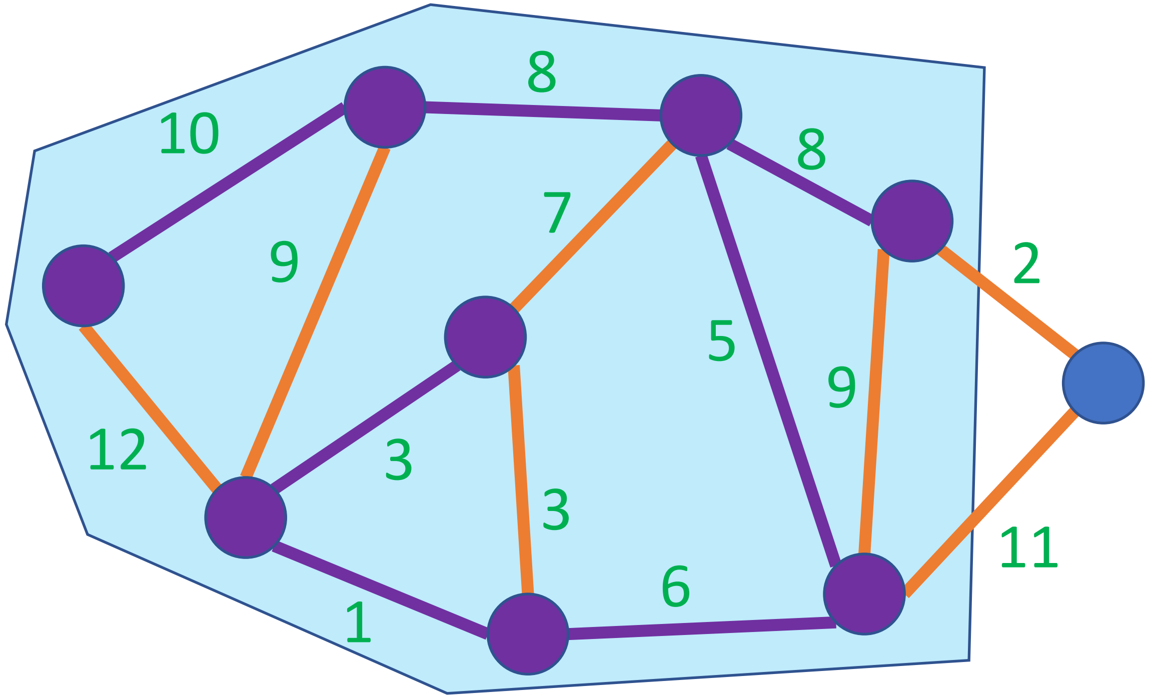
Prim's Algorithm

1. Start with an empty tree T and pick a start node and add it to T
2. Repeat $|V| - 1$ times:
   - Add the min-weight edge which connects a node in T with a node not in T



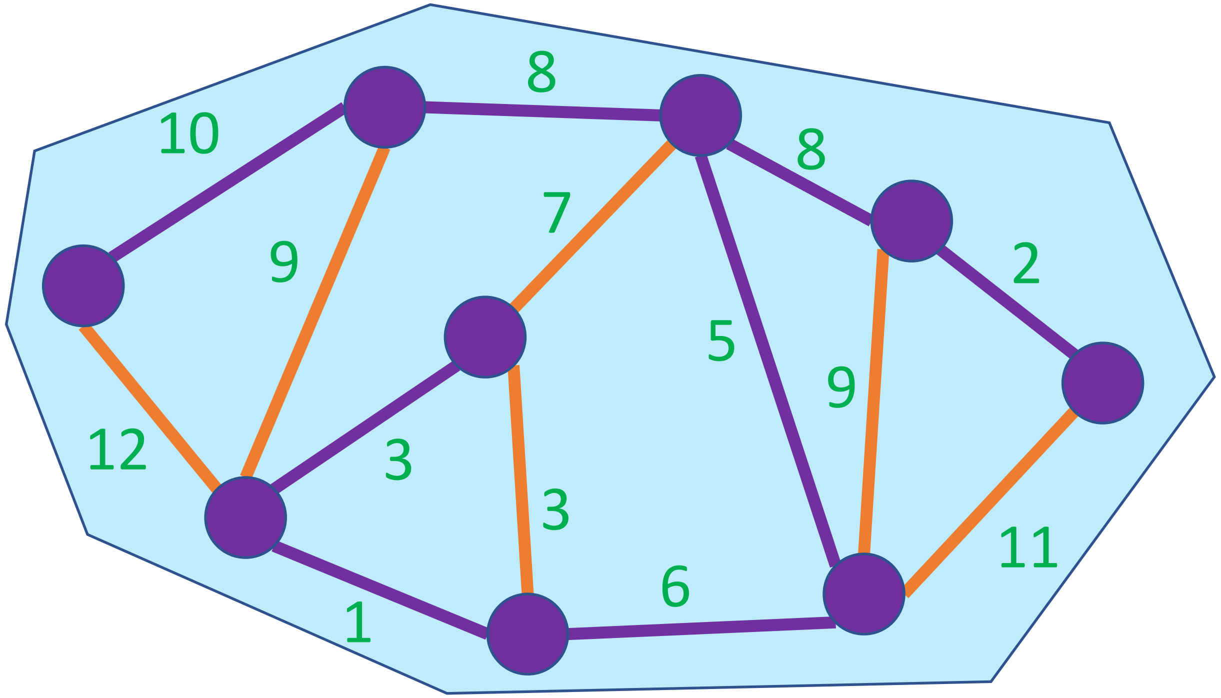
Prim's Algorithm

1. Start with an empty tree T and pick a start node and add it to T
2. Repeat $|V| - 1$ times:
   - Add the min-weight edge which connects a node in T with a node not in T



Prim’s Algorithm

1. Start with an empty tree T and pick a start node and add it to T
2. Repeat $|V| - 1$ times:
    - Add the min-weight edge which connects a node in T with a node not in T



Prim's Algorithm

1. Start with an empty tree T and pick a start node and add it to T
2. Repeat $|V| - 1$ times:
   - Add the min-weight edge which connects a node in T with a node not in T

Implementation:

- Maintain nodes **not in** T in a min-heap (priority queue)
- Find the next closest node v by extracting min from priority queue
- Each time node v is added to the tree, update keys for neighbors still in min-heap
- Repeat until no nodes left in min-heap

Prim's Algorithm Implementation

1. Start with an empty tree T and pick a start node and add it to T
2. Repeat $|V| - 1$ times:
   - Add the min-weight edge which connects a node in T with a node not in T

Implementation:

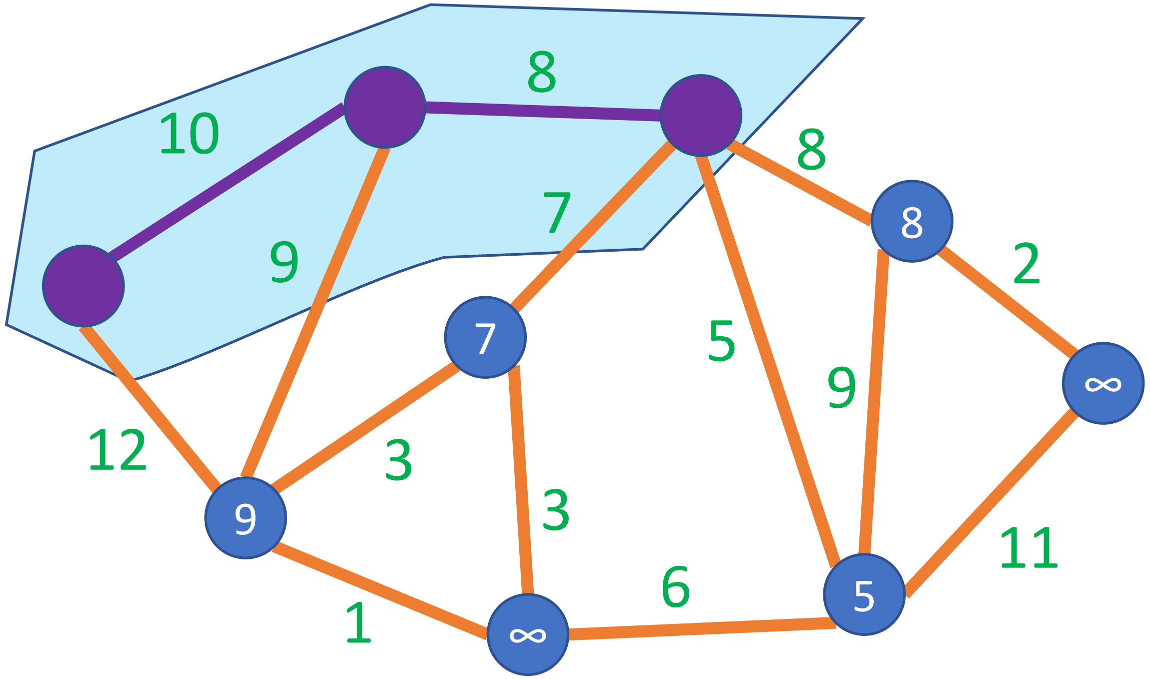
```
    initialize $d_v = \infty$ for each node $v$
    add all nodes $v \in V$ to the priority queue $\mathrm{PQ}$, using $d_v$ as the key
    pick a starting node $s$ and set $d_s = 0$
    while $\mathrm{PQ}$ is not empty:
        $v = \mathrm{PQ.extractMin}()$
        for each $u \in V$ such that $(v, u) \in E$:
            if $u \in \mathrm{PQ}$ and $w(v, u) < d_u$:
                $\mathrm{PQ.decreaseKey}\big(u, w(v, u)\big)$
                $u.\mathrm{parent} = v$
```

each node also maintains a parent, initially NULL

key: minimum cost to connect u to nodes in PQ

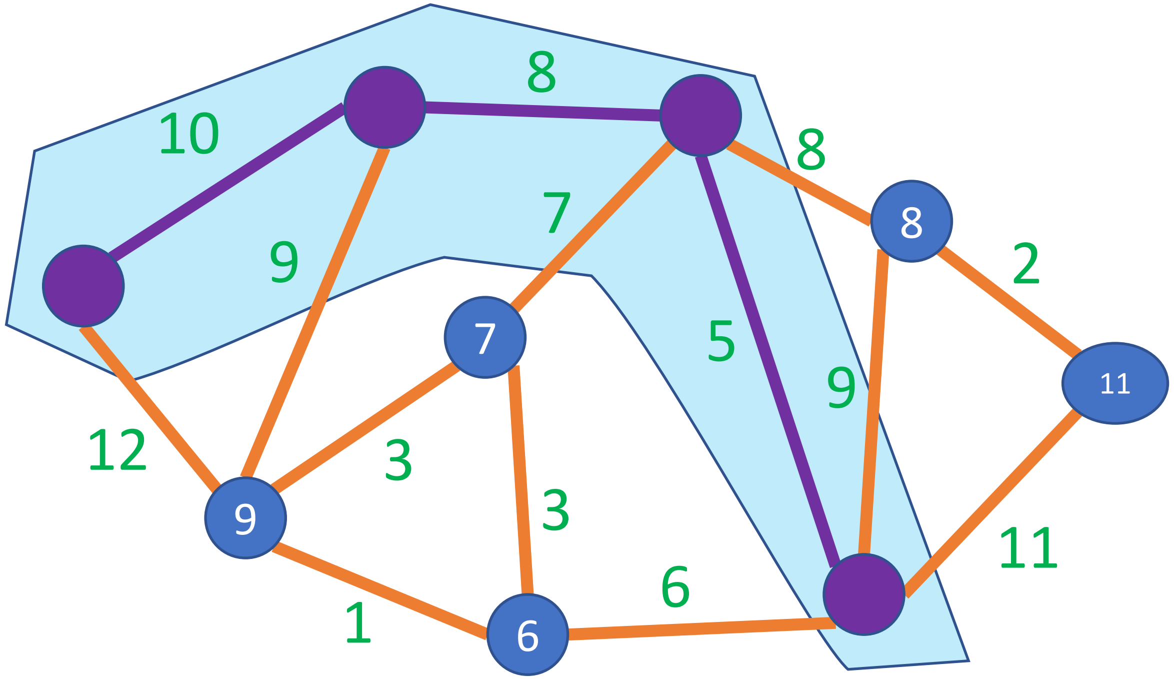
Prim's Algorithm

1. Start with an empty tree T and pick a start node and add it to T
2. Repeat $|V| - 1$ times:
   - Add the min-weight edge which connects a node in T with a node not in T



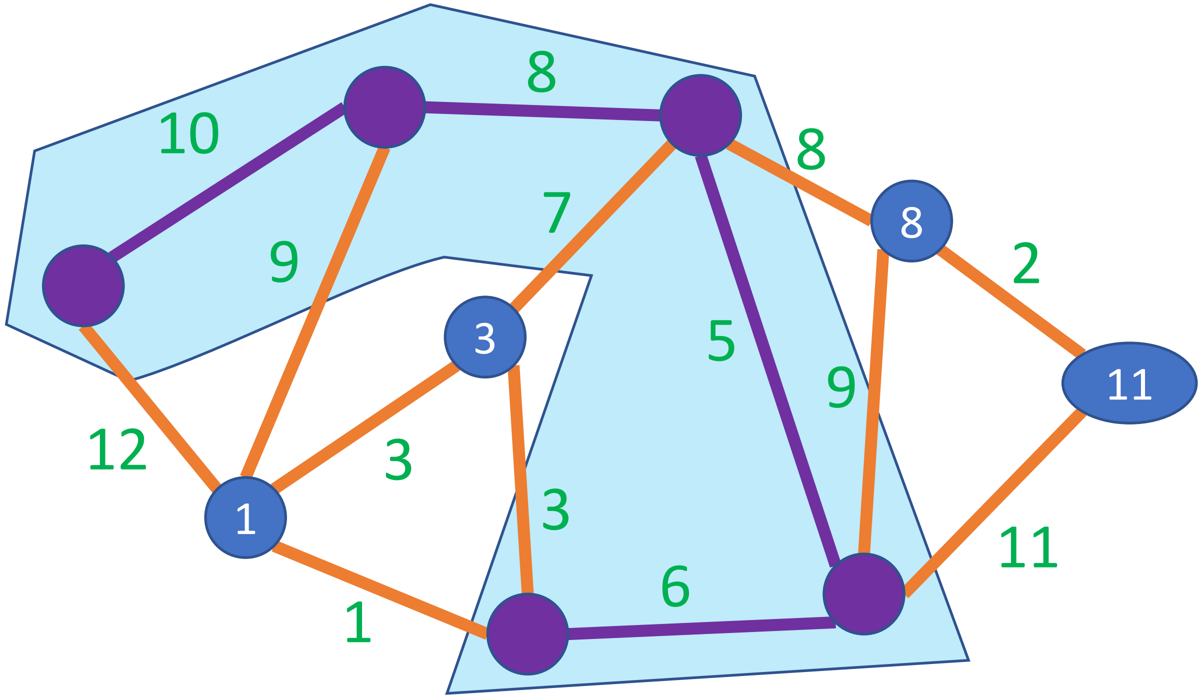
Prim's Algorithm

1. Start with an empty tree T and pick a start node and add it to T
2. Repeat $|V| - 1$ times:
   - Add the min-weight edge which connects a node in T with a node not in T



Prim’s Algorithm

1. Start with an empty tree T and pick a start node and add it to T
2. Repeat $|V| - 1$ times:
   - Add the min-weight edge which connects a node in T with a node not in T



Reminder: Dijkstra's Algorithm Implementation

1. Start with an empty tree T and add the source to T
2. Repeat $|V| - 1$ times:
   - Add the "nearest" node not yet in T to T

Implementation:

```
initialize $d_v = \infty$ for each node $v$
add all nodes $v \in V$ to the priority queue $\mathrm{PQ}$, using $d_v$ as the key
set $d_s = 0$
while $\mathrm{PQ}$ is not empty:
    $v = \mathrm{PQ.extractMin}()$
    for each $u \in V$ such that $(v, u) \in E$:
        if $u \in \mathrm{PQ}$ and $d_v + w(v, u) < d_u$:
            $\mathrm{PQ.decreaseKey}(u, d_v + w(v, u))$
            $u.\mathrm{parent} = v$
```

each node also maintains a parent, initially `NULL`

key: length of shortest path $s \to u$ using nodes in PQ

Prim's Algorithm Implementation

1. Start with an empty tree T and pick a start node and add it to T
2. Repeat $|V| - 1$ times:
    - Add the min-weight edge which connects a node in T with a node not in T

Implementation:

```
    initialize $d_v = \infty$ for each node $v$
    add all nodes $v \in V$ to the priority queue $\mathrm{PQ}$, using $d_v$ as the key
    pick a starting node $s$ and set $d_s = 0$
    while $\mathrm{PQ}$ is not empty:
        $v = \mathrm{PQ}.\mathrm{extractMin}()$
        for each $u \in V$ such that $(v, u) \in E$:
            if $u \in \mathrm{PQ}$ and $w(v, u) < d_u$:
                $\mathrm{PQ}.\mathrm{decreaseKey}(u, w(v, u))$
                $u.\mathrm{parent} = v$
```

each node also maintains a parent, initially `NULL`

key: minimum cost to connect u to nodes in PQ

Prim’s Algorithm Running Time

Same as for Dijkstra’s Shortest Path algorithm!

Implementation (with nodes in the priority queue):

```
initialize $d_v = \infty$ for each node $v$                              Initialization:
add all nodes $v \in V$ to the priority queue $\mathrm{PQ}$, using $d_v$ as the key     $O(|V|)$
pick a starting node $s$ and set $d_s = 0$
while $\mathrm{PQ}$ is not empty:                                        $|V|$ iterations
    $v = \mathrm{PQ.extractMin}()$                                       $O(\log|V|)$
    for each $u \in V$ such that $(v, u) \in E$:                         $|E|$ iterations total
        if $u \in \mathrm{PQ}$ and $w(v, u) < d_u$:
            $\mathrm{PQ.decreaseKey}(u, w(v, u))$                        $O(\log|V|)$
            $u.\mathrm{parent} = v$
```

Using indirect heaps

Overall running time: $O(|V|\log|V| + |E|\log|V|) = O(|E|\log|V|)$

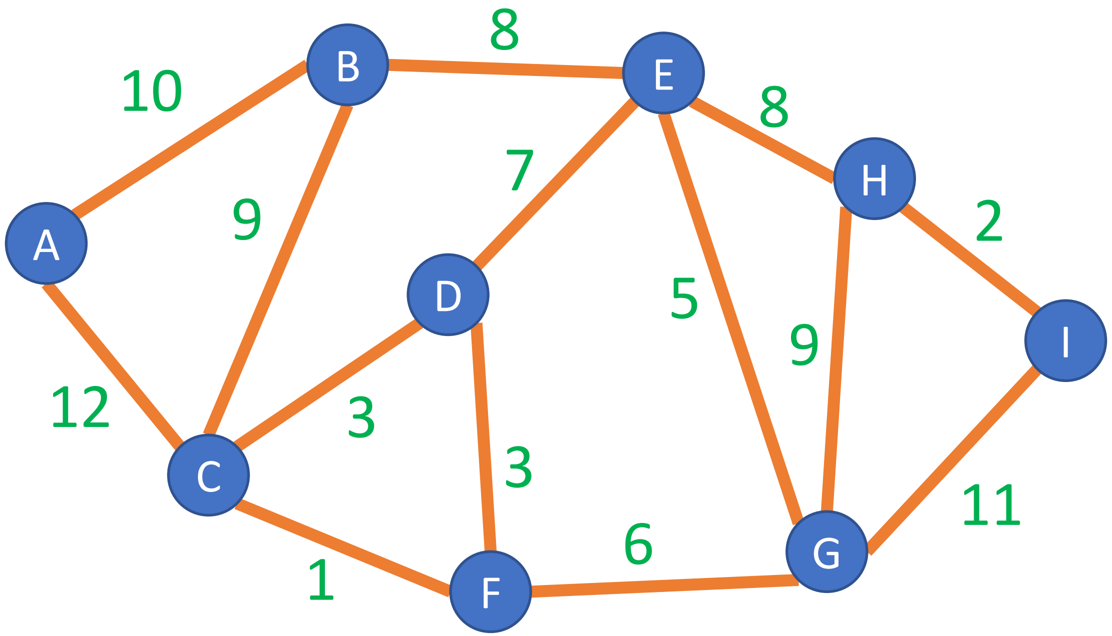
Kruskal’s MST Algorithm

Readings: CLRS first part of 21.2

Kruskal's Algorithm

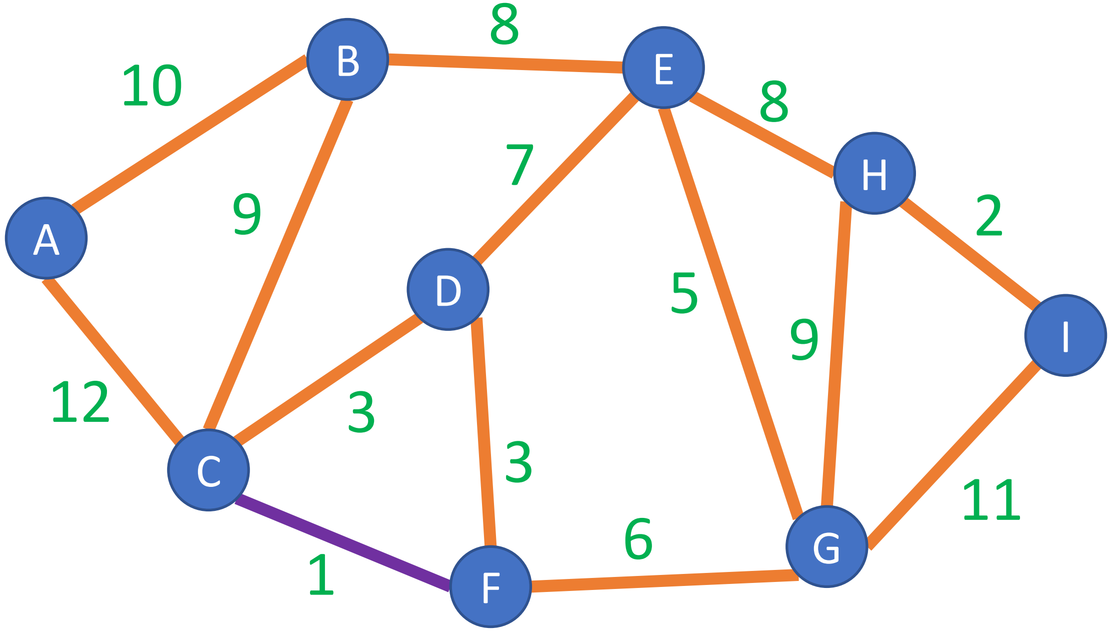
The *Greedy Choice* for Kruskal's

1. Start with an empty set of edges T
2. Repeatedly add to T the lowest-weight edge that does not create a cycle. (Stop when we've added $n - 1$ edges.)



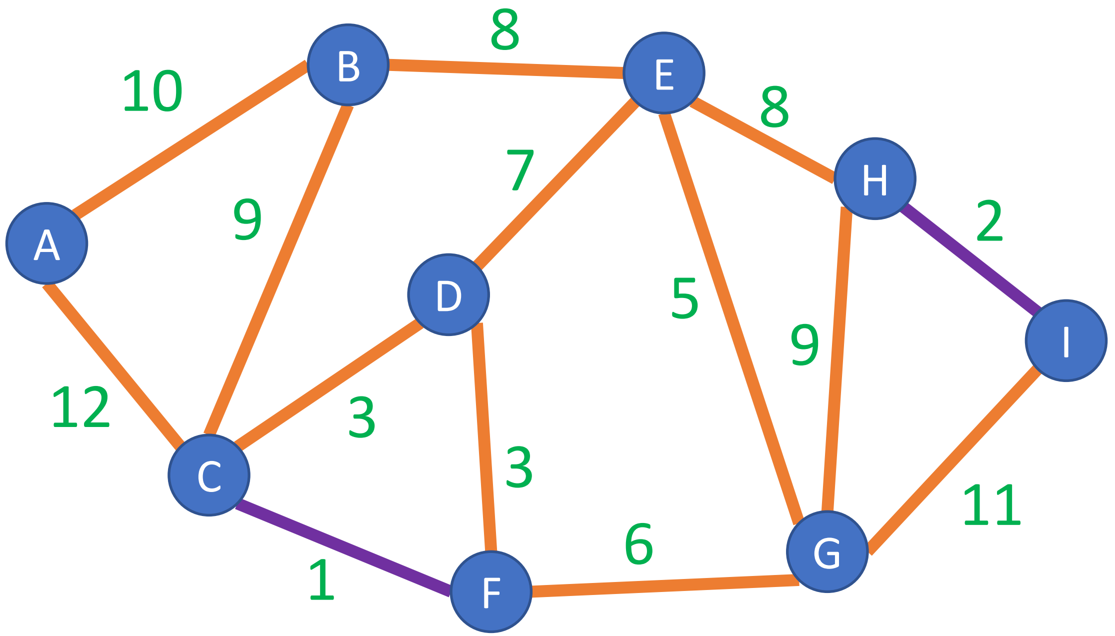
Kruskal's Algorithm

1. Start with an empty set of edges T
2. Repeatedly add to T the lowest-weight edge that does not create a cycle. (Stop when we've added $n - 1$ edges.)



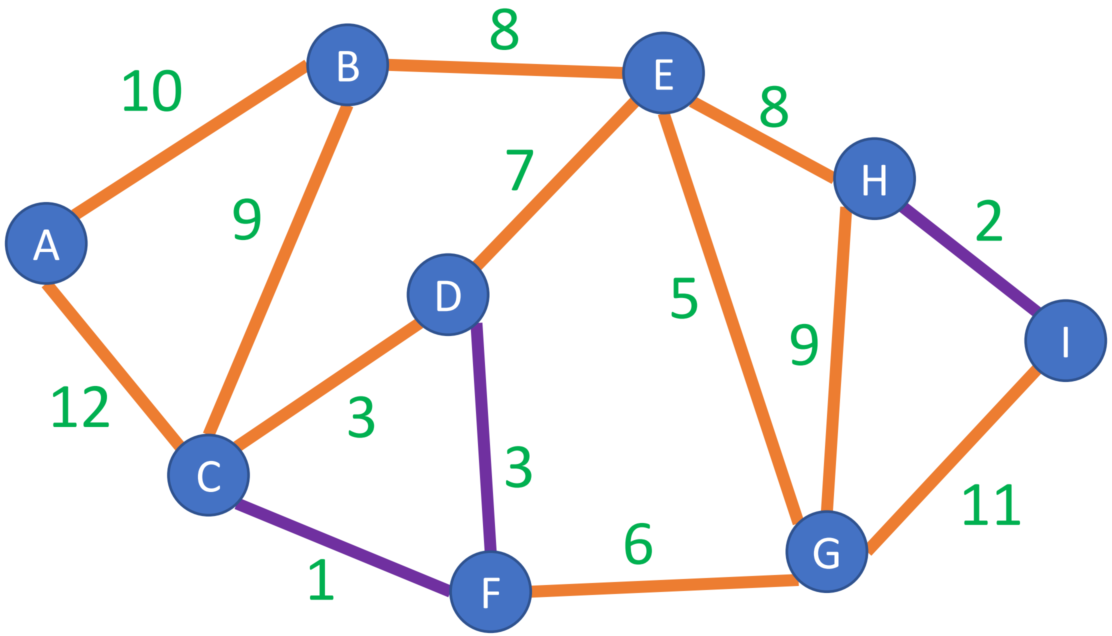
Kruskal's Algorithm

1. Start with an empty set of edges T
2. Repeatedly add to T the lowest-weight edge that does not create a cycle. (Stop when we've added $n - 1$ edges.)



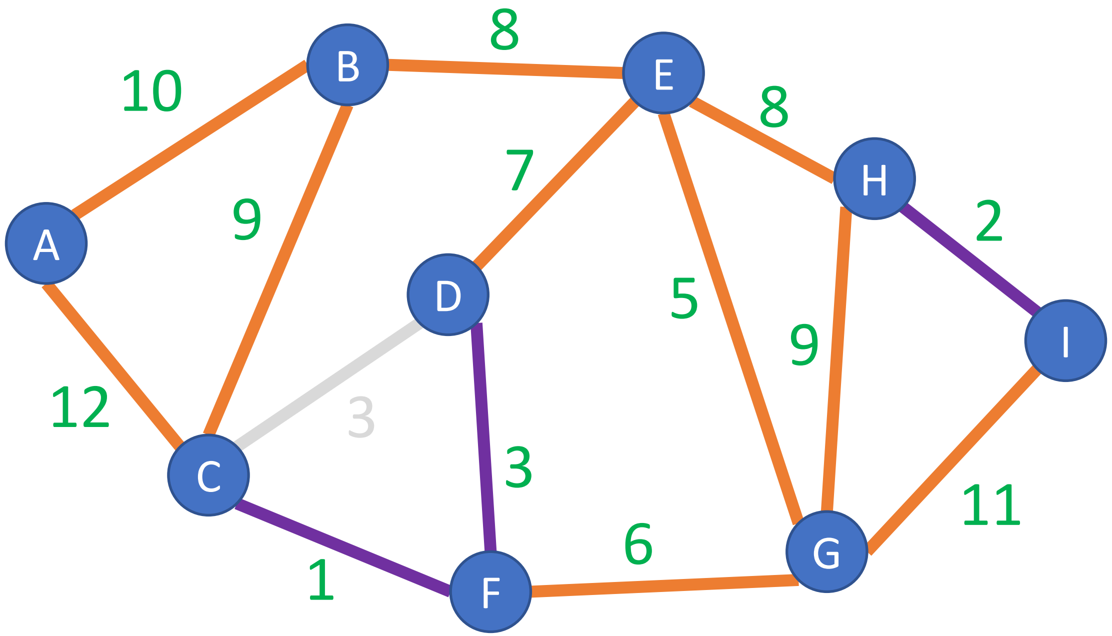
Kruskal’s Algorithm

1. Start with an empty set of edges T
2. Repeatedly add to T the lowest-weight edge that does not create a cycle. (Stop when we’ve added $n - 1$ edges.)



Kruskal's Algorithm

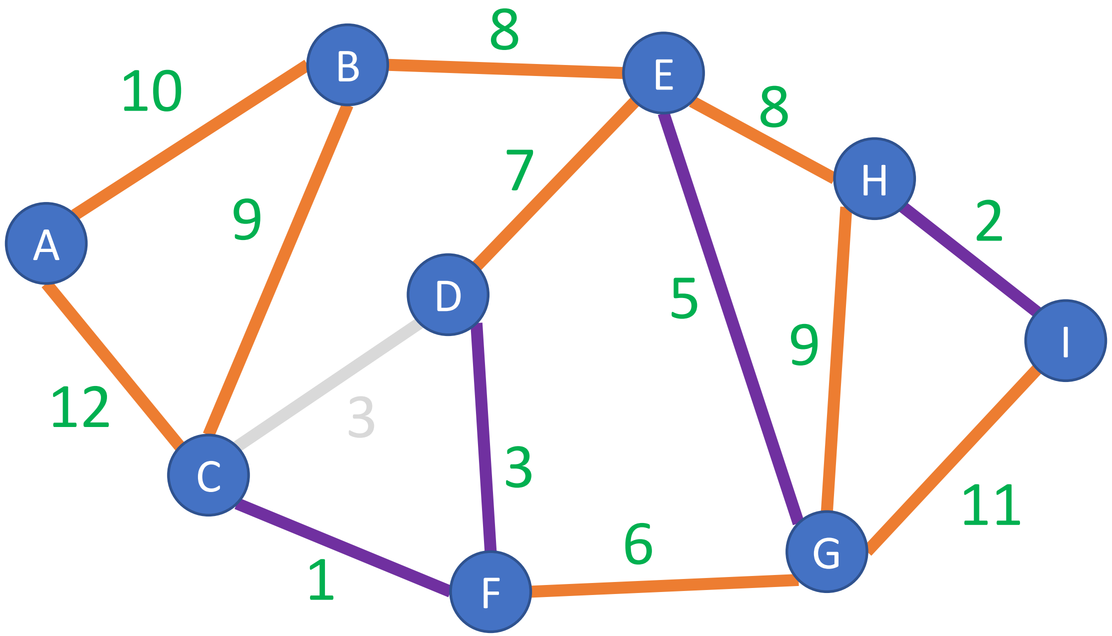
1. Start with an empty set of edges T
2. Repeatedly add to T the lowest-weight edge that does not create a cycle. (Stop when we've added $n-1$ edges.)



Edge forms a cycle, so do not include

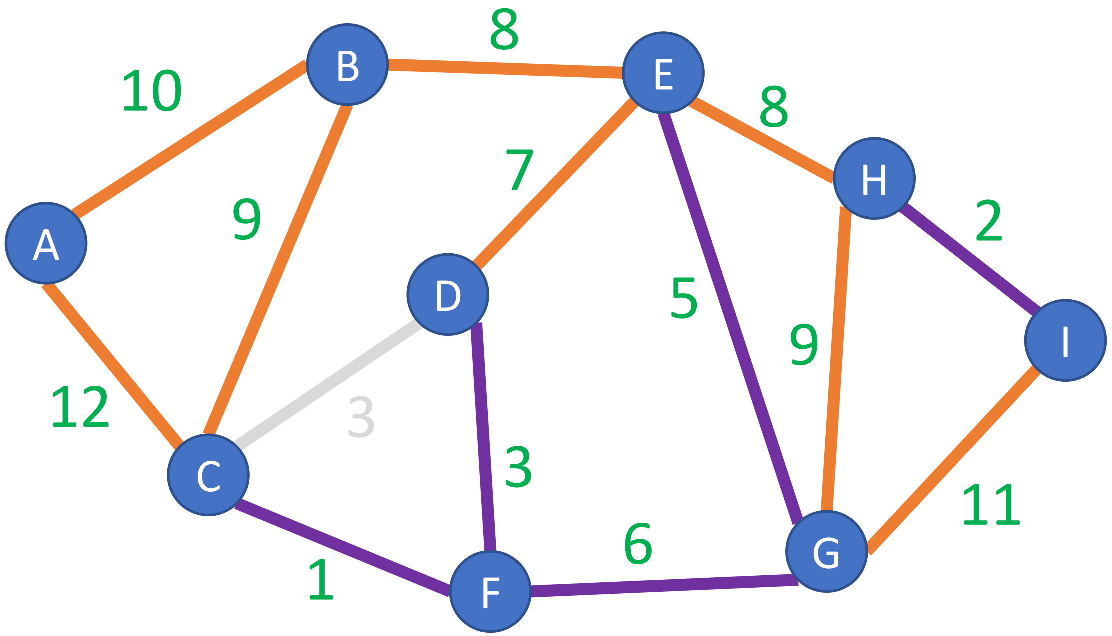
Kruskal's Algorithm

1. Start with an empty set of edges T
2. Repeatedly add to T the lowest-weight edge that does not create a cycle. (Stop when we've added $n - 1$ edges.)



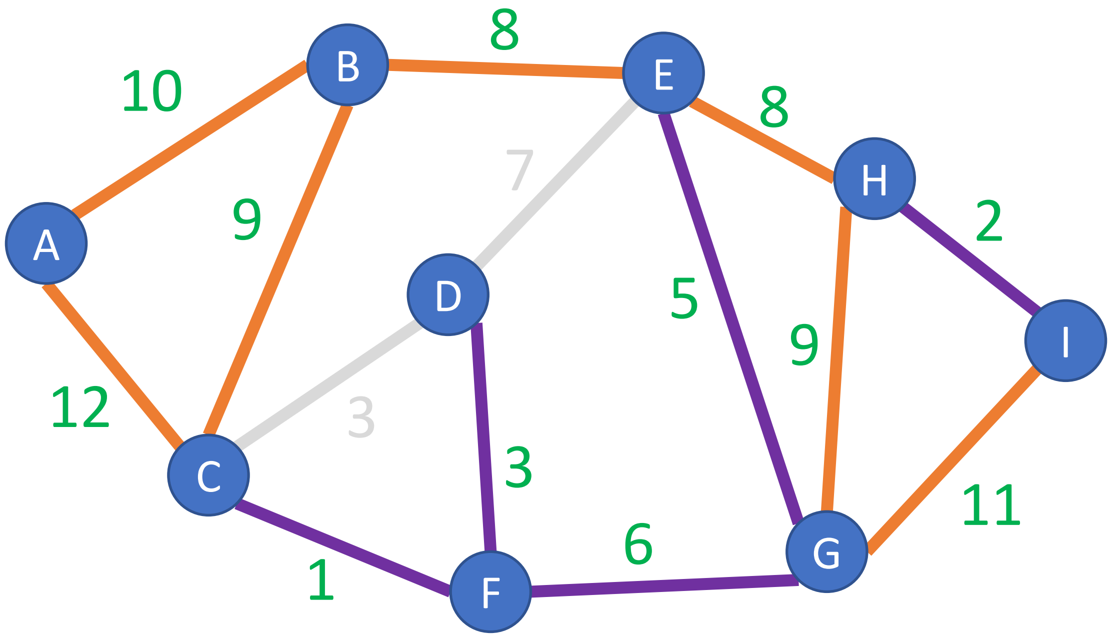
Kruskal's Algorithm

1. Start with an empty set of edges T
2. Repeatedly add to T the lowest-weight edge that does not create a cycle. (Stop when we've added $n - 1$ edges.)



Kruskal’s Algorithm

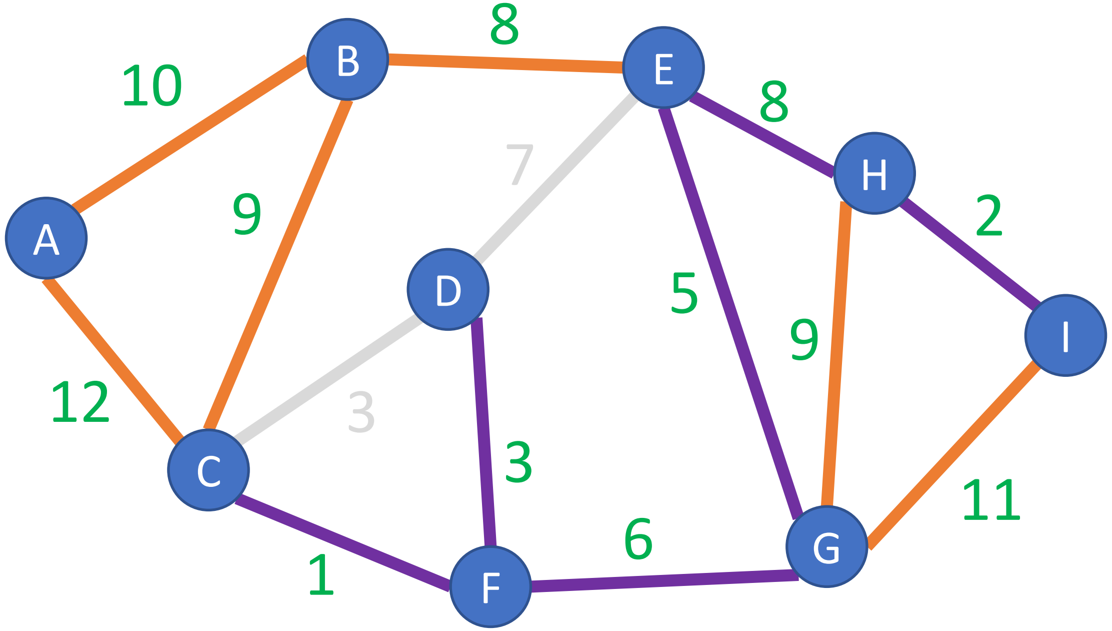
1. Start with an empty set of edges T
2. Repeatedly add to T the lowest-weight edge that does not create a cycle. (Stop when we’ve added $n - 1$ edges.)



Edge forms a cycle, so do not include

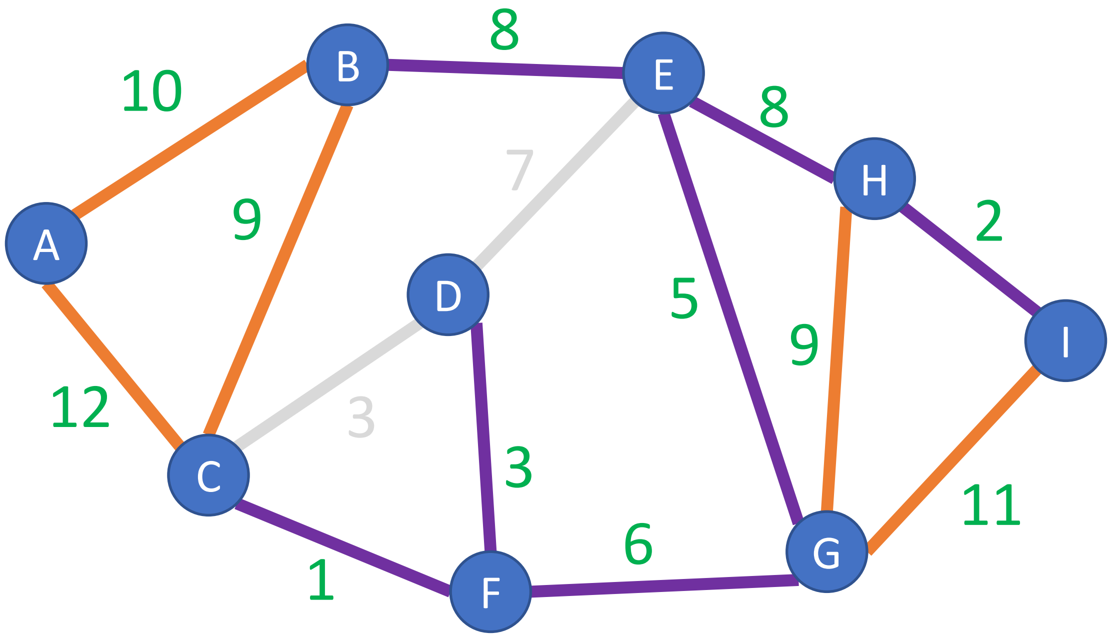
Kruskal's Algorithm

1. Start with an empty set of edges T
2. Repeatedly add to T the <u>lowest-weight</u> edge that does not create a cycle. (Stop when we've added $n - 1$ edges.)



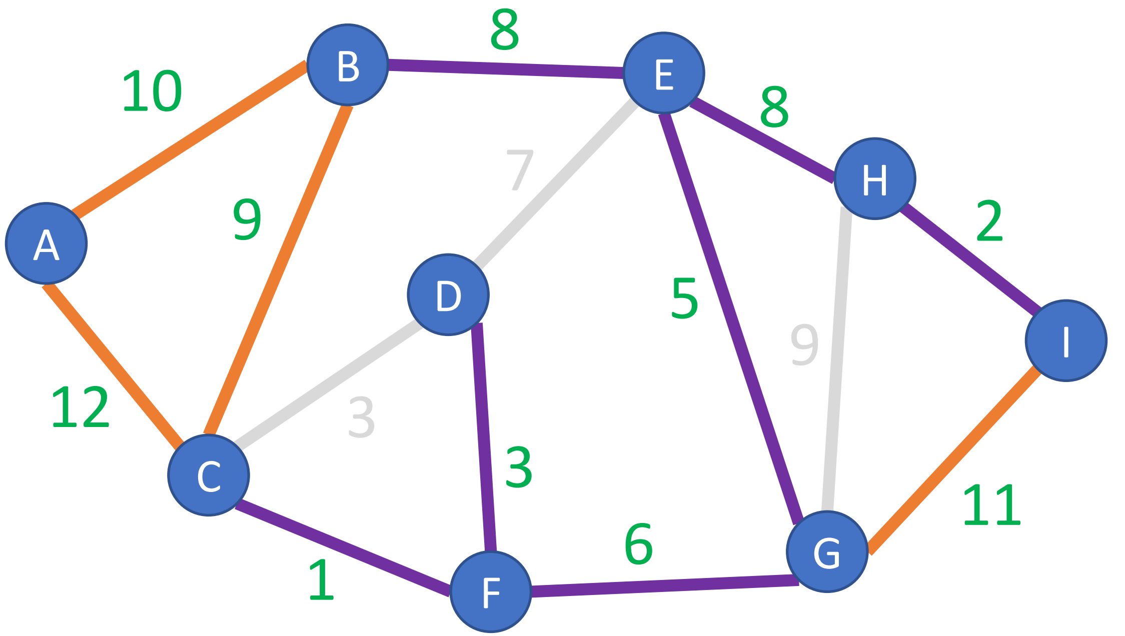
Kruskal’s Algorithm

1. Start with an empty set of edges T
2. Repeatedly add to T the lowest-weight edge that does not create a cycle. (Stop when we’ve added $n - 1$ edges.)



Kruskal's Algorithm

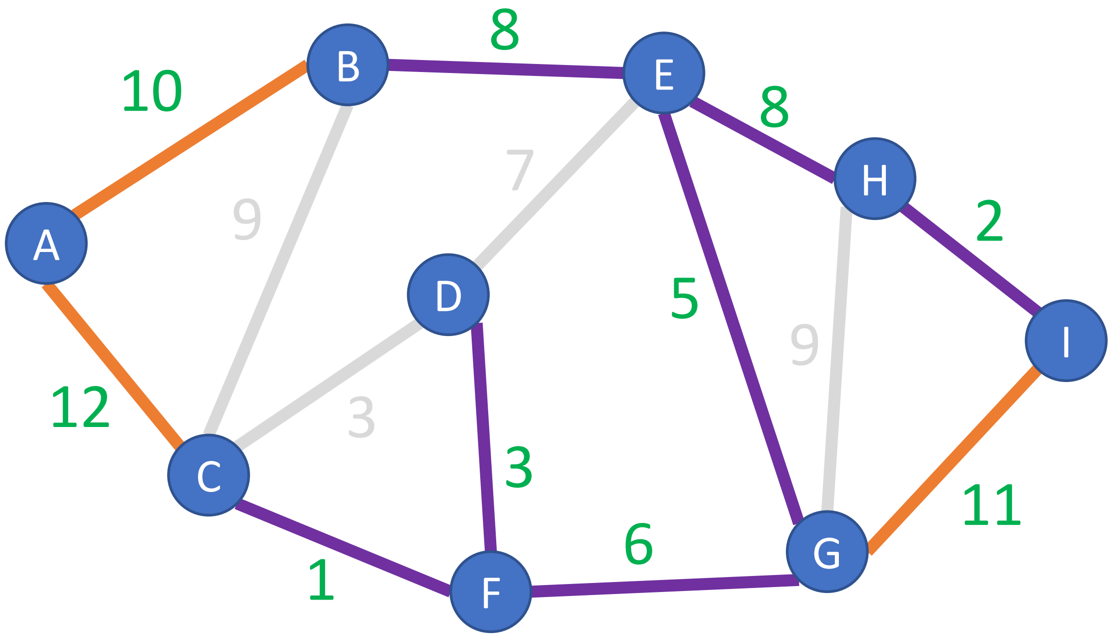
1. Start with an empty set of edges T
2. Repeatedly add to T the lowest-weight edge that does not create a cycle. (Stop when we've added $n - 1$ edges.)



Edge forms a cycle, so do not include

Kruskal's Algorithm

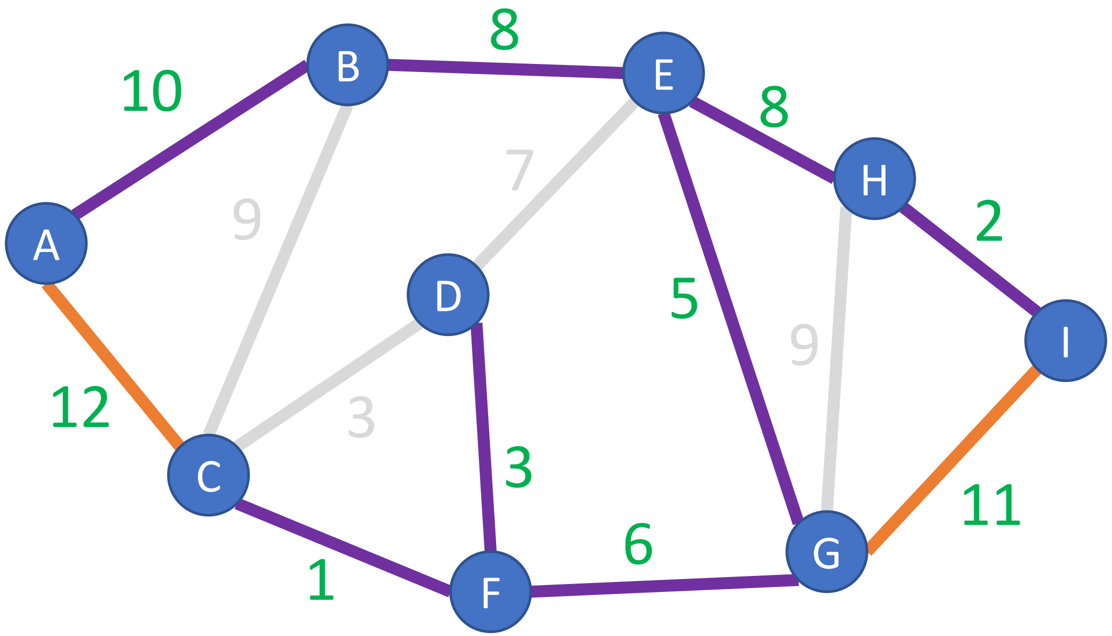
1. Start with an empty set of edges T
2. Repeatedly add to T the lowest-weight edge that does not create a cycle. (Stop when we've added $n-1$ edges.)



Edge forms a cycle, so do not include

Kruskal's Algorithm

1. Start with an empty set of edges T
2. Repeatedly add to T the lowest-weight edge that does not create a cycle. (Stop when we've added $n-1$ edges.)



Now $n-1$ edges have been added.
All nodes are connected.
Algorithm is done!

Kruskal’s Algorithm

1. Start with an empty tree T
2. Repeatedly add to T the lowest-weight edge that does not create a cycle

Implementation: iterate over each of the edges in the graph (sorted by weight), and maintain nodes in a union-find (also called disjoint-set) data structure:

- Data structure that tracks elements partitioned into different sets
- **Union:** Merges two sets into one
- **Find:** Given an element, return the index of the set it belongs to
- Both “union” and “find” operations are very fast

Time complexity: $O\big(\alpha(n)\big)$,
where α is the “inverse Ackermann function” (extremely slow-growing function)
for all “practical” n, $\alpha(n) < 5$ (e.g., for all $n < 2^{2^{2^{65536}}} - 3$)

Time Complexity: Kruskal’s Algorithm

1. Start with an empty tree T
2. Repeatedly add to T the lowest-weight edge that does not create a cycle

Implementation: iterate over each of the edges in the graph (sorted by weight), and maintain nodes in a union-find (also called disjoint-set) data structure:

- Data structure that tracks elements partitioned into different sets
- **Union:** Merges two sets into one
- **Find:** Given an element, return the index of the set it belongs to
- Both “union” and “find” operations are very fast

- **Overall running time:** $O(|E| \log |E|) = O(|E| \log |V|)$

$$|E| \leq |V|^2 \Rightarrow \log|E| = O(\log|V|)$$

More on Implementation for Kruskal’s

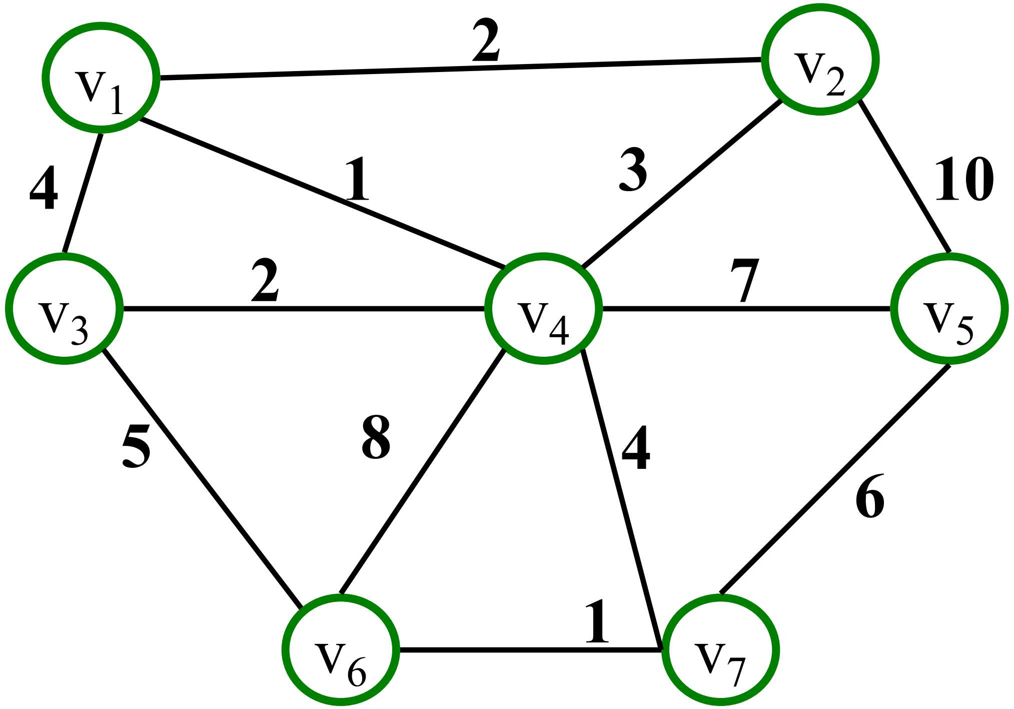
```
Let EL be the set of edges sorted ascending by weight
Consider each vertex to be in a tree of size 1
For each edge e in EL
    T1 = tree ID for vertex head(e)
    T2 = tree ID for vertex tail(e)
    if (T1 != T2)    // the nodes are not in the same Tree
        Add e to the output set of edges T (which becomes the MST)
        Combine trees T1 and T2
```

Seems simple, no?

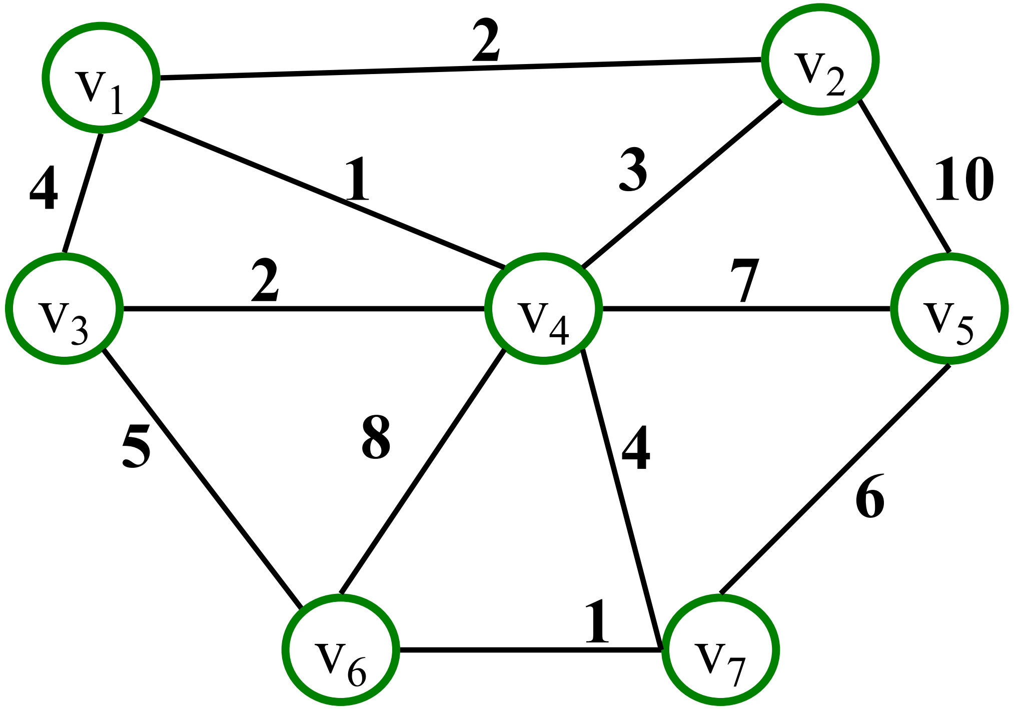
- But, how do you keep track of what tree a vertex is in?
- Trees are sets of vertices. Need to findset(v) and “union” two sets

Practice

Can you do Prim's MST on This?



MST



v1

{v1, v4}

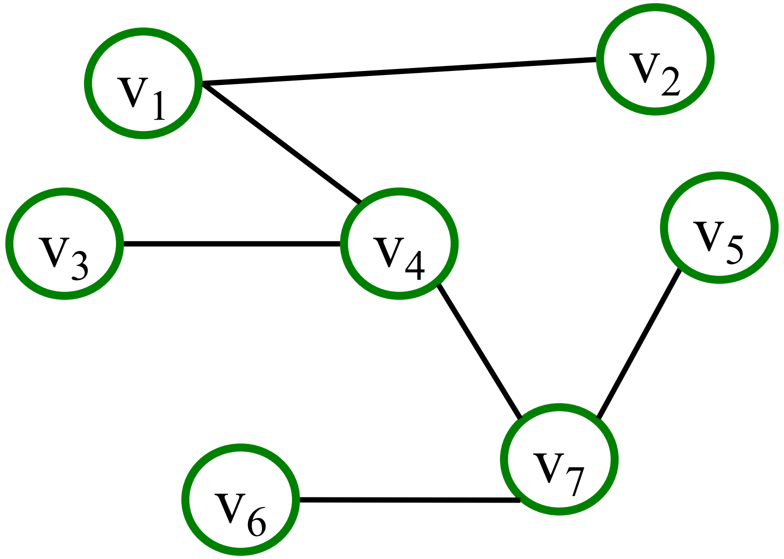
{v1, v2}

{v4, v3}

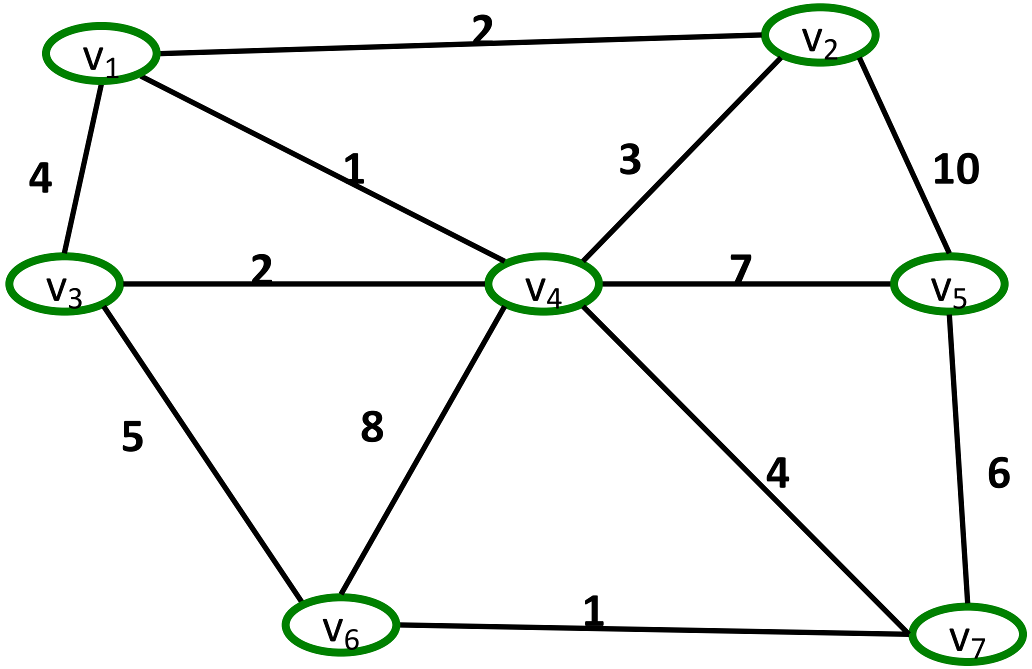
{v4, v7}

{v7, v6}

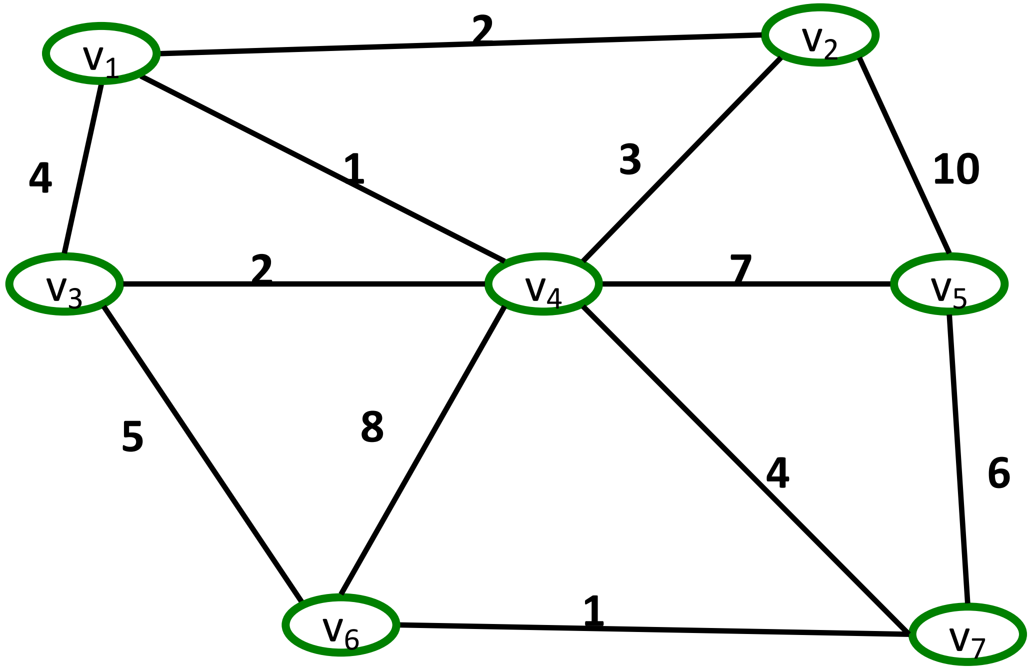
{v7, v5}

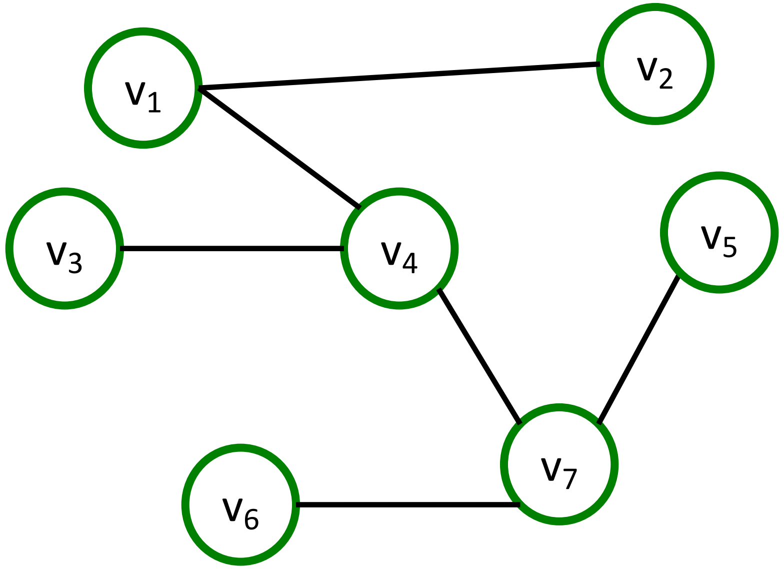


Can you do Kruskal’s MST on This?



MST and Kruskal's Example





Cost(MST) = 16

Disjoint Sets and Find/Union Algorithms

Readings: CLRS 19.3

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

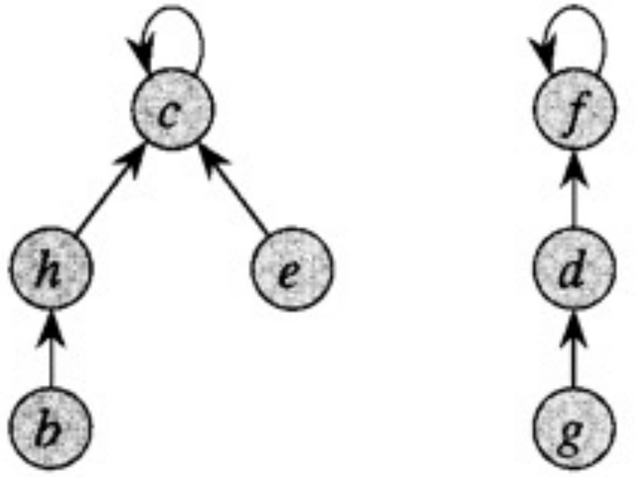
- void makeSet(int n)   // construct n independent sets
- int findSet(int i)   // given i, which set does i belong to?
- void union(int i, int j)   // merge sets containing i and j

Represent Sets As Trees

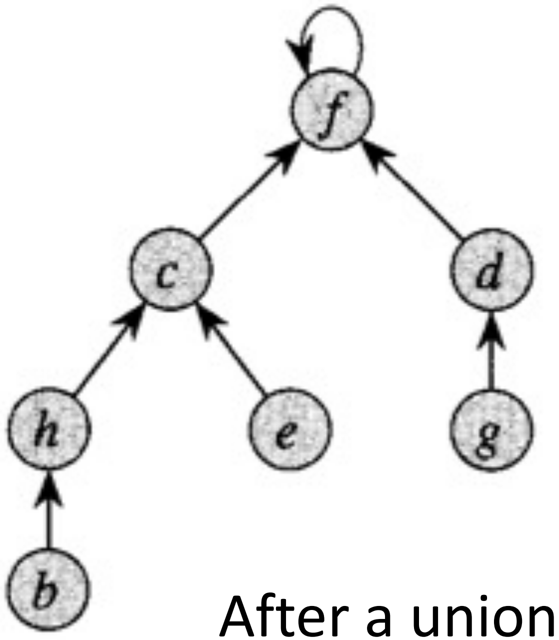
In our implementation, we’ll represent each set as a tree

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*

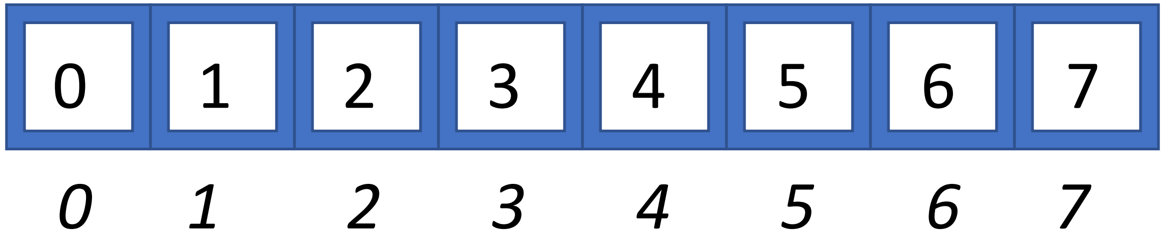
Union/Find and Disjoint Sets

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*

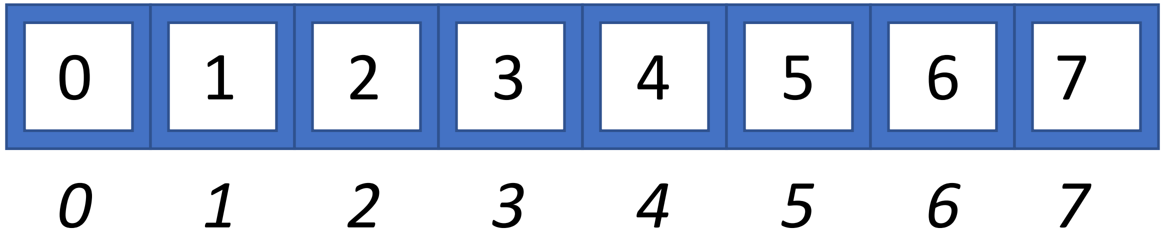
Union/Find and Disjoint Sets

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

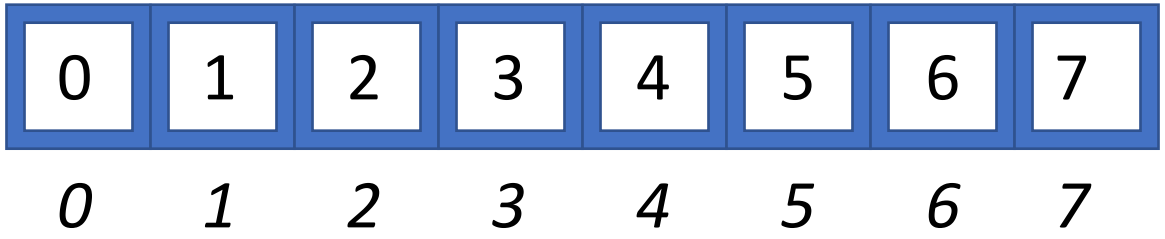


0	1	2	3	4	5	6	7
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*

Union/Find and Disjoint Sets

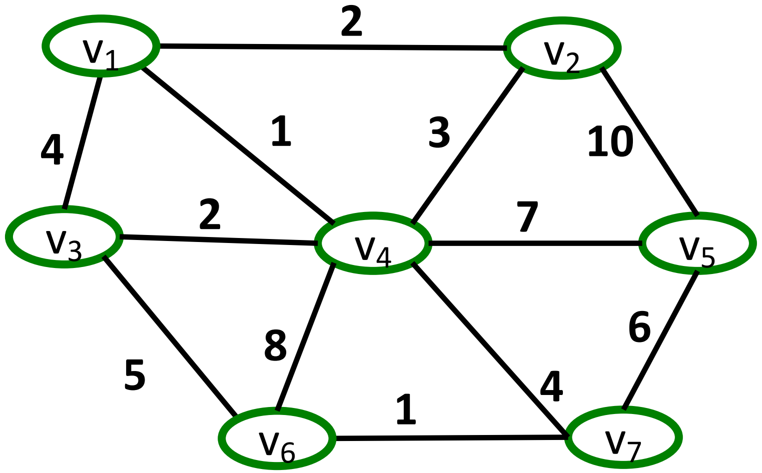
Example:

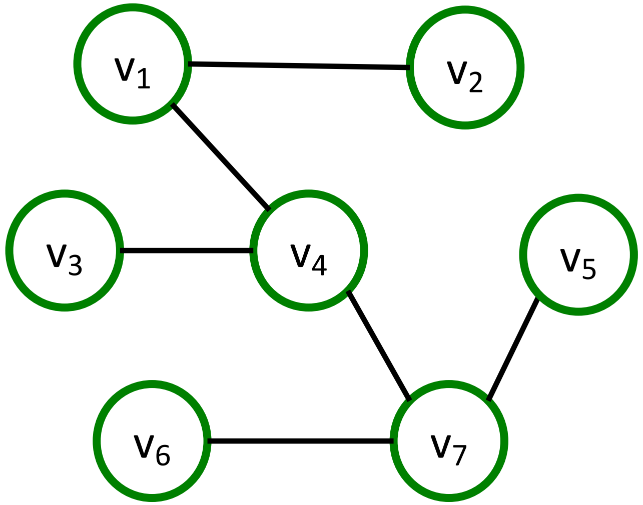
- union(4,5)
- union(6,7)
- union(1,2)
- union(5,6)
- find(1); find(4); find(6)



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()

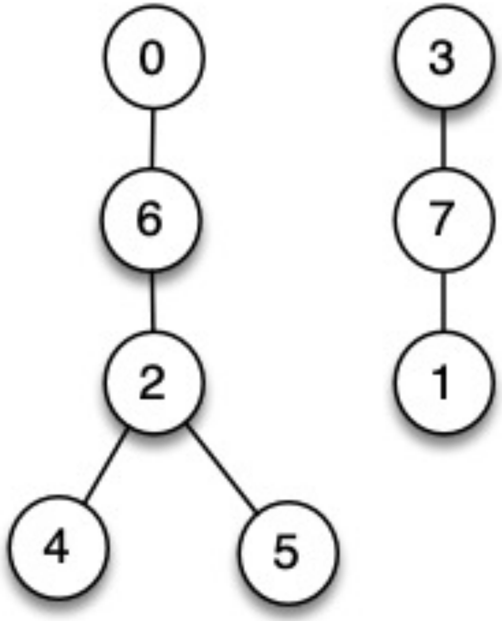
- Linear if have to trace a long way to get to label
- Constant if lucky and input is the label (root note) or near it

union()

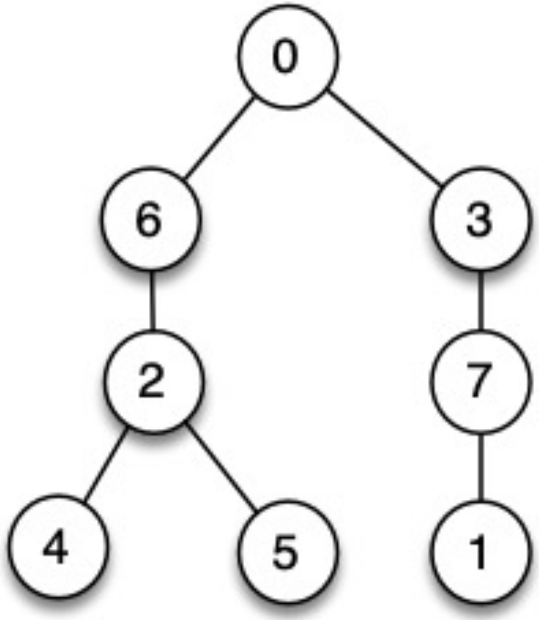
- Constant to change the label BUT…
- Could be linear to find the two labels first.

Optimization 1: Union by rank

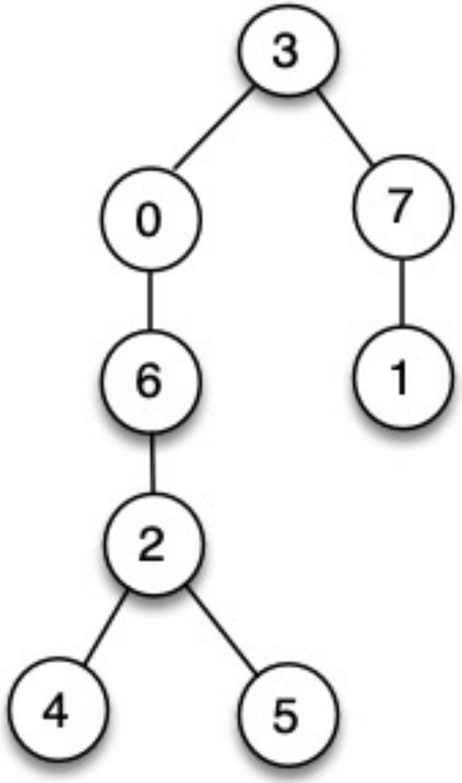
Two Sets:



Union’d under 0:



Union’d under 3:



Optimization 1: Union by rank

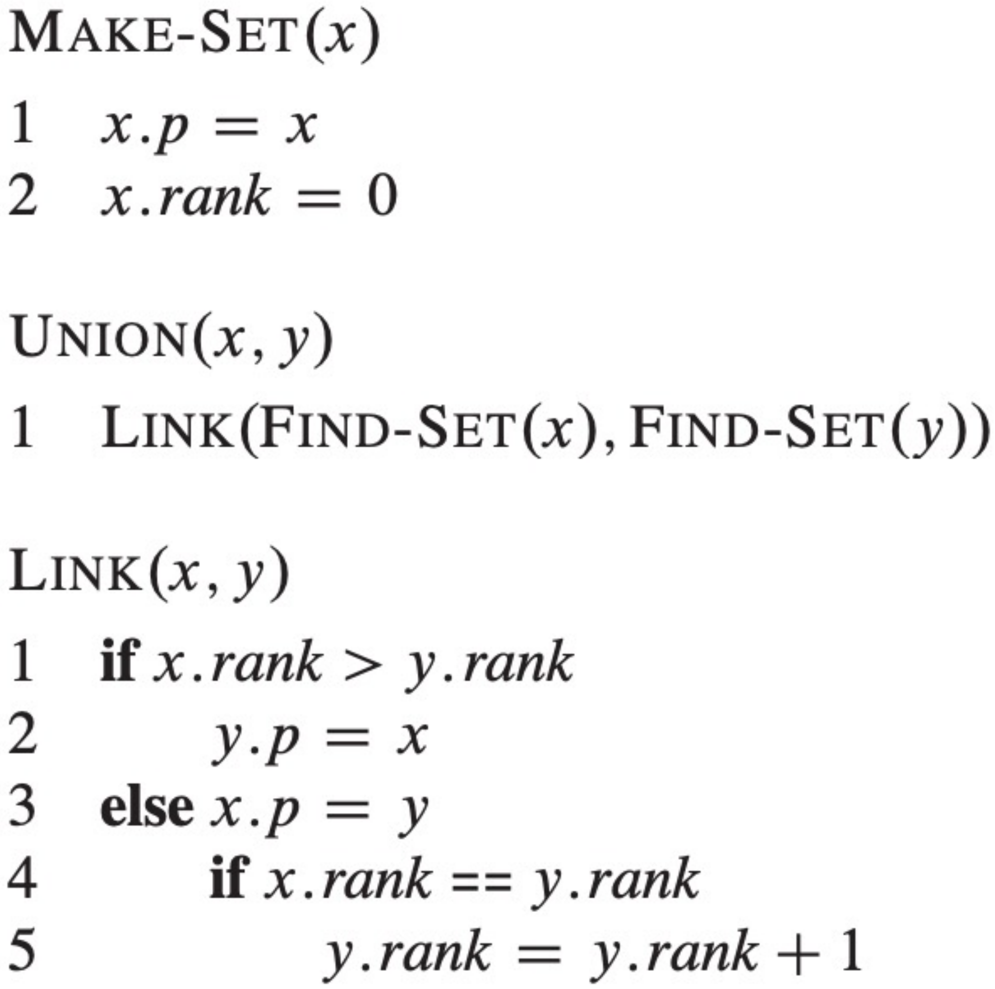
Easy to implement!!

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



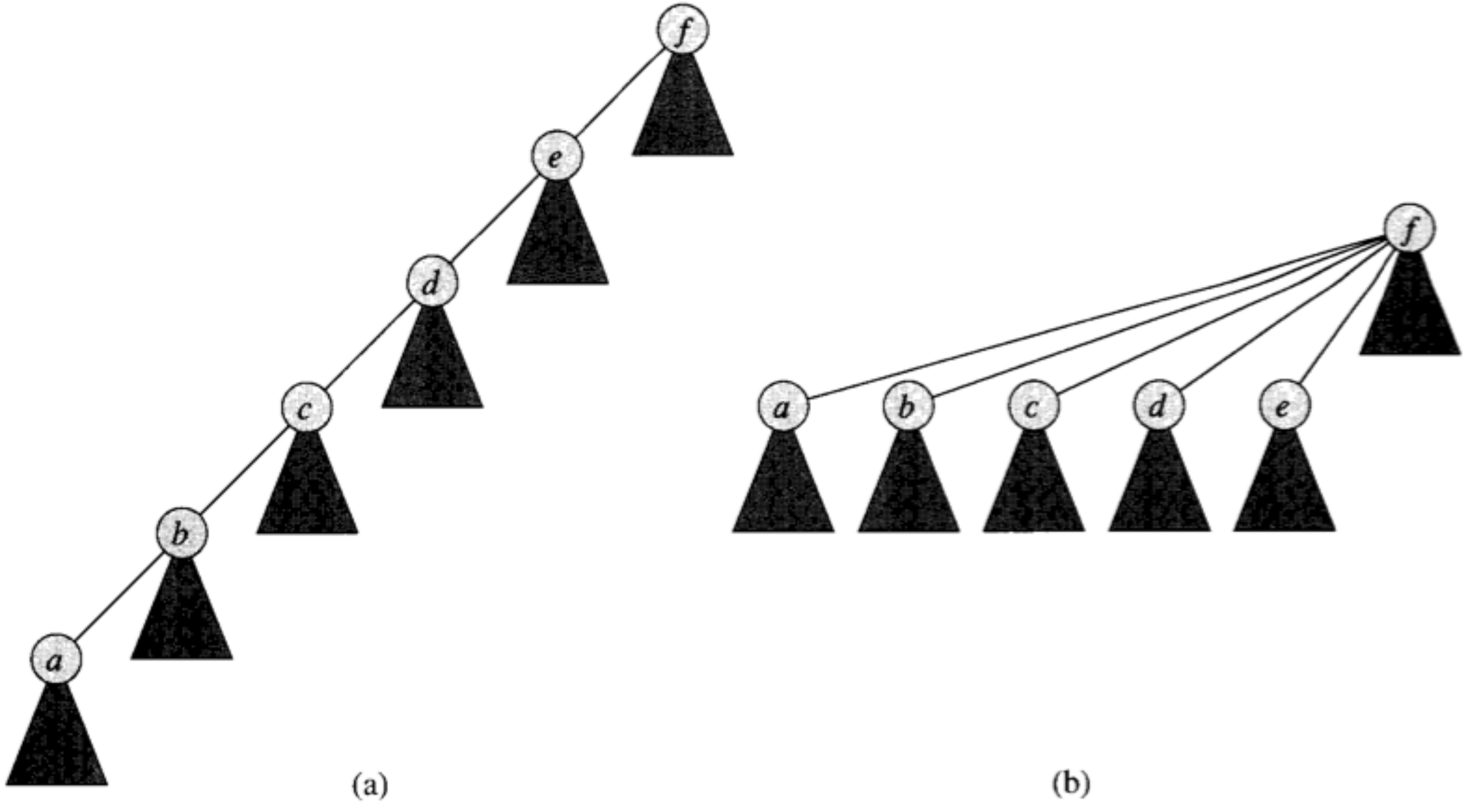
```
MAKE-SET($x$)
    $x.p = x$
    $x.rank = 0$

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

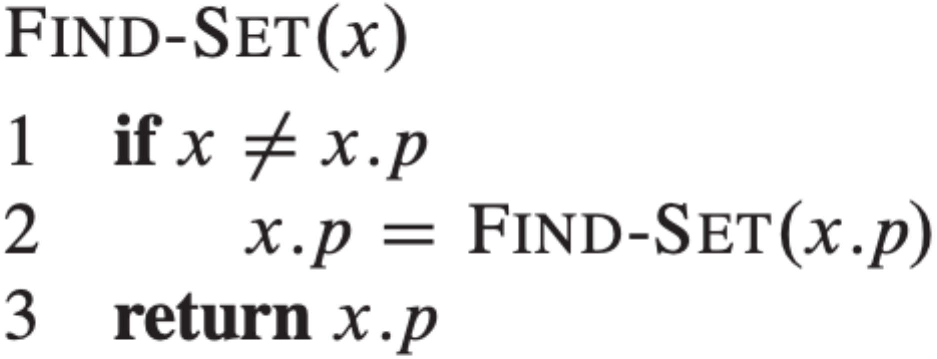
Later finds will be faster



Optimization 2: Path Compression

Also easy to implement

- CLRS code uses recursion →
- Or would loop and keep a list



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

```
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
```

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*

Summary

What did we learn?

Minimum Spanning Trees

Prim’s Algorithm

- Very similar to Dijkstra’s SP algorithm
- Different greedy choice to add next edge to tree

Kruskal’s Algorithm

Find-union

- How to implement
- How to optimize
- How it affects runtime of Kruskal’s algorithm.