Lecture 19
Minimum Spanning Trees

EECS 281: Data Structures & Algorithms

The Minimum Spanning Tree Problem

Given: edge-weighted, *undirected* graph $G = (V,E)$

Find: subgraph $T = (V, E')$, $E' \subseteq E$ such that

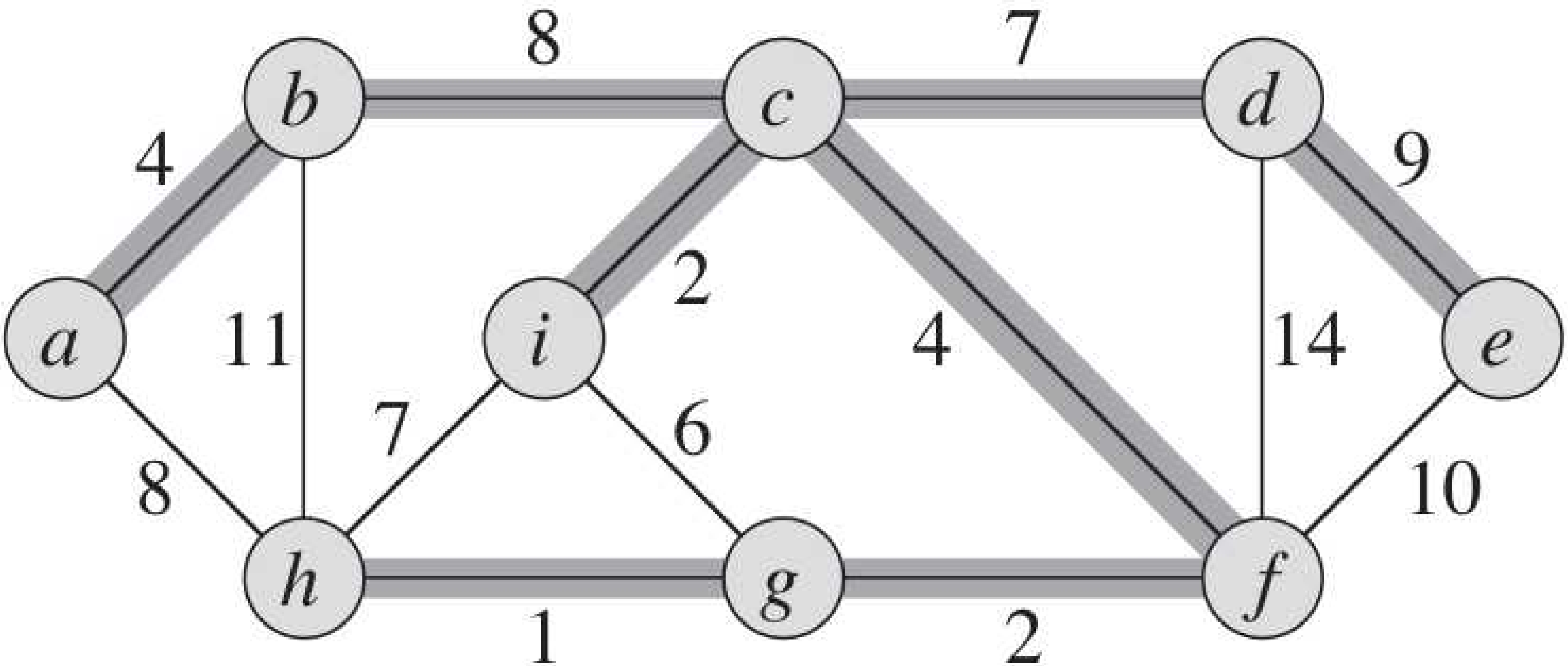
- All vertices are pair-wise connected
- The sum of all edge weights in T is minimal

- See a cycle in T? - Get rid of an edge
  - Therefore, T must be a tree (no cycles)

Planar MST: vertices are planar points

- All pair-wise edges are present
- Weights are distances

Example



CLRS

MST Quiz

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

Prim's & Kruskal's Algorithms

- Algorithms for finding MSTs on edge-weighted, connected, *undirected* graphs
- Greedily select edges one by one and add to a growing sub-graph
  - Prim grows a real tree
  - Kruskal grows a forest of trees that eventually merges into a single 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)
- Iteratively (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

- Three arrays
- 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 $v \in V$)

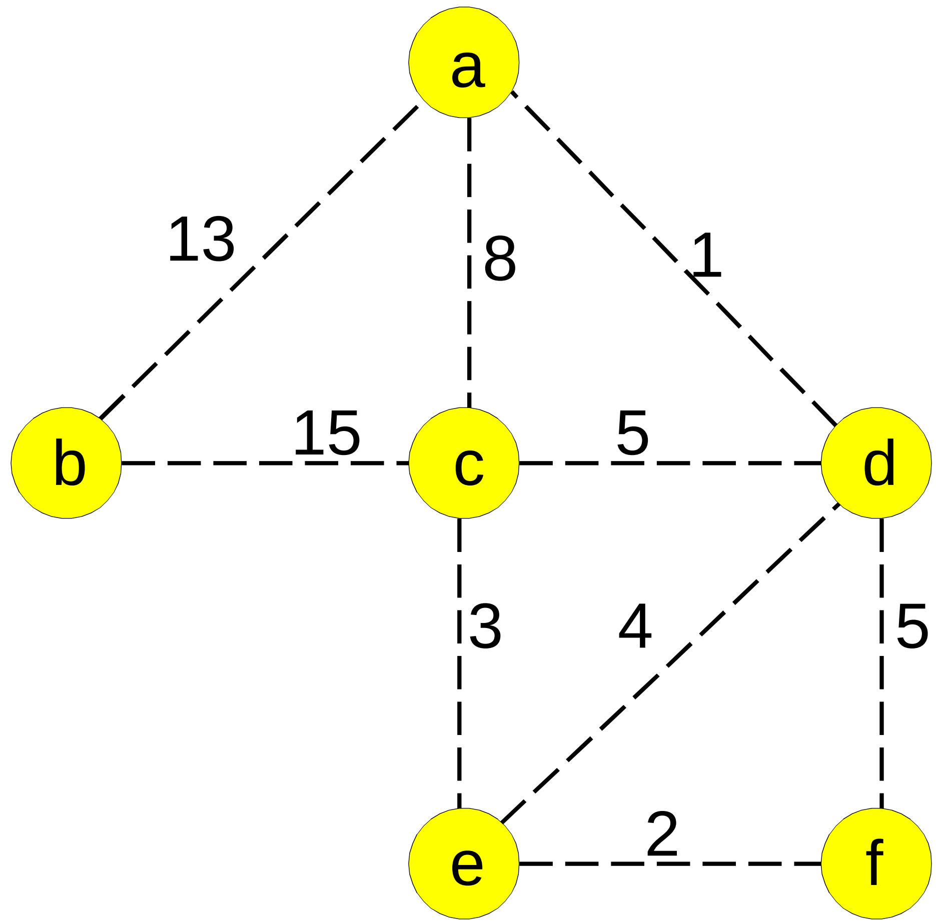
Prim's Algorithm

Set starting point distance to 0

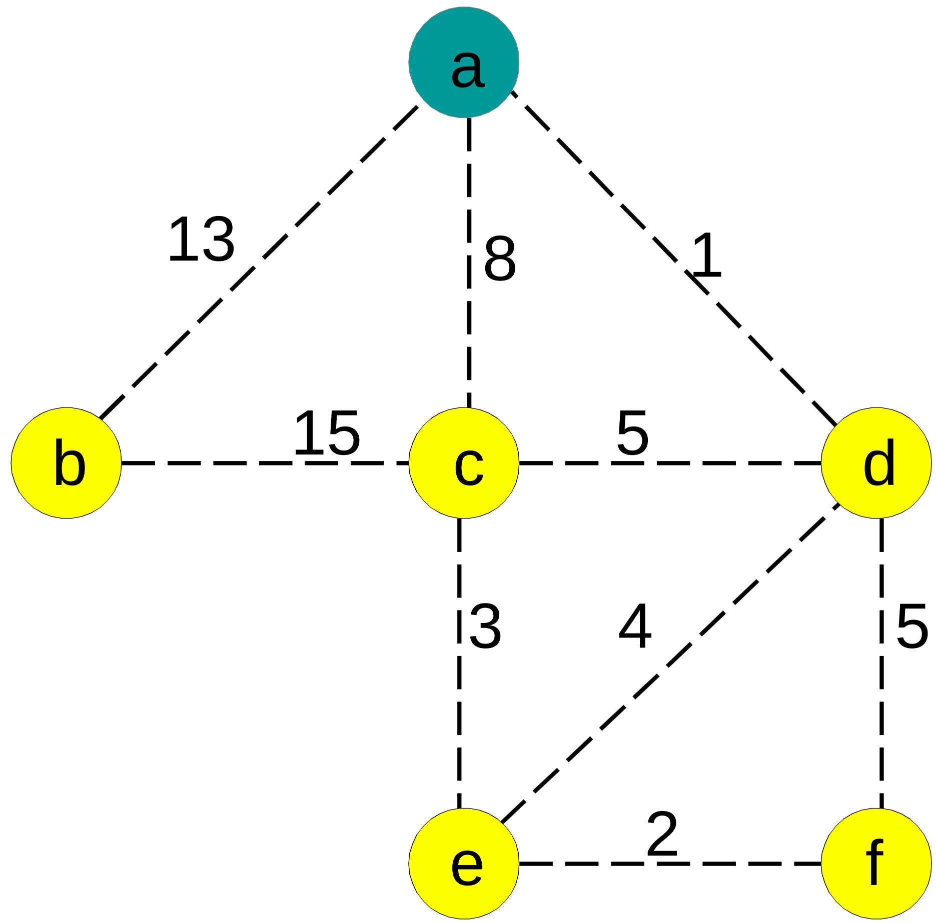
Repeat until every k_v is true:

1. **From the set of vertices for which k_v is false, select the vertex v having the smallest tentative distance d_v**
2. **Set k_v to true**
3. **For each vertex w adjacent to v for which k_w is false, test whether d_w is greater than distance(v,w). If it is, set d_w to distance(v,w) and set p_w to v.**

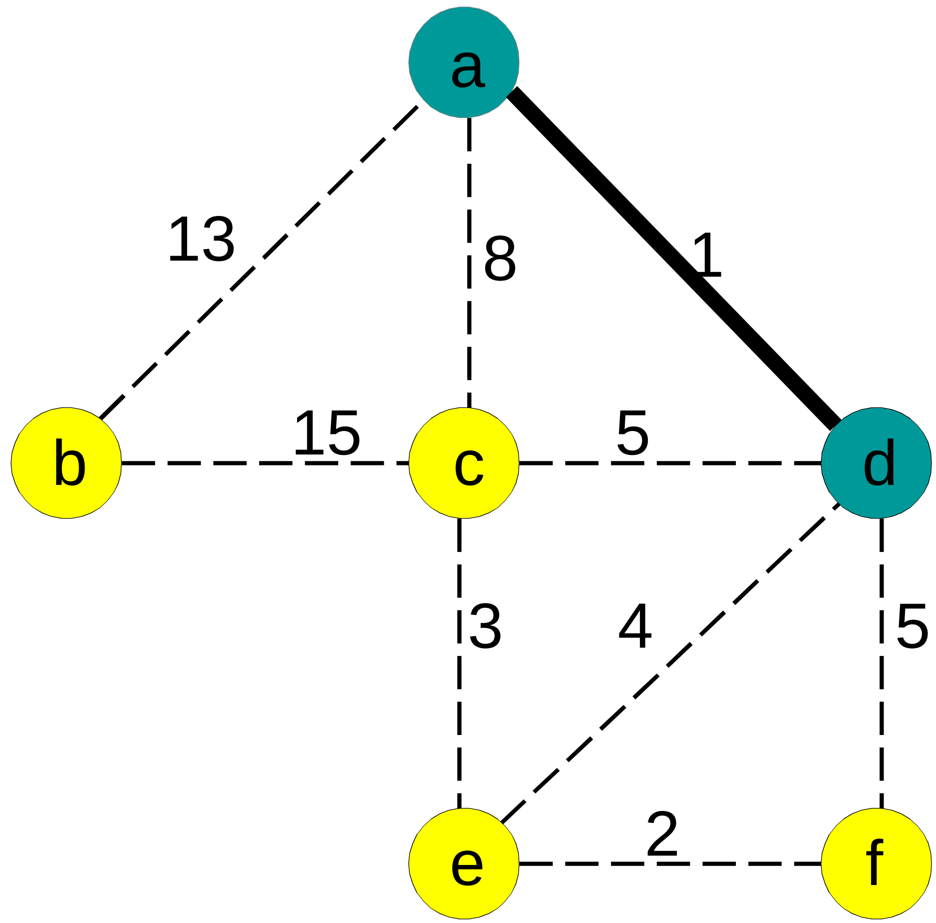
v	k_v	d_v	p_v
a	$\boldsymbol{F}$	0	-
b	$\boldsymbol{F}$	∞	
c	$\boldsymbol{F}$	∞	
d	$\boldsymbol{F}$	∞	
e	$\boldsymbol{F}$	∞	
f	$\boldsymbol{F}$	∞	



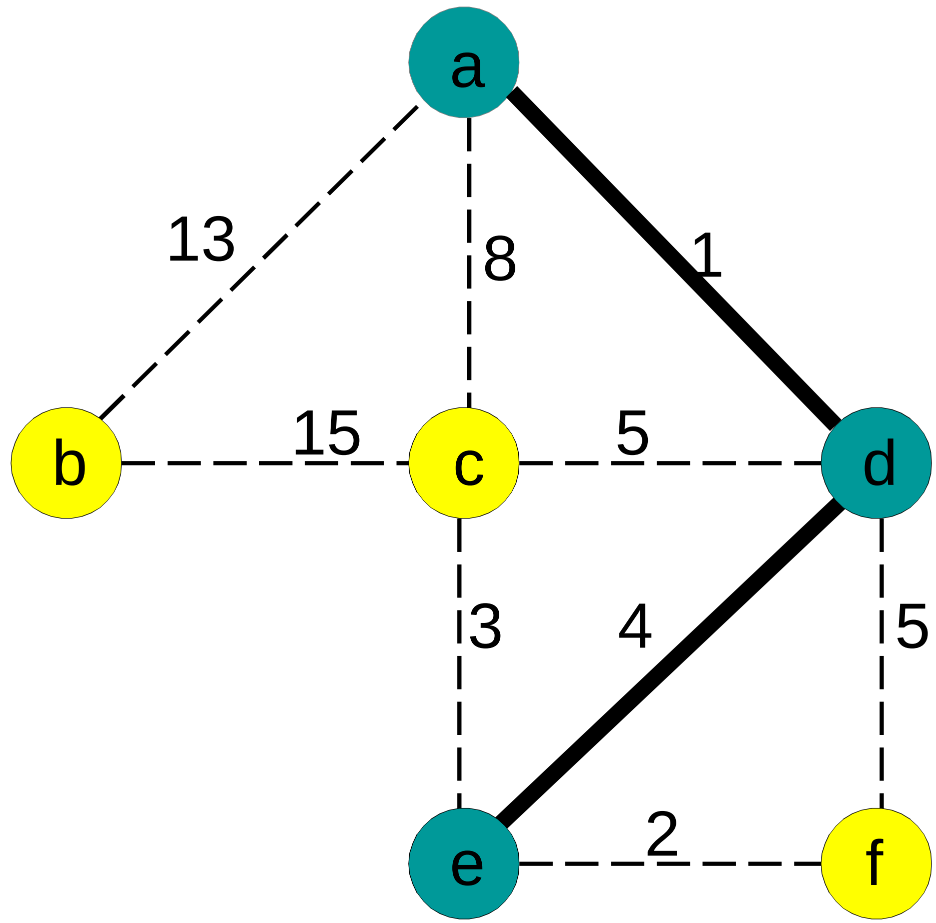
v	k_v	d_v	p_v
a	***T***	0	-
b	***F***	13	*a*
c	***F***	8	*a*
d	***F***	1	*a*
e	***F***	∞	
f	***F***	∞	



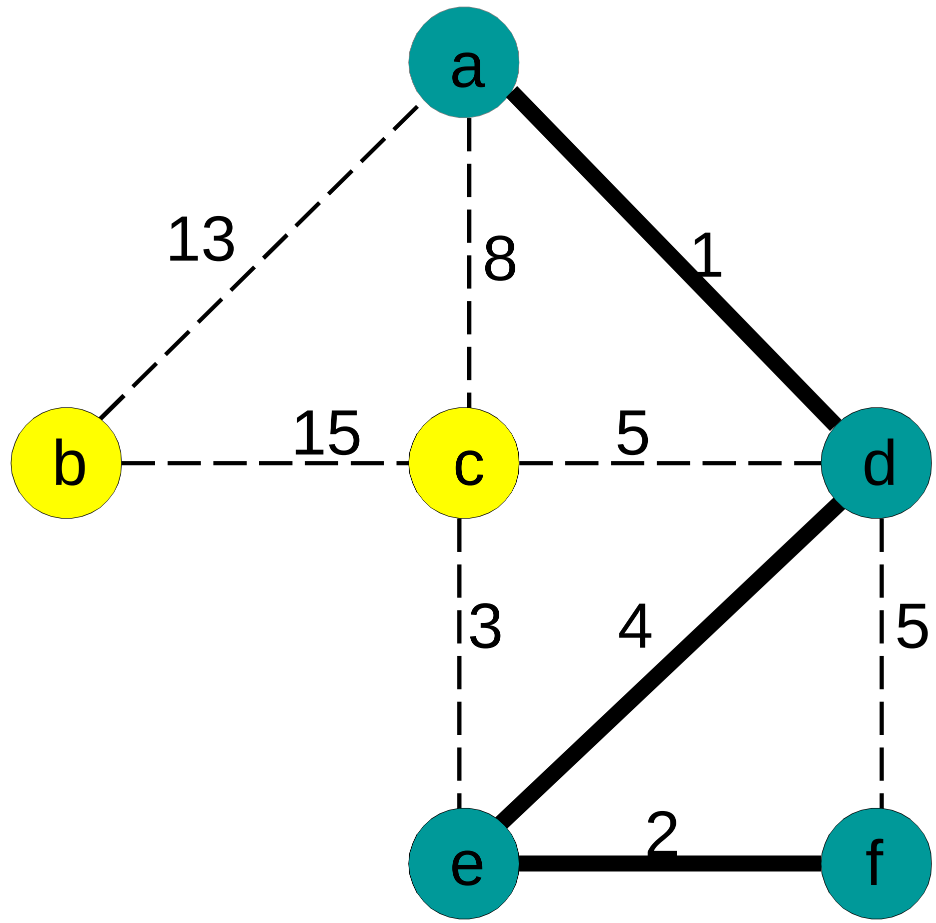
v	k_v	d_v	p_v
a	***T***	0	-
b	***F***	13	a
c	***F***	5	d
d	***T***	1	a
e	***F***	4	d
f	***F***	5	d



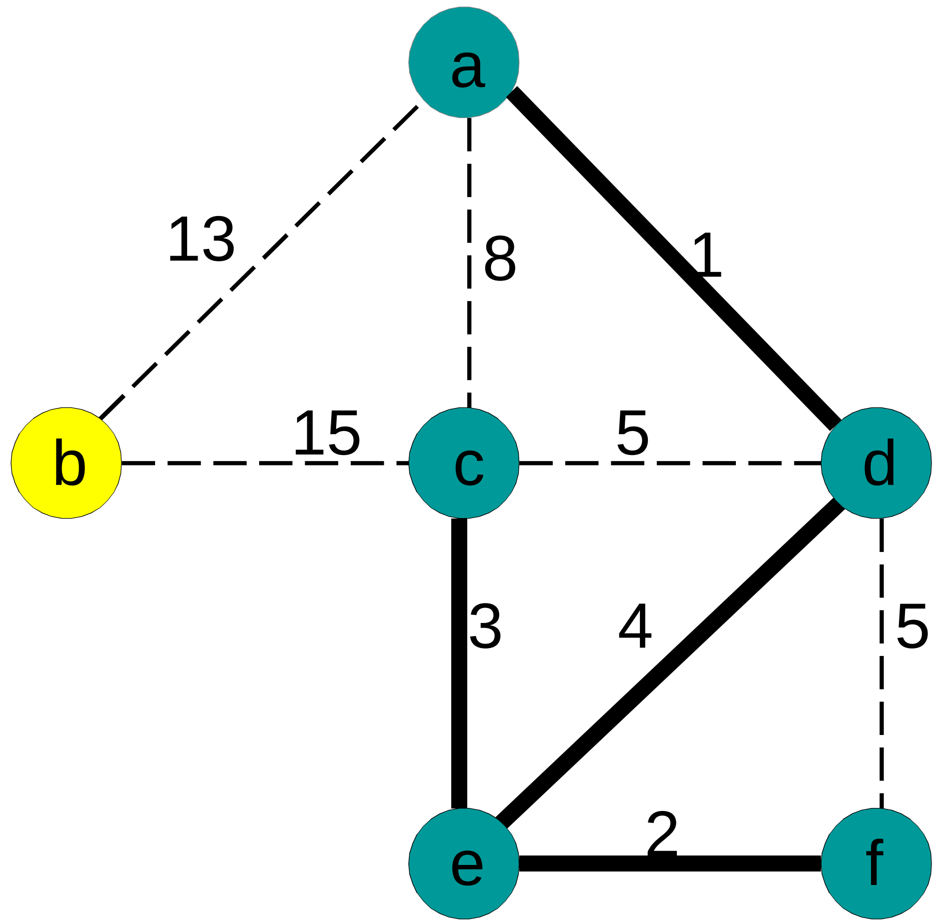
v	k_v	d_v	p_v
a	***T***	0	-
b	***F***	13	a
c	***F***	3	e
d	***T***	1	a
e	***T***	4	d
f	***F***	2	e



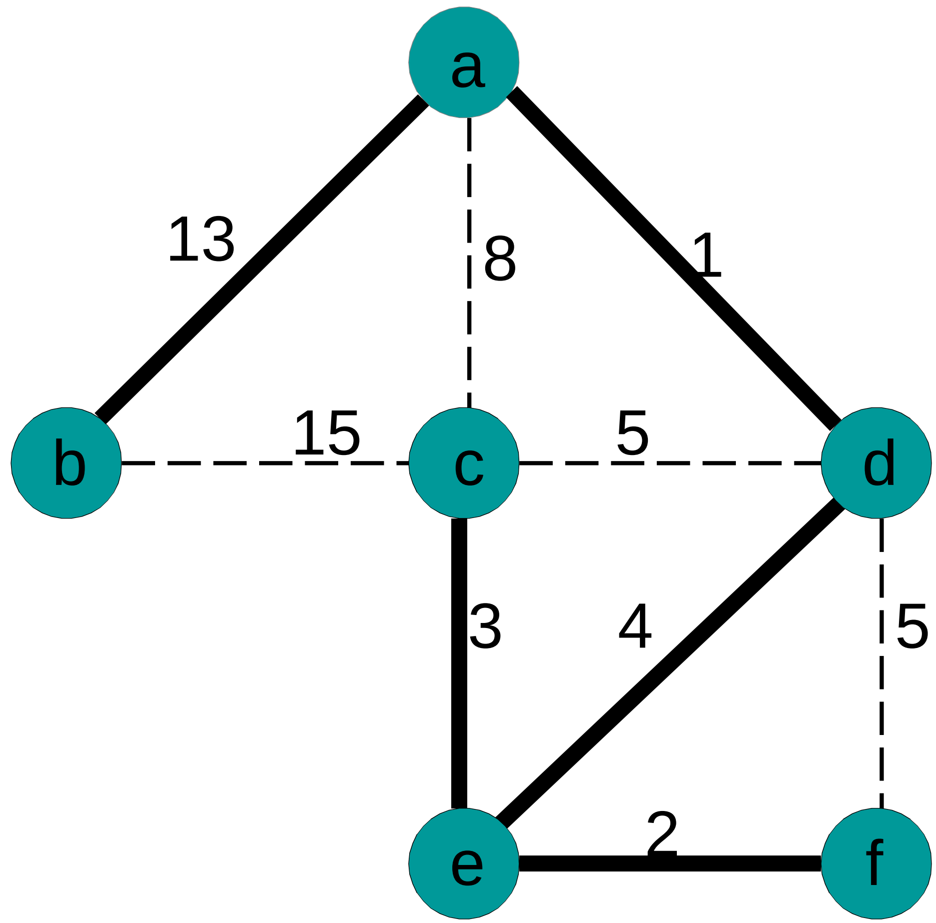
v	k_v	d_v	p_v
a	***T***	0	-
b	***F***	13	a
c	***F***	3	e
d	***T***	1	a
e	***T***	4	d
f	***T***	2	e



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

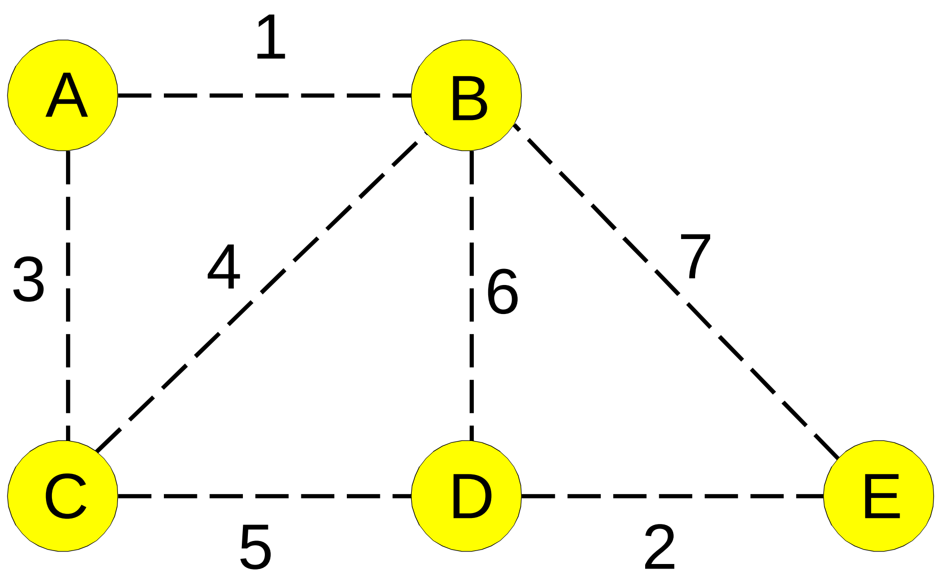


v	k_v	d_v	p_v
a	$\boldsymbol{T}$	0	
b	$\boldsymbol{T}$	13	a
c	$\boldsymbol{T}$	3	e
d	$\boldsymbol{T}$	1	a
e	$\boldsymbol{T}$	4	d
f	$\boldsymbol{T}$	2	e



MST this!

Using Prim's; start at node A



Algorithm – Linear Search

Repeat until every k_v is true:

1. **From the set of vertices for which k_v is false, select the vertex v having the smallest tentative distance d_v**
2. **Set k_v to true**
3. **For each vertex w adjacent to v for which k_w is false, test whether d_w is greater than distance(v,w). If it is, set d_w to distance(v,w) and set p_w to v.**

Complexity – Linear Search

|V| times

`Repeat until every` k_v `is true:`

1. `From the set of vertices for which` k_v `is false, select the vertex v having the smallest tentative distance` d_v

O(|V|)

2. `Set` k_v `to true`

O(1)

3. `For each vertex w adjacent to v for which` k_w `is false, test whether` d_w `is greater than distance(v,w). If it is, set` d_w `to distance(v,w) and set` p_w `to v.`

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

Algorithm – Heaps

Repeat until every k_v is true:

1. **From the set of vertices for which k_v is false, select the vertex v having the smallest tentative distance d_v**
2. **Set k_v to true**
3. **For each vertex w adjacent to v for which k_w is false, test whether d_w is greater than distance(v,w). If it is, set d_w to distance(v,w) and set p_w to v.**

Complexity – Heaps

|V| times

Repeat until every k_v is true:

1. **From the set of vertices for which k_v is false, select the vertex v having the smallest tentative distance d_v**

$O(\log |V|)$

2. **Set k_v to true**

O(1)

3. **For each vertex w adjacent to v for which k_w is false, test whether d_w is greater than distance(v,w). If it is, set d_w to distance(v,w) and set p_w to v.**

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

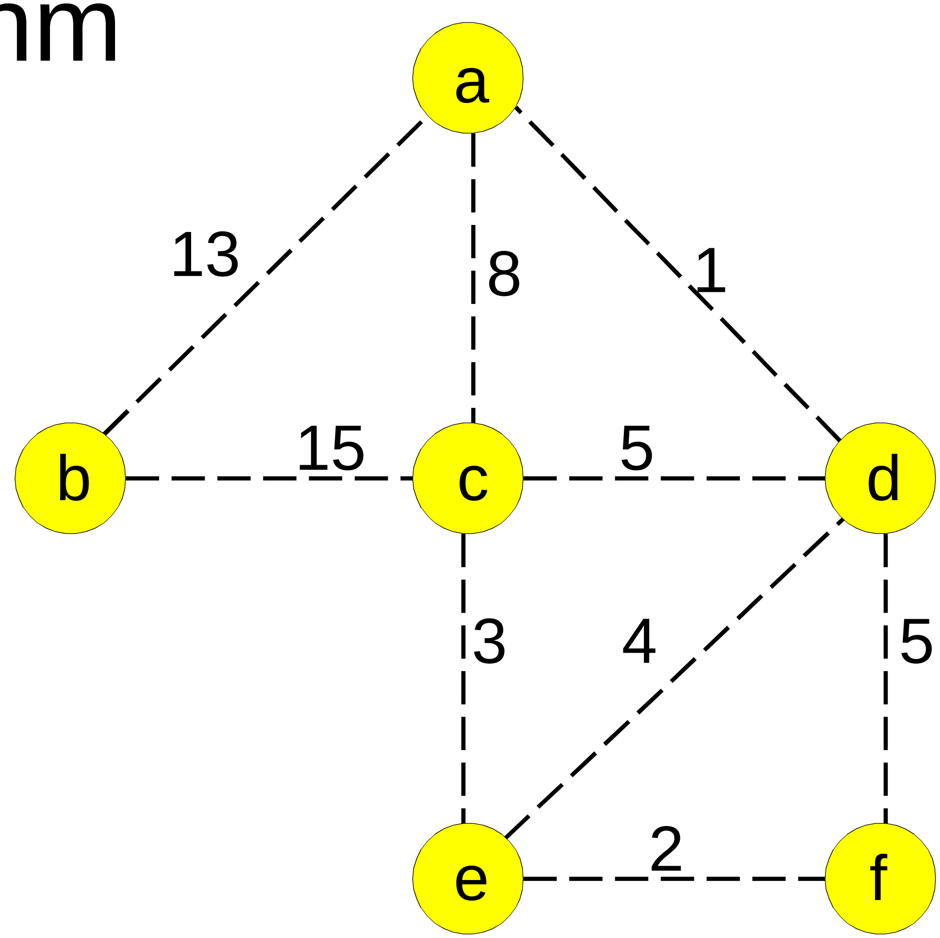
Prim: Asymptotic Complexity

- $O(V^2)$ for the simplest two-loop implementation; summary of complexity analysis: $V * (V + 1 + V) = 2 * V^2 + V$
- $O(E \log V)$ with heaps; summary of analysis: $V * \log V + E * \log V \approx E \log V$
- Same trade-offs for sparsity
- Optimizations for the two-loop implementation

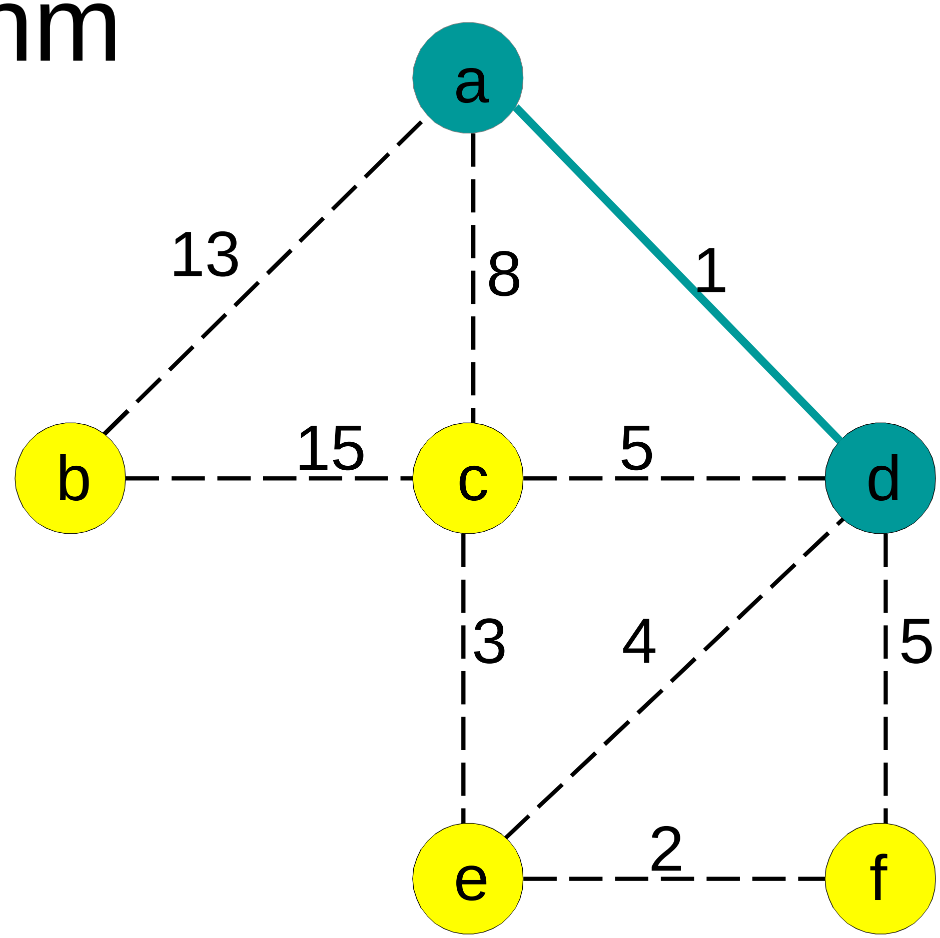
Kruskal’s Algorithm

- Greedy MST algorithm for edge-weighted, connected, *undirected* graph
  - Presort all edges: $O(E \log E) \approx O(E \log V)$ time
  - Try inserting in order of increasing weight
  - Some edges will be discarded so as not to create cycles
- Initial two edges may be disjoint
  - We are growing a <u>forest</u> (union of disjoint trees)

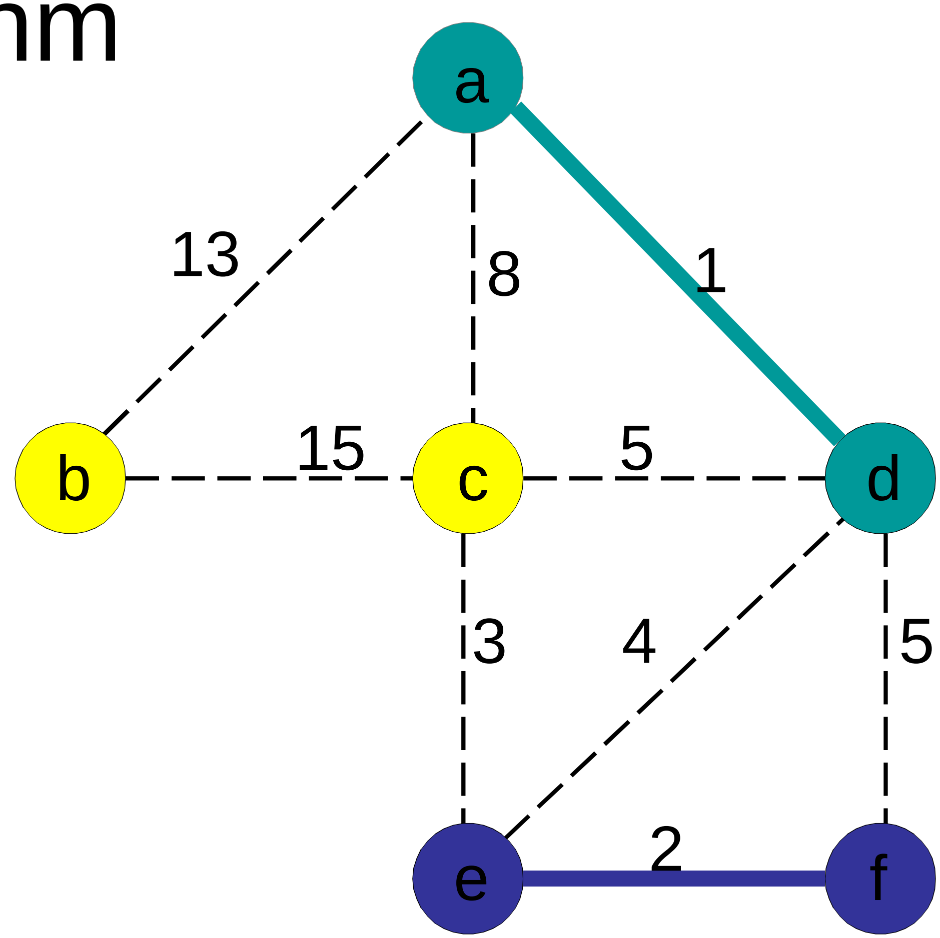
Kruskal's Algorithm



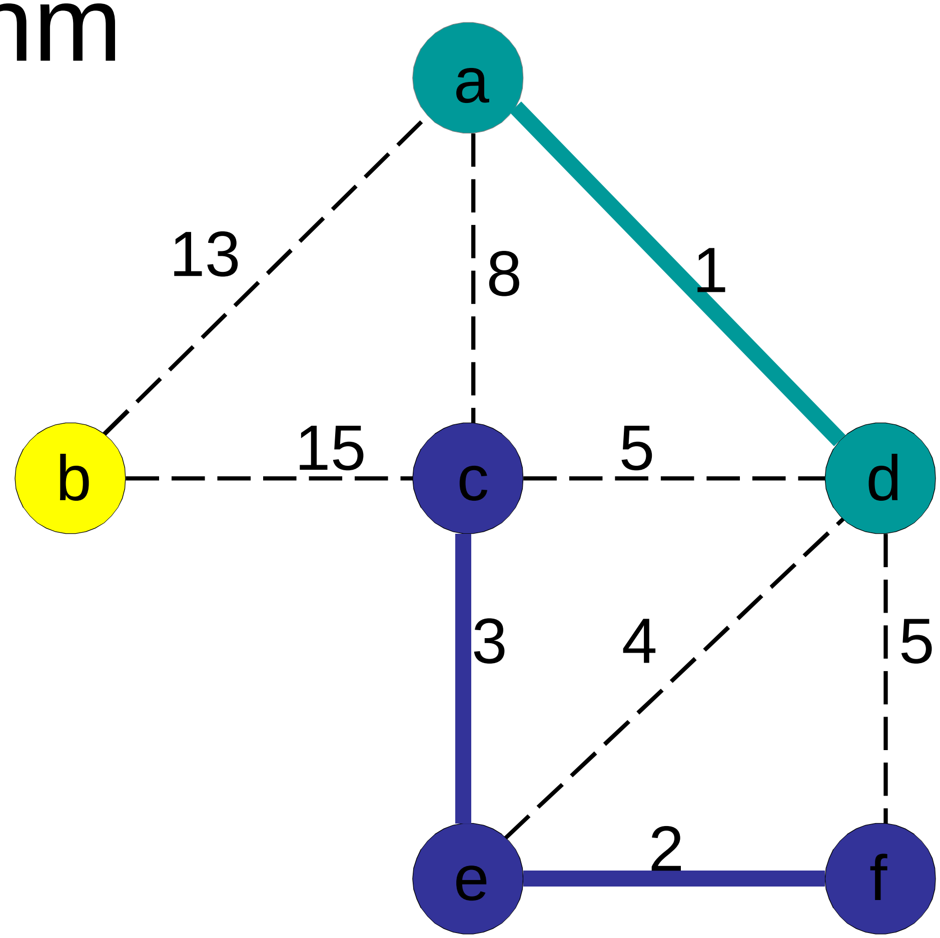
Kruskal's Algorithm



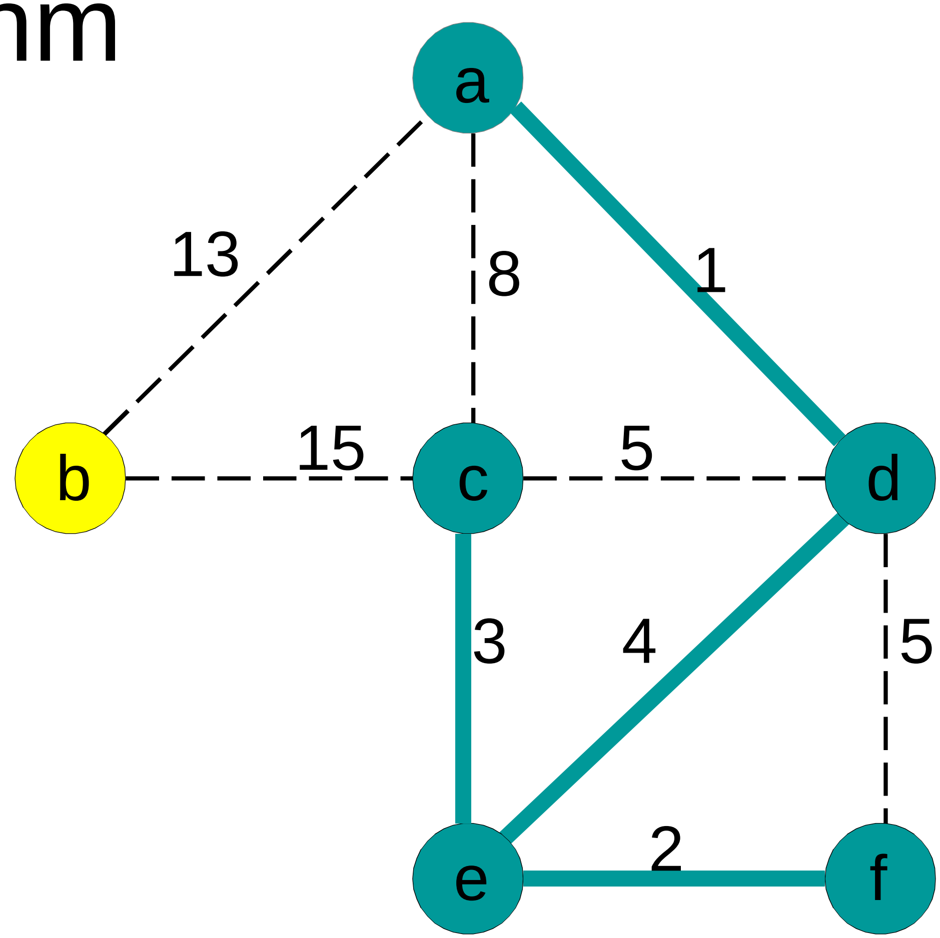
Kruskal’s Algorithm



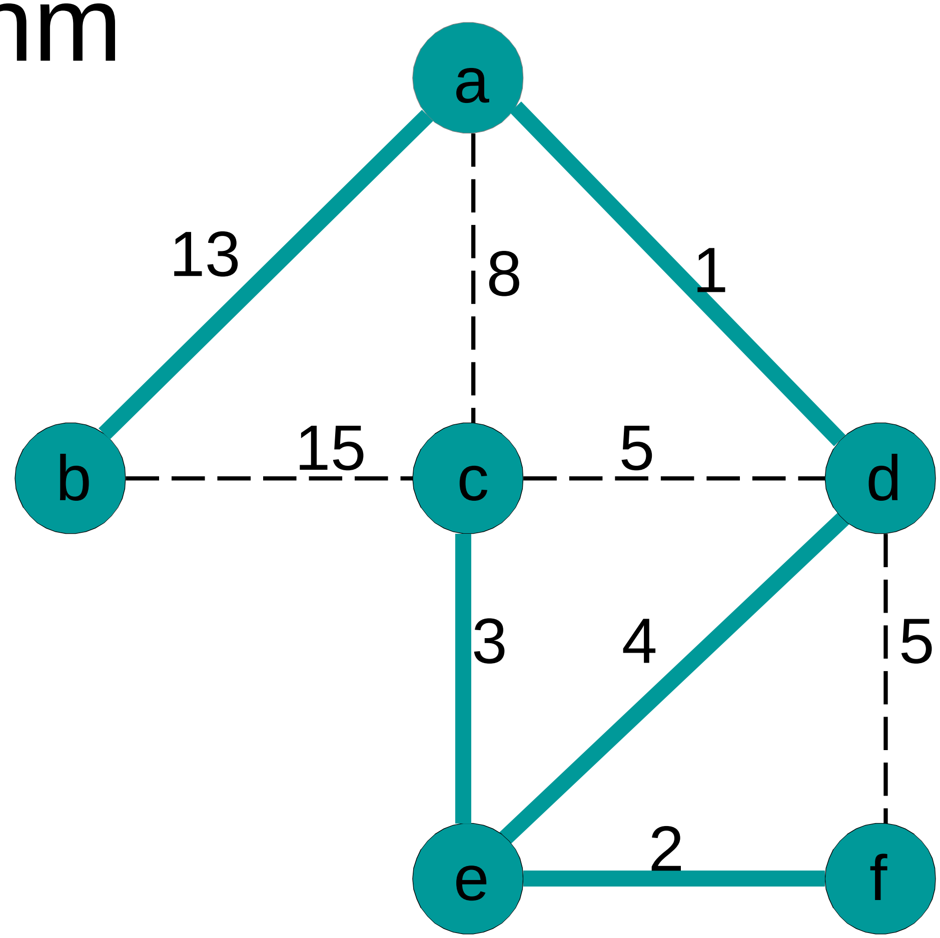
Kruskal's Algorithm



Kruskal’s Algorithm



Kruskal’s Algorithm



Kruskal: Complexity Analysis

- Sorting takes $E \log V$
  - 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 $\log E \approx \log V$
- Key idea: if vertices v_i and v_j are 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
- **For two cities *k* and *j*, would road *(k, j)* be redundant ?**

Union-Find Data Structure

- **Idea 1**: every *disjoint* set should have its unique representative (selected element)
  - Every set element *k* must know its representative *j*
- **Idea 2**: to tell if *k* and *m* are in the same set, *compare their representatives*
  - Redundancy check becomes fast
- Two main operations: *Union( )* and *Find( )*
- Lifecycle of a union-find data structure
  - Starts with *N* entirely disjoint elements
  - Ends up with all of them in one set

Union-Find Example

Everything is stored in an array

- A[j] is the representative of j

1 2 3 4 5 6 7 8 9 10

1. Connect 2 and 6

   1 <u>2</u> 3 4 5 <u>2</u> 7 8 9 10

2. Connect 8 and 6

   1 <u>2</u> 3 4 5 <u>2</u> 7 <u>2</u> 9 10

3. Connect 9 and 4

   1 <u>2</u> 3 <u>4</u> 5 <u>2</u> 7 <u>2</u> <u>4</u> 10

Making Union-Find Faster

- **Idea 3**: When performing union of two sets, <u>update the smaller set</u> (less work)
- Measure complexity of all unions throughout the lifecycle (together)
  - We call union <u>exactly</u> *N-1* times
  - If we connect to a disjoint element every time, it will take *N* time total (best case)
  - But merging large sets, say *$N/2$* and *$N/2$* elements, will take *$O(N)$* time for one union() – too slow!

Smarter Union-Find

- **Idea 4**: No need to store actual representative for each element, as long as can find it quickly
  - Each element knows someone who knows the representative (may need more steps)
  - *Union*() becomes very fast: one of representatives will need to know the other
  - *Find*() becomes slower
  - *Union*() cannot be faster than *Find*()

Another Optimization: Path Compression

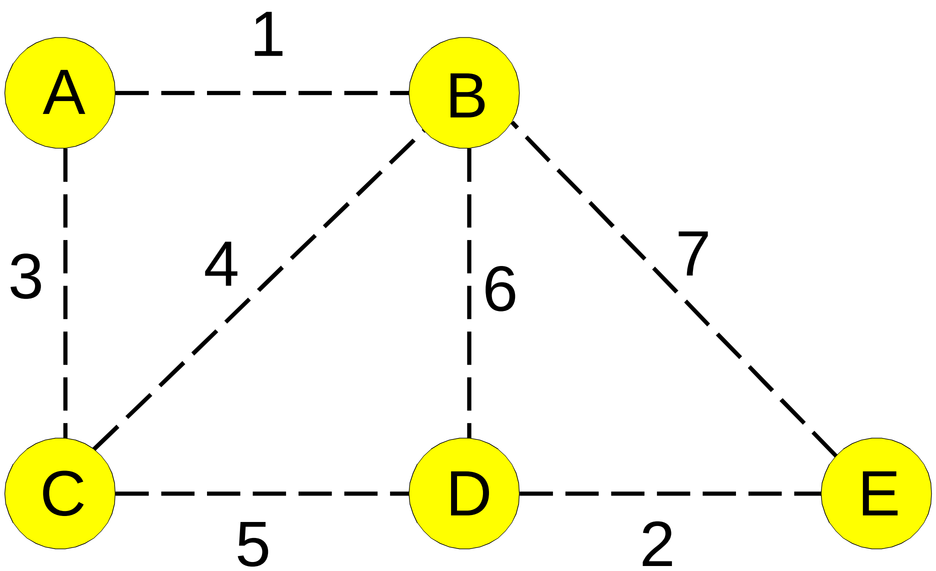
- So far, *Find*() was read-only
  - For element j, finds the representative k
  - Traverses other elements on the way (for which k is also the representative)
- **Idea 5:**

  We can tell j that it's representative is k
  - Same for other elements on path from $j \rightarrow k$
  - Doubles runtime of *Find*(), but same O()

Asymptotic Complexity?

- Must use amortized analysis over the life cycle of union-find
- Result is surprising
  - *$O(N\alpha(N))$*, where $\alpha()$ grows very slowly
  - $\alpha()$ is the reverse-Ackerman function
  - In practice, almost-linear-time performance
- Details taught in more advanced courses

MST this! (Kruskal's)



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 $O(E \log V)$
  - Kruskal: iteratively builds forest by adding minimal edges, $O(E \log V)$
- For dense *G*, use the two-loop Prim variant
- For sparse *G*, Kruskal is faster
  - Relies on the efficiency of sorting algorithms
  - Relies on the efficiency of union-find

Take-home MST Quiz

- Prove that Kruskal always finds an MST
- Prove that Prim always finds an MST
- Prove that Prim can start at any vertex
- Hint: revisit in-class MST quiz