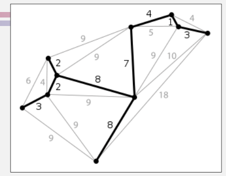
Minimum Spanning Tree



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

A *minimum spanning tree* is the spanning tree 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

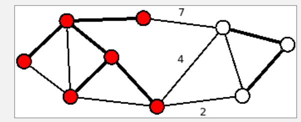
Observation: (*cut property*)

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.

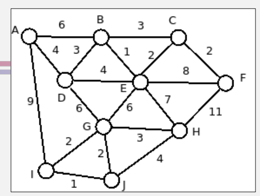
intuition –

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



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

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

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, 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

Running time using union-find?

- initialization: makeset(v) for each vertex
  - O(makeset) per iteration, n iterations
- 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
- determine if an edge connects two separate chunks
  - O(find) per iteration, m iterations
- combine two chunks when an edge is chosen
  - O(union) per edge chosen, n-1 edges chosen

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