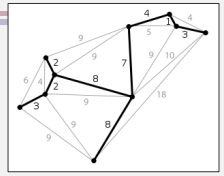
## Minimum Spanning Tree



A *spanning tree* is a tree (no cycles) connecting all of the vertices of the graph.

A *minimum spanning tree* is a spanning with the lowest total cost of its edges.

Observations –

- every spanning tree on a connected graph with n vertices has exactly n-1 edges
  - justification: repeatedly remove a degree 1 vertex and its incident edge until there is only one vertex (and no edges) left – n-1 vertices and edges have been removed
    - there is always at least one such vertex in a tree with n > 1 or else there would be a cycle
    - there is still a tree after removing a vertex and incident edge – removing from a tree doesn't introduce cycles and a leaf is never a cut vertex so its removal doesn't disconnect the tree
- if the edge weights are distinct, there is a unique MST
- if the edge weights are not distinct, the MST may not be unique



## Observations – Cut Property

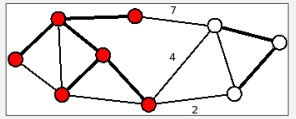
Observation: (*cut property*)

Let G = (V,E) and let S be a subset of V. Then the cheapest edge *e* connecting a vertex in S and a vertex in V-S is part of some MST of G.

intuition –

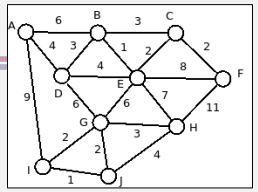
let S be the red vertices and V-S be the white vertices

exactly one of the three labeled edges is needed to complete the spanning tree (shown in bold) – anything but the cheapest won't be an MST






## Algorithms for MST




Prim's algorithm –

- start with a tree T containing a single vertex S
- repeatedly add the cheapest edge connecting a vertex in S and a vertex in V-S to T

Kruskal's algorithm –

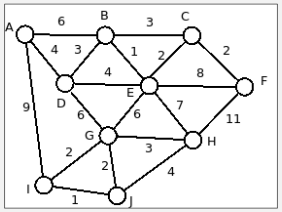
- start with a tree T containing no edges
- repeatedly add the lowest-cost edge remaining that connects two different chunks of the tree-in-progress



## Algorithms for MST

Prim's algorithm –

- start with a tree T containing a single vertex S
- repeatedly add the cheapest edge connecting a vertex in S and a vertex in V-S to T



# Prim's Algorithm

The idea:

- repeatedly add the cheapest edge connecting a vertex in S and a vertex in V-S to T

Implementation details:

- cheapest edge connecting S to V-S → ??
  - the set of eligible edges changes as new vertices are added to the tree → sounds like a priority queue ordered by edge weight

```
mark s as visited
add s's incident edges to PQ
while PQ is not empty (and T has fewer than n-1 edges)
  e ← PQ.removeMin()
  if e has an unvisited end vertex v,
    add e to T
    mark v as visited
    add v's incident edges to PQ (omitting those connecting
      to already-visited vertices)
```



# Prim's Algorithm

```
5  mark s as visited
3 4 add s's incident edges to PQ
1  while PQ is not empty (and T has fewer than
     n-1 edges)
2    e ← PQ.removeMin()
5    if e has an unvisited end vertex v,
       add e to T
5      mark v as visited
3 4    add v's incident edges to PQ (omitting
         those connecting to already-visited
         vertices)
```

other steps contribute O(1) per

Running time?

- pick any starting vertex
  → O(1)

❶ one iteration per edge
  → O(m)

❷ removeMin
  → O(log m) per iteration, up to m iterations

❸ traverse incident edges
  → O(m) total
  - iterate through 2m edges (once from each end) at O(1) per
  - need incident edges → choose adjacency list implementation for graph

❹ add incident edges to queue
  → O(m log n) total
  - O(log m) to add to queue; each edge is added at most once

why O(m log n) instead of O(m log m)? for a simple graph, $m = O(n^2)$ and $\log n^2 = 2 \log n$

❺ mark as visited / check status
  → O(1) per

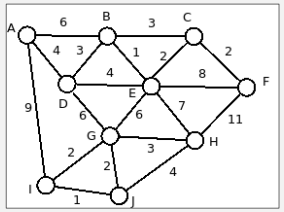
→ total: O(m log n)
  – using heap implementation of priority queue



# Algorithms for MST

Kruskal's algorithm –

- start with a tree T containing no edges
- repeatedly add the lowest-cost edge remaining that connects two different chunks of the tree-in-progress






# Kruskal's Algorithm

The idea:

- repeatedly add the lowest-cost edge remaining that connects two different chunks of the tree-in-progress

Implementation details:

- "lowest-cost edge remaining"
  → edges are considered in order by weight and are all known at the beginning, so sort them
- "connects two different chunks of the tree-in-progress"
  → need a data structure which efficiently supports
  - determine if two vertices belong to the same component
  - merge two components
  - initialize with each vertex in a separate component

## Union-Find (Disjoint-Set)

The *disjoint-set* (or *union-find*) ADT supports the following operations –

- makeset(x) – create a set containing a single element x
- find(x) – determine the set x belongs to
- union(x,y) – merge two sets x and y

In the context of Kruskal's algorithm –

- at the beginning, every vertex is in its own set – makeset(x)
- an edge (u,v) connects different sets if find(u) ≠ find(v)
- adding an edge (u,v) to the spanning tree combines two sets – union(u,v)



## Kruskal's Algorithm

```
5 MST ← empty set
1 for each v in V
    makeset(v)
2 E' ← sort E by weight
  for each edge e=(u,v) in E'
3   if find(u) != find(v)
5     add e to MST
4     union(u,v)
```

Running time using union-find?

1. initialization: makeset(v) for each vertex
   - O(makeset) per iteration, n iterations
2. finding the lowest-cost edge
   - can sort edges by weight, then iterate through
   - O(m log n) to sort + O(1) time per iteration, m iterations
3. determine if an edge connects two separate chunks
   - O(find) per iteration, m iterations
4. combine two chunks when an edge is chosen
   - O(union) per edge chosen, n-1 edges chosen

5 steps contribute O(1) per

→ total: O(n × makeset + m log n + m × find + n × union)

if we can get O(log n) makeset, find, union then this will be the same as Prim's



## Union-Find Summary

total: O(n × makeset + m log n + m × find + n × union)

- union-by-rank list implementation yields O((n+m) log n) for Kruskal's algorithm
  - O(1) makeset(x)
  - O(1) find(x)
  - O(n log n) for a series of *n* union(x,y)
- union-by-rank tree implementation with path compression yields O(m log n) for Kruskal's algorithm
  - O(1) makeset(x)
  - effectively O(1) find(x) and union(x,y)
    - the tree height is a very slow-growing log*
      - log* = number of times log must be applied to get a value ≤ 1
    - *amortized* over a series of operations



## MST

- Prim's algorithm is O(m log n) for a standard (heap-based) PQ
  - O(n log n) for a sparse graph, O($n^2$ log n) for a dense graph
- Kruskal's algorithm is O((n+m) log n) or O(m log n) amortized depending on union-find implementation
  - O(n log n) for a sparse graph, O($n^2$ log n) for a dense graph

Can we do better?

- Prim's running time is dominated by the queue operations
  - more efficient insert and remove isn't that feasible (we need both), but what about doing fewer operations?

## MST

Prim's running time is dominated by the queue operations.

Observation: many of the edges in the priority queue aren't useful because they connect within S.

- store the vertices in V-S in the priority queue instead of edges, ordered by the cost of the cheapest edge between the vertex and a vertex of S
  - for a connected graph, $n \leq m+1$
  - the idea is to maintain a collection of what could be the next vertex added to the spanning tree, along with the cheapest cost of connecting that vertex



## Prim's Algorithm

```
algorithm prim(G,w)
 input: connected undirected
  graph G with edge weights w
 output: MST defined by the
  'prev' labels

for all u in V
  cost[u] ← ∞
  prev[u] ← null
s ← a vertex of G
cost[s] ← 0

PQ ← makeQueue(V)
while PQ is not empty
  v ← PQ.removeMin()
  for each incident edge e=(v,z)
    if cost[z] > w(v,z) then
      cost[z] = w(v,z)
      prev[z] = (v,z)
      PQ.decreaseKey(z)
```

For each vertex in V-S, keep track of the cheapest known edge connecting it to S.

- prev(v) = the cheapest known edge connecting v to S
- cost(v) = weight of edge prev(v)

"Known" edges are those incident on vertices in S.

- the information is complete for any vertex in V-S connected to one in S
- update prev/cost information when we add a vertex to S



## Prim's Algorithm

Running time.

1. initialize `cost`, `prev` labels
   - O(n) with O(1) record-keeping and O(1) per element iteration (possible with any graph implementation)
2. make PQ with n elements
   - O(makePQ)
3. n is-empty checks + n vertices removed from PQ
   - O(n × 1) + O(n × removeMin) [O(1) isEmpty() possible with any PQ implementation)
4. 2m edges visited over all vertices [2m because an undirected edge is visited from each of its endpoints]
   - O(m) assuming O(1) per element iteration (possible with adjacency list)
5. up to O(1) + O(decreaseKey) per edge (if label is updated)
   - assuming O(1) record-keeping

→ O(makePQ + n × removeMin + m × decreaseKey) total

```
algorithm prim(G,s):
 for all u in V do                    1
   cost[u] ← ∞
   prev[u] ← null
 s ← a vertex of G
 cost[s] ← 0
PQ ← makeQueue(V)                     2
while PQ is not empty do              3
  v ← PQ.removeMin()
for each incident edge e=(v,z)        4
  if cost[z] > w(v,z) then
    cost[z] = w(v,z)                  5
    prev[z] = (v,z)
    PQ.decreaseKey(z)
```



## Prim's Algorithm

→ O(makePQ + n × removeMin + m × decreaseKey) total

| operation | heap |
|---|---|
| makeQueue | O(n log n) – repeated insert<br>O(n) – heapify |
| removeMin | O(log n) |
| decreaseKey | O(log n) (**) |
| total running time for Prim's algorithm | O(n + n log n + m log n) = O((n+m) log n)<br>= O(m log n) for connected graphs |

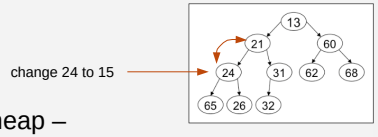
This is –

- O(n log n) for sparse graphs [$m = O(n)$]
- $O(n^2 \log n)$ for dense graphs [$m = O(n^2)$]

(**) assuming O(1) to locate element within PQ (which can be done with a *locator*), otherwise O(n) to search PQ to find entry

# Locators

- decrease key can be done in O(log n) time if the location (index) of the key in the heap is known
  - update key, then bubble up (min heap) to fix ordering property




- searching for a key in a heap –
  - O(n) – may need to check all elements
- how to avoid search?
  - look up instead!
  - what if we add a map key → array index?
    - O(1) to locate key with hashtable
    - map must be updated for each bubble up or bubble down swap
      - but that only involves two elements, so two updates – +O(1) per swap doesn't change the big-Oh for bubbling