Greedy algorithms

Constructing minimum spanning trees

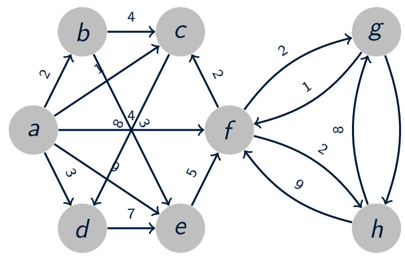
Tyler Moore

CSE 3353, SMU, Dallas, TX

Lecture 14

Some slides created by or adapted from Dr. Kevin Wayne. For more information see http://www.cs.princeton.edu/~wayne/kleinberg-tardos. Some code reused from Python Algorithms by Magnus Lie Hetland.

Weighted Graph Data Structures



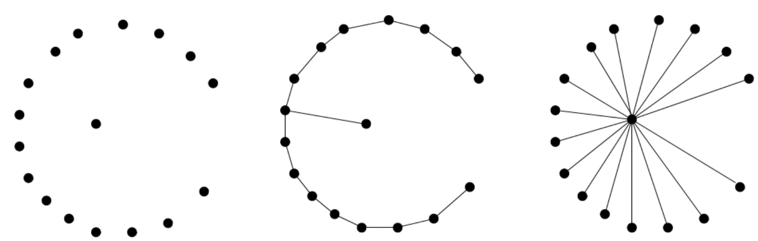
Nested Adjacency Dictionaries w/ Edge Weights

```
N = {
 'a':{'b':2,'c':1,'d':3,'e':9,'f':4},
 'b':{'c':4,'e':3},
 'c':{'d':8},
 'd':{'e':7},
 'e':{'f':5},
 'f':{'c':2,'g':2,'h':2},
 'g':{'f':1,'h':6},
 'h':{'f':9,'g':8}
}
>>> 'b' in N['a'] # Neighborhood membership
True
>>> len(N['f'])   # Degree
3
>>> N['a']['b']
# Edge weight for (a, b)
2
```


Minimum Spanning Trees

- A tree is a connected graph with no cycles
- A spanning tree is a subgraph of G which has the same set of vertices of G and is a tree
- A minimum spanning tree of a weighted graph G is the spanning tree of G whose edges sum to minimum weight
- There can be more than one minimum spanning tree in a graph (consider a graph with identical weight edges)
- Minimum spanning trees are useful in constructing networks, by describing the way to connect a set of sites using the smallest total amount of wire

Minimum Spanning Trees



Why Minimum Spanning Trees

- The minimum spanning tree problem has a long history – the first algorithm dates back to at least 1926!
- Minimum spanning trees are taught in algorithms courses since
  1. it arises in many applications
  2. it gives an example where greedy algorithms always give the best answer
  3. Clever data structures are necessary to make it work efficiently
- In greedy algorithms, we decide what to do next by selecting the best local option from all available choices, without regard to the global structure.

Prim's algorithm

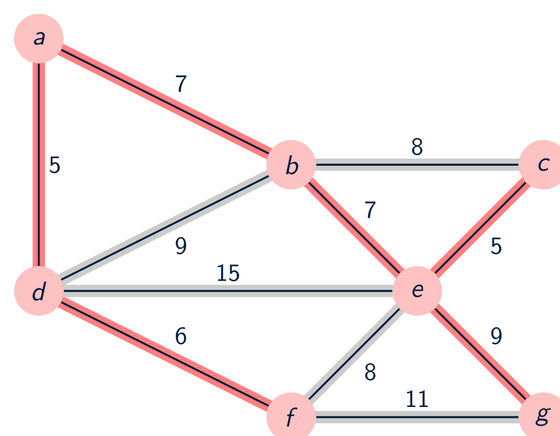
- If G is connected, every vertex will appear in the minimum spanning tree. (If not, we can talk about a minimum spanning forest.)
- Prims algorithm starts from one vertex and grows the rest of the tree an edge at a time.
- As a greedy algorithm, which edge should we pick? The cheapest edge with which can grow the tree by one vertex without creating a cycle.

Prim's algorithm

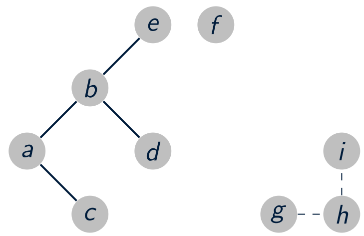
- During execution each vertex v is either in the tree, fringe (meaning there exists an edge from a tree vertex to v) or unseen (meaning v is more than one edge away).

```
def Prim-MST(G):
    Select an arbitrary vertex s to start the tree from.
    While (there are still non-tree vertices)
        Select the edge of minimum weight between
            a tree and nontree vertex.
        Add the selected edge and vertex to the
            minimum spanning tree.
```


Example run of Prim's algorithm



Correctness of Prim's algorithm

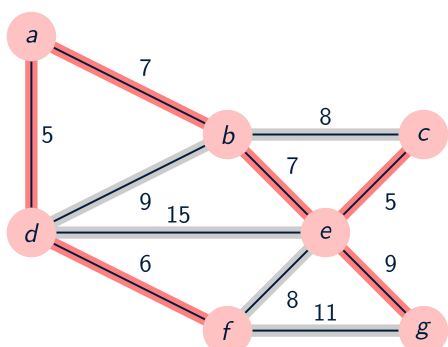


- Let's talk through a "proof" by contradiction
  1. Suppose there is a graph G where Prim's alg. does not find the MST
  2. If so, there must be a first edge (e, f) Prim adds so that the partial tree cannot be extended to an MST
  3. But if (e, f) is not in $MST(G)$, there must be a path in $MST(G)$ from e to f since the tree is connected. Suppose (d, g) is the first path edge.
  4. $W(e, f) \geq W(d, g)$ since (e, f) is not in the MST
  5. But $W(d, g) \geq W(e, f)$ since we assume Prim made a mistake
  6. Thus, by contradiction, Prim must find an MST

Efficiency of Prim's algorithm

- Efficiency depends on the data structure we use to implement the algorithm
- Simplest approach is $O(nm)$:
  1. Loop through all vertices ($O(n)$)
  2. At each step, check edges and find the lowest-cost fringe edge that finds an unseen vertex ($O(n)$)
- But we can do better ($O(m + n \lg n)$) by using a priority queue to select edges with lower weight

Prim's algorithm implementation

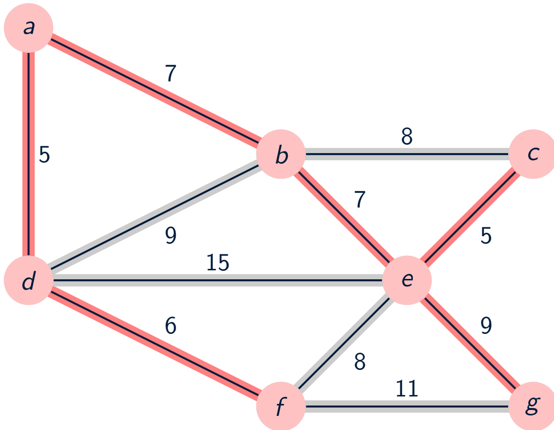



```
G = {
  'a':{'b':7,'d':5},
  'b':{'a':7,'d':9,'c':8,'e':7},
  'c':{'b':8,'e':5},
  'd':{'a':5,'b':9,'e':15,'f':6},
  'e':{'b':7,'c':5,'d':15,'f':8,'g':9},
  'f':{'d':6,'e':8,'g':11},
  'g':{'e':9,'f':11}
}
```


Prim's algorithm implementation

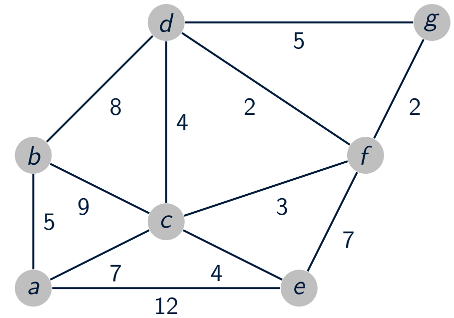
```
from heapq import heappop, heappush
def prim_mst(G, s):
    V, T = [],{}  #V: vertices in MST, T: MST
    # Priority Queue (weight, edge1, edge2)
    Q = [(0, None, s)]
    while Q:
        _, p, u = heappop(Q)#choose edge w/ smallest weight
        if u in V: continue #skip any vertices already in MST
        V.append(u)
        #build MST structure
        if p is None:
            pass
        elif p in T:
            T[p].append(u)
        else:
            T[p]=[u]
        for v, w in G[u].items(): #add new edges to fringe
            heappush(Q, (w, u, v))
    return T
"""
>>> prim_mst(G,'d')
{'a': ['b'], 'c': ['e'], 'b': ['c'], 'e': ['g'], 'd': ['a', 'f']}
```

Output from Prim's algorithm implementation




```
>>> prim_mst(G,'d')
{'a': ['b'], 'b': ['e'], 'e': ['c', 'g'], 'd': ['a', 'f']}
```


Exercise: Compute Prim's algorithm starting from *a* (number edges by time added)

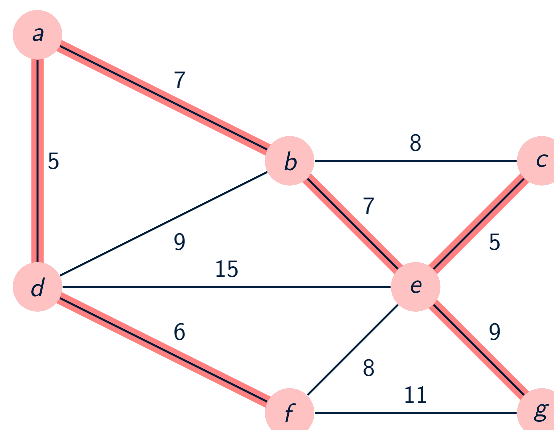


Kruskal's algorithm

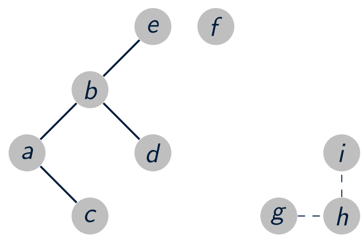
- Instead of building the MST by incrementally adding vertices, we can incrementally add the smallest edges to the MST so long as they don't create a cycle

```
def Kruskal-MST(G):
    Put the edges in a list sorted by weight
    count = 0
    while (count<n-1) do
        Get the next edge from the list (v,w)
        if (component(v) != component(w))
            add (v,w) to MST
            count+=1
            merge component(v) and component(w)
```


Example run of Kruskal's algorithm

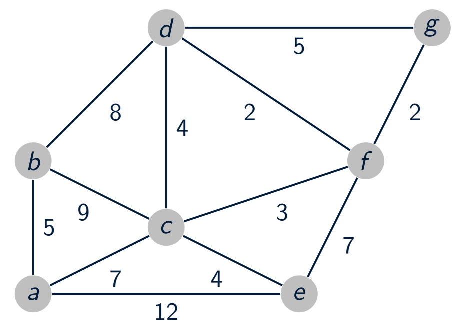


Correctness of Kruskal's algorithm



- Let's talk through a "proof" by contradiction
  1. Suppose there is a graph G where Kruskal does not find the MST
  2. If so, there must be a first edge (e, f) Kruskal adds so that the partial tree cannot be extended to an MST
  3. Inserting (e, f) in $MST(G)$ creates a cycle
  4. Since e & f were in different components when (e, f) was inserted, at least one edge (say (d, g)) in $MST(G)$ must be evaluated after (e, f).
  5. Since Kruskal adds edges by increasing weight, $W(d, g) \geq W(e, f)$
  6. But then replacing (d, g) with (e, f) in the MST creates a smaller tree
  7. Thus, by contradiction, Kruskal must find an MST

Exercise: Compute Kruskal's algorithm (number edges by time added)



How fast is Kruskal's algorithm?

- What is the simplest implementation?
  - Sort the m edges in O(m lg m) time.
  - For each edge in order, test whether it creates a cycle in the forest we have thus far built
  - If a cycle is found, then discard, otherwise add to forest. With a BFS/DFS, this can be done in $O(n)$ time (since the tree has at most n edges).
- What is the running time?
  - $O(mn)$
  - Can we do better?
  - Key is to increase the efficiency of testing component membership

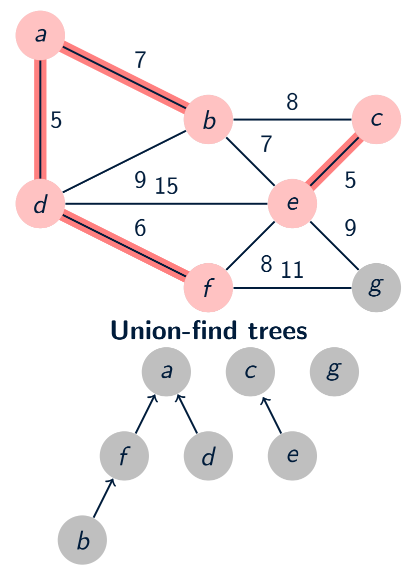
A necessary detour: set partition

- A set partition is a partitioning of the elements of a universal set (i.e., the set containing all elements) into a collection of disjoint subsets
- Consequently, each element must be in exactly one subset
- We've already seen set partitions with bipartite graphs
- We can represent the connected components of a graph as a set partition
- So we need to find an algorithm that can solve the set partition problem efficiently: enter the union-find algorithm

Union-Find Algorithm

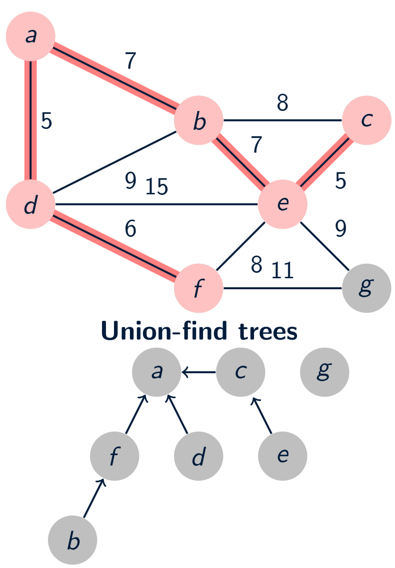
- We need a data structure for maintaining sets which can test if two elements are in the same and merge two sets together.
- These can be implemented by <u>union</u> and <u>find</u> operations, where
  - `find(`i`)` Return the label of the root of tree containing element i, by walking up the parent pointers until there is no where to go.
  - `union(`i`,`j`)`: Link the root of one of the trees (say containing i) to the root of the tree containing the other (say j) so `find(`i`)` now equals `find(j)`.
- Ideally, we'd like the find to be logarithmic in the number of nodes and the union to take constant time
- Why do we only link the root of the trees together in `union` and not all nodes in the tree?

Example of Union-Find



v	C[v]	findk(v)	R[v]
a	a	a	2
b	f	a	0
c	c	c	1
d	a	a	0
e	c	c	0
f	a	a	1
g	g	g	0

Example of Union-Find



v	C[v]	findk(v)	R[v]
a	a	a	2
b	f	a	0
c	~~c~~ a	~~c~~ a	1
d	a	a	0
e	c	~~c~~ a	0
f	a	a	1
g	g	g	0

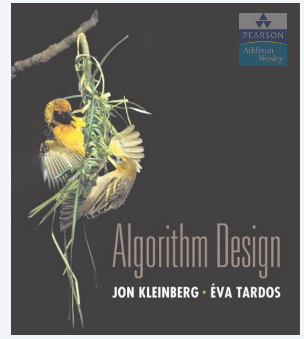
Implementing Union-Find

```
def findk(C, u):      # Find component rep.
    while C[u] != u: # Rep. would point to itself
        u = C[u]
    return u

def unionk(C, R, u, v):
    u, v = findk(C, u), findk(C, v)
    if R[u] > R[v]:    # Union by rank
        C[v] = u
    else:
        C[u] = v
    if R[u] == R[v]:   # A tie: Move v up a level
        R[v] += 1
```

Implementing Kruskal's algorithm

```
def kruskal(G):
    E = [(G[u][v],u,v) for u in G for v in G[u]]
    T = set()              # Empty partial solution
    C = {u:u for u in G} # Component reps
    R = {u:0 for u in G}
    for _, u, v in sorted(E):  # Edges, sorted by weight
        if findk(C, u) != findk(C, v):
            T.add((u, v))      # Different reps? Use it!
            unionk(C, R, u, v) # Combine components
    return T
```

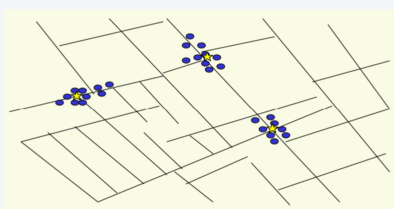
SECTION 4.7

4. GREEDY ALGORITHMS II

- Dijkstra's algorithm
- minimum spanning trees
- Prim, Kruskal, Boruvka
- *single-link clustering*
- min-cost arborescences

Clustering

Goal. Given a set U of n objects labeled $p_1, \ldots, p_n$, partition into clusters so that objects in different clusters are far apart.



outbreak of cholera deaths in London in 1850s (Nina Mishra)

Applications.

- Routing in mobile ad hoc networks.
- Document categorization for web search.
- Similarity searching in medical image databases
- Skycat: cluster 10^9 sky objects into stars, quasars, galaxies.
- ...

Clustering of maximum spacing

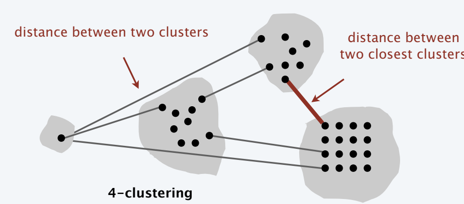
k-clustering. Divide objects into k non-empty groups.

Distance function. Numeric value specifying "closeness" of two objects.

- $d(p_i, p_j) = 0$ iff $p_i = p_j$ [identity of indiscernibles]
- $d(p_i, p_j) \geq 0$ [nonnegativity]
- $d(p_i, p_j) = d(p_j, p_i)$ [symmetry]

Spacing. Min distance between any pair of points in different clusters.

Goal. Given an integer k, find a k-clustering of maximum spacing.

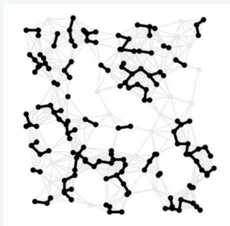


4-clustering

Greedy clustering algorithm

"Well-known" algorithm in science literature for single-linkage k-clustering:

- Form a graph on the node set U, corresponding to n clusters.
- Find the closest pair of objects such that each object is in a different cluster, and add an edge between them.
- Repeat $n - k$ times until there are exactly k clusters.



Key observation. This procedure is precisely Kruskal's algorithm (except we stop when there are k connected components).

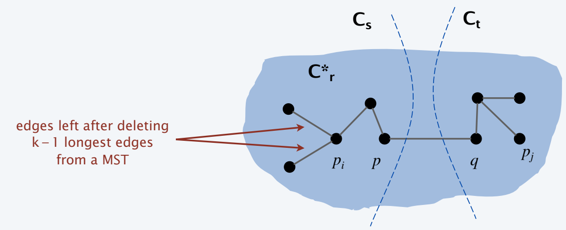
Alternative. Find an MST and delete the $k - 1$ longest edges.

Greedy clustering algorithm: analysis

Theorem. Let C^* denote the clustering $C^*_1, \ldots, C^*_k$ formed by deleting the $k - 1$ longest edges of an MST. Then, C^* is a k-clustering of max spacing.

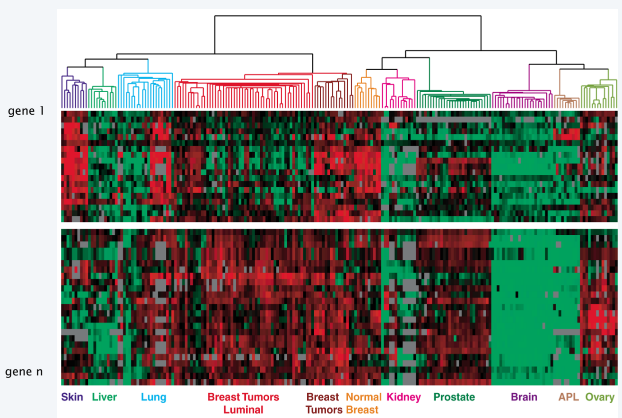
Pf. Let C denote some other clustering $C_1, \ldots, C_k$.

- The spacing of C^* is the length d^* of the $(k-1)^{\text{st}}$ longest edge in MST.
- Let p_i and p_j be in the same cluster in C^*, say C^*_r, but different clusters in C, say C_s and C_t.
- Some edge (p, q) on $p_i - p_j$ path in C^*_r spans two different clusters in C.
- Edge (p, q) has length $\leq d^*$ since it wasn't deleted.
- Spacing of C is $\leq d^*$ since p and q are in different clusters. ▪



Dendrogram of cancers in human

Tumors in similar tissues cluster together.



Reference: Botstein & Brown group

gene expressed

gene not expressed