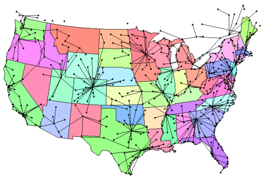
Lecture 20
Minimum Spanning Trees



EECS 281: Data Structures & Algorithms

The Minimum Spanning Tree Problem

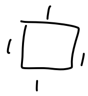
- **Given**: edge-weighted, *undirected*, connected graph $G = (V, E)$
- **Find**: subgraph $T = (V, E')$, $E' \subseteq E$ such that
  - All vertices are pair-wise connected
  - The sum of all edge weights in T is minimal
- See a cycle in T?
  - Remove edge with highest weight
- Therefore, T must be a tree (no cycles)

MST Quiz

1. Prove that a unique shortest edge must be included in every MST
2. ~~Prove for second shortest edge~~
3. ~~What about third shortest edge?~~

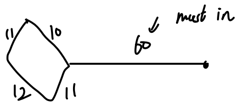
without creating cycle with smaller ones

4. Show a graph with > 1 MST
5. Show a graph and its MST which avoids *some* shortest edge



6. Show a graph where every longest edge must be in every MST

must in



Prim's Algorithm

- Find an MST on edge-weighted, connected, *undirected* graphs
- Greedily select edges one by one and add to a growing sub-graph
- Grows a tree from a single vertex

Prim's Algorithm

1. Initialize a tree with a single vertex, chosen arbitrarily from the graph
2. Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and add that vertex to the tree
3. Repeat step 2 (until all vertices are in the tree)

Prim's Algorithm

- Given graph $G = (V, E)$
- Start with 2 sets of vertices: 'innies' & 'outies'
  - 'Innies' are visited nodes (initially empty)
  - 'Outies' are not yet visited (initially V)
- Select first innie arbitrarily (root of MST)
- Repeat until no more outies
  - Choose outie (v') with smallest distance from <u>*any*</u> innie
  - Move v' from outies to innies
- Implementation issue: use linear search or PQ*?*

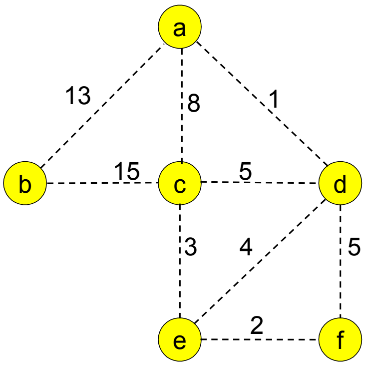
Prim: Data structures

- A vector of classes or structures
- For each vertex v, record:
  - k_v: Has v been visited? (initially **false** for all $v \in V$)
  - d_v: What is the minimal edge weight to v? (initially **∞** for all $v \in V$, except $v_r = 0$)
  - p_v: What vertex precedes (is parent of) v? (initially **unknown** for all $v \in V$)

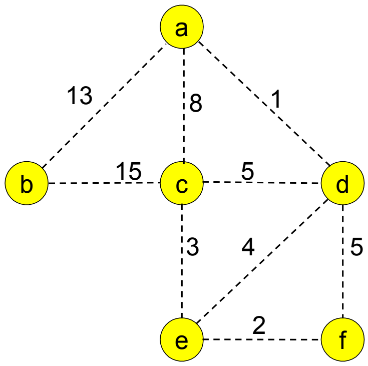
Implementing Prim's

- Implement in the <u>order listed</u>:
  - 1: Loop over <u>all</u> vertices: find smallest false k_v
  - 2: Mark k_v as true
  - 3: Loop over all vertices: update false neighbors of k_v
- Common Mistake: Set the first vertex to true outside the loop
- Reordering this can result in a simple algorithm that simply doesn't work

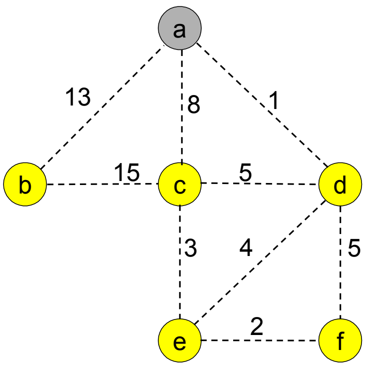
v	k_v	d_v	p_v
a	***F***	∞	-
b	***F***	∞	-
c	***F***	∞	-
d	***F***	∞	-
e	***F***	∞	-
f	***F***	∞	-



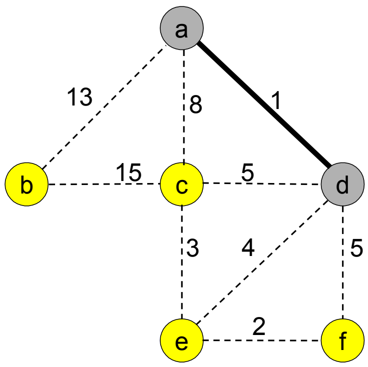
v	k_v	d_v	p_v
a	***F***	0	-
b	***F***	∞	-
c	***F***	∞	-
d	***F***	∞	-
e	***F***	∞	-
f	***F***	∞	-



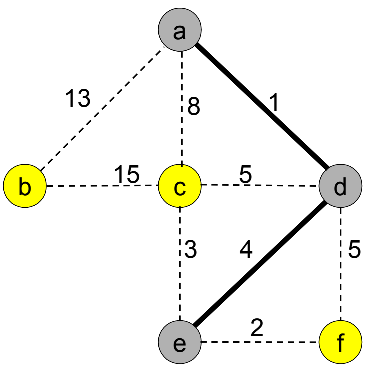
v	k_v	d_v	p_v
a	***T***	0	-
b	***F***	13	a
c	***F***	8	a
d	***F***	1	a
e	***F***	∞	-
f	***F***	∞	-



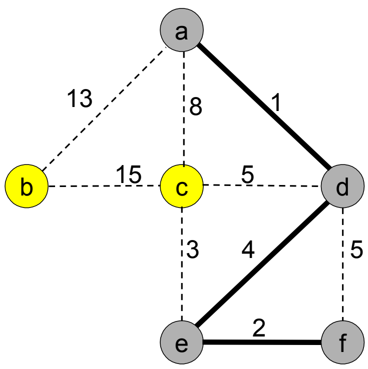
v	k_v	d_v	p_v
a	***T***	0	-
b	***F***	13	a
c	***F***	5	d
d	***T***	1	a
e	***F***	4	d
f	***F***	5	d



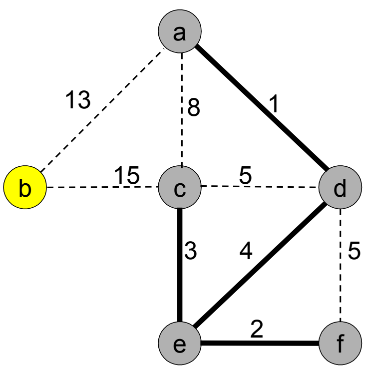
v	k_v	d_v	p_v
a	***T***	0	-
b	***F***	13	a
c	***F***	3	e
d	***T***	1	a
e	***T***	4	d
f	***F***	2	e



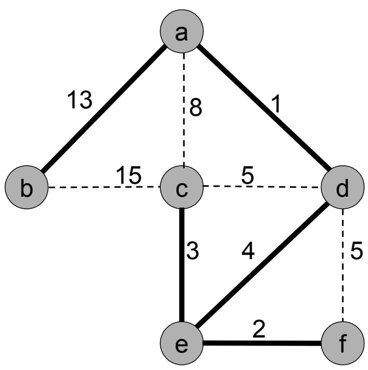
v	k_v	d_v	p_v
a	***T***	0	-
b	***F***	13	a
c	***F***	3	e
d	***T***	1	a
e	***T***	4	d
f	***T***	2	e



v	k_v	d_v	p_v
a	***T***	0	-
b	***F***	13	a
c	***T***	3	e
d	***T***	1	a
e	***T***	4	d
f	***T***	2	e

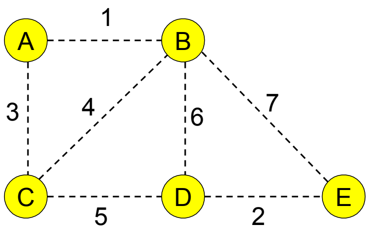


v	k_v	d_v	p_v
a	***T***	0	-
b	***T***	13	a
c	***T***	3	e
d	***T***	1	a
e	***T***	4	d
f	***T***	2	e



MST This! (Prim's)

Using Prim's; start at node A



v	K_v	D_v	p_v
A	F	0	-
B	F	∞	-
C	F	∞	-
D	F	∞	-
E	F	∞	-

Complexity – Linear Search

```
Loop v times:
1. From the set of vertices for which kv is
   false, select the vertex v having the
   smallest tentative distance dv.
2. Set kv to true.
3. For each vertex w adjacent to v for which
   kw is false, test whether dw is greater
   than distance(v,w). If it is, set dw to
   distance(v,w) and set pw to v.
```

$|V|$ times

$O(1)$

$O(|V|)$

Most at this vertex: $O(|V|)$. Cost of each: $O(1)$.

Prim's (Heap) Algorithm

Algorithm Prims_Heaps(G, s_0)	
//Initialize	
n = |V|	***O*(1)**
create_table(n) //stores k,d,p	***O*(*V*)**
create_pq() //empty heap	***O*(1)**
table[s_0].d = 0	***O*(1)**
insert_pq(0, s_0)	***O*(1)**

Prim's (Heap) Algorithm

while (!pq.isempty)	***O*(*E*)**
v_0 = getMin() //heap top() & pop()	***O*(log *E*)**
if (!table[v_0].k) //not known	***O*(1)**
table[v_0].k = true	***O*(1)**
for each $v_i \in$ Adj[v_0]	***O*(1 + *E*/*V*)**
if (!table[v_i].k)	***O*(1)**
distance = weight(v_0, v_i)	***O*(1)**
if (distance < table[v_i].d)	***O*(1)**
table[v_i].d = distance	***O*(1)**
table[v_i].p = v_0	***O*(1)**
insert_pq(distance, v_i)	***O*(log *E*)**

Complexity – Heaps

```
Repeat until the PQ is empty:
1. From the set of vertices for which kv is
   false, select the vertex v having the
   smallest tentative distance dv.
2. Set kv to true.
3. For each vertex w adjacent to v for which
   kw is false, test whether dw is greater
   than distance(v,w). If it is, set dw to
   distance(v,w) and set pw to v.
```

$|E|$ times

$O(1)$

$O(\log |E|)$

Most at this vertex: $O(|V|)$. Cost of each: $O(\log |E|)$.
Note: Visits every edge once (over all iterations) = $O(|E|)$.

Prim's: Complexity Summary

- ***O*(V^2)** for the simplest nested-loop implementation
  $\geq |V|-1$ and usually much larger (if dense)
- ***O*(*E* log *E*)** with heaps
  - Is this always faster?
  - Think about the complexity of the PQ version for dense versus sparse graphs

Kruskal's Algorithm

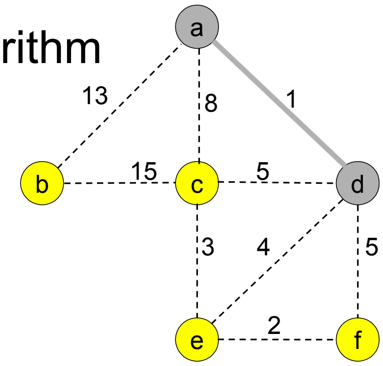
- Find an MST on edge-weighted, connected, *undirected* graphs
- Greedily select edges one by one and add to a growing sub-graph
- Grows a <u>forest</u> of trees that eventually merges into a single tree

Kruskal's Algorithm

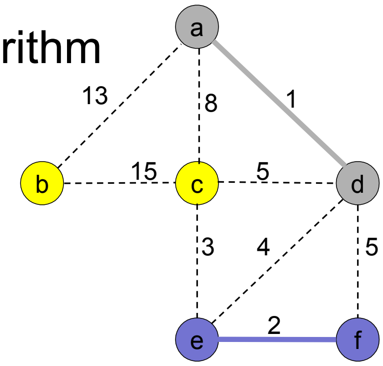
1. Presort all edges: ***O*(*E* log *E*) ≈ *O*(*E* log *V*)** time
2. Try inserting in order of increasing weight
3. Some edges will be discarded so as not to create cycles

- Initial edges may be disjoint
  - Kruskal's grows a <u>forest</u> (union of disjoint trees)

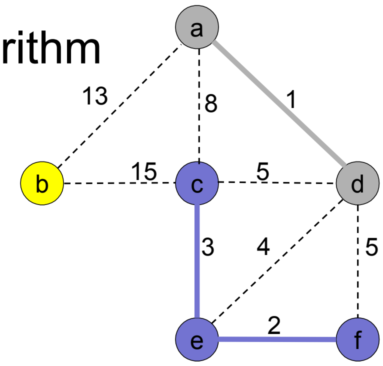
Kruskal's Algorithm



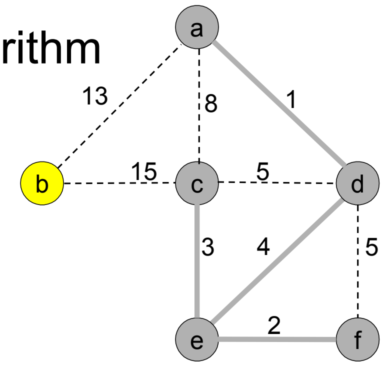
Kruskal's Algorithm



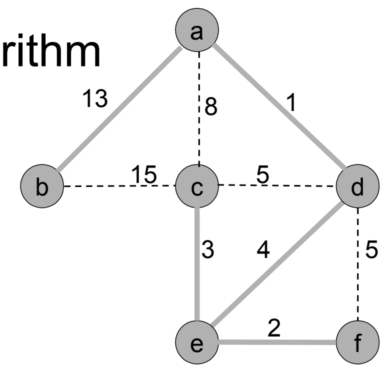
Kruskal's Algorithm



Kruskal's Algorithm



Kruskal's Algorithm



Kruskal: Complexity Analysis

- Sorting takes ***O*(*E* log *E*)**
  - Happens to be the bottleneck of entire algorithm
- Remaining work: a loop over *E* edges
  - Discarding an edge is trivial ***O*(1)**
  - Adding an edge is easy ***O*(1)**
  - Most time spent testing for cycles *O*(?)
  - Good news: takes less than log *E* ≈ log *V*
- Key idea: if vertices *k* and *j* are *already* connected, then a new edge would create a cycle
  - Only need to maintain disjoint sets

Maintaining Disjoint Sets

- *N* locations with no connecting roads
- Roads are added one by one
  - Distances are unimportant (for now)
  - Connectivity is important
- Want to connect cities ASAP
  - Redundant roads would slow us down

Q: For two cities *k* and *j*, would road *(k, j)* be redundant?

A: Use a Union-Find data structure.

MST Summary

- MST is lowest-cost sub-graph that
  - Includes all nodes in a graph
  - Keeps all nodes connected
- Two algorithms to find MST
  - Prim: iteratively adds closest node to current tree – very similar to Dijkstra, ***O*(V^2)** or ***O*(*E* log *E*)**
  - Kruskal: iteratively builds forest by adding minimal edges, ***O*(*E* log *E*)**
- For dense *G*, use the nested-loop Prim variant
- For sparse *G*, Kruskal is faster
  - Relies on the efficiency of sorting algorithms
  - Relies on the efficiency of union-find