Minimum Spanning Trees

CSE 332 Summer 2021

Instructor: Kristofer Wong

Teaching Assistants:

Alena Dickmann	Arya GJ	Finn Johnson
Joon Chong	Kimi Locke	Peyton Rapo
Rahul Misal	Winston Jodjana	

Announcements

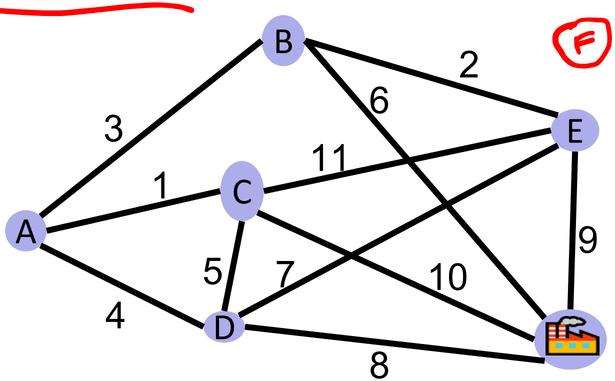
- Going to start importing grades from Gradescope to Canvas
  - Do not panic!
  - I'll be adjusting the points in canvas based on special cases in Canvas
    - If I didn't adjust something for you that I said I would, please let me know
- Today's and Monday's material not testable
  - Concepts that are casually thrown around, so you'll want to understand them

Lecture Outline

- **Minimum Spanning Tree**
  - Prim's Algorithm
  - Kruskal's Algorithm

Problem Statement

- Your friend at the electric company needs to connect all these cities to the power plant
- She knows the cost to lay wires between any pair of cities and wants the cheapest way to ensure electricity gets to every city

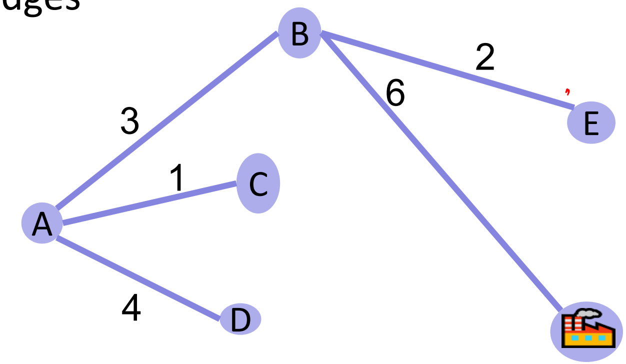


- Assume:
  - The graph is connected and undirected
  - *(In general, edge weights can be negative; just not in this example)*

Solution Statement

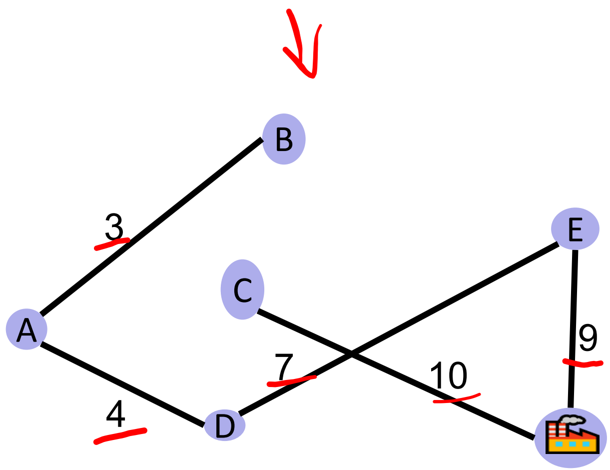
- We need a set of edges such that:
  - Every vertex touches at least one edge (“the edges **span** the graph”)
  - The graph using just those edges is **connected**
  - The total weight of these edges is **minimized**

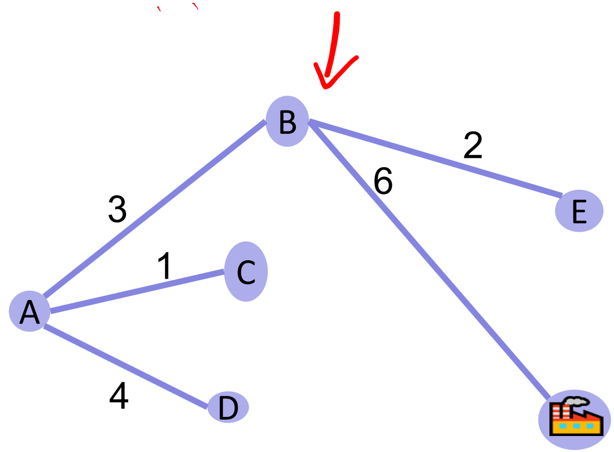
- *Claim*: The set of edges we pick never forms a cycle. Why?
  - V-1 edges is the exact number of edges to connect all vertices
  - Taking away 1 edge breaks connectiveness
  - Adding 1 edge makes a cycle



Solution Statement (v2)

- We need a ~~set of edges such that~~ Minimum Spanning Tree:
  - Every vertex touches at least one edge ("the edges **span** the graph")
  - The graph using just those edges is **connected**
  - The total weight of these edges is **minimized**





Minimum Spanning Trees

- Given an undirected graph G = (V,E), a minimum spanning tree is a graph G' = (V, E') such that:
  - E' is a subset of E
  - |E'| = |V| - 1
  - G' is connected
  - $\sum_{(u,v) \in E'} c_{uv}$ is minimal

Applications of MSTs

- Handwriting recognition
  - http://dspace.mit.edu/bitstream/handle/1721.1/16727/43551593-MIT.pdf;sequence=2

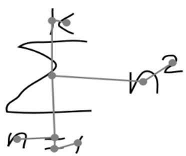
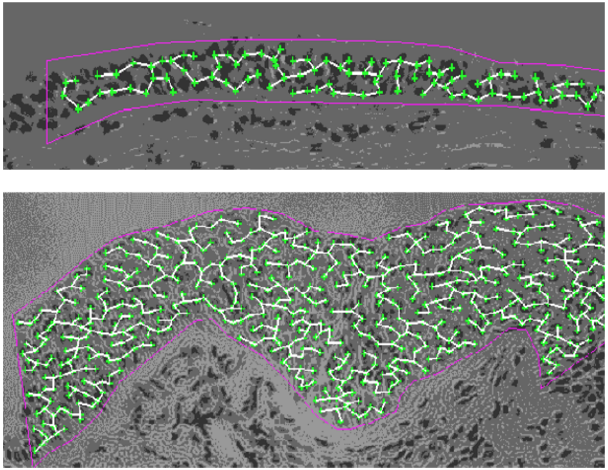


Figure 4-3: A typical minimum spanning tree

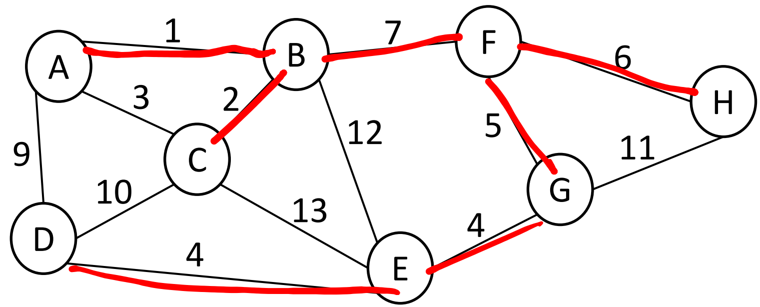
- Medical imaging
  - e.g. arrangement of nuclei in cancer cells

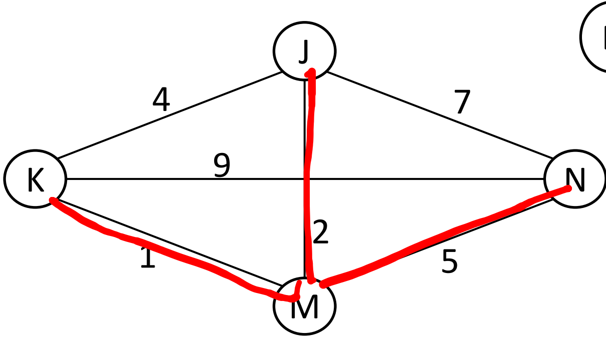


For more, see: http://www.ics.uci.edu/~eppstein/gina/mst.html

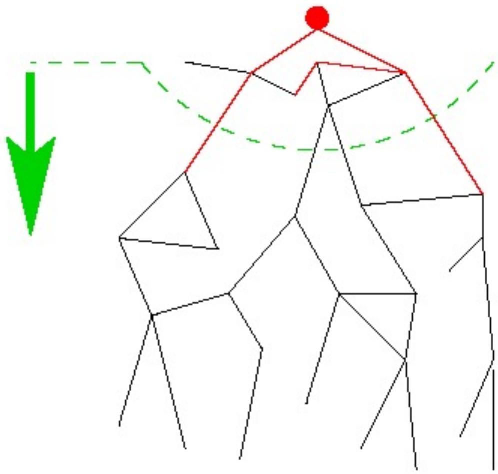
Exercise (not on Gradescope again..)

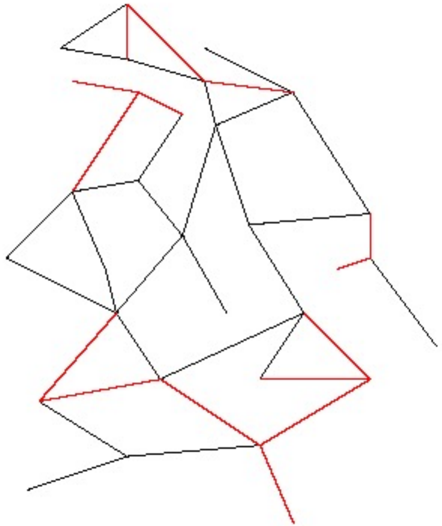
- Grab something to write with & something to write on!
- Draw the MST for each of the following:





MST Algorithms: Two Different Approaches





Prim's Algorithm

Almost identical to Dijkstra's
Start with one node, grow greedily

Kruskals's Algorithm

Completely different!
Start with a *forest* of MSTs, union them together
(Need a new data structure for this)

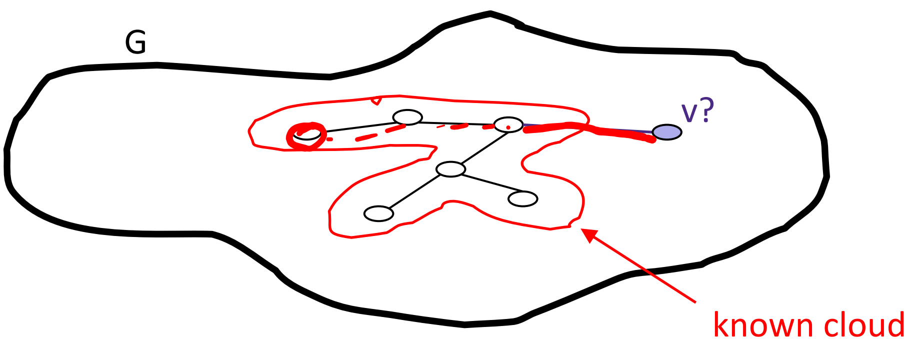
Lecture Outline

- Minimum Spanning Tree
  - **Prim's Algorithm**
  - Kruskal's Algorithm

Prim's Algorithm**

- *Intuition*: a vertex-based greedy algorithm
  - Builds MST by greedily adding vertices

- *Summary*: Grow a single tree by picking a vertex from the fringe that has the smallest cost
  - Unlike Dijkstra's, cost is the *edge weight* into the known set



*** This algorithm was developed in 1930 by Votěch Jarník, then independently rediscovered by Robert Prim in 1957 and then Dijkstra in 1959. It's also known as Jarník's, Prim-Jarník, or DJP*

Prim's Algorithm: Pseudocode

```
prims(Graph g) {
  foreach vertex v in g:
    v.distance = ∞
  start = g.getSomeArbitraryVertex()
  start.distance = 0

  mst = {}
  heap = buildHeap(g.vertices - {start})
  foreach vertex v in start.neighbors():
    v.distance = g.weight(start, v)
    v.previous = start
    heap.decreaseKey(v, v.distance)

  while (! heap.empty()):
    v = heap.deleteMin()
    mst.addEdge(v, v.previous)
    foreach edge (v, u) in g:
      d1 = v.distance
      d2 = u.distance
      if (d1 < d2):
        u.previous = v
}
```

Remember our 5-step pattern for a graph traversal?

Prim's Algorithm vs. Dijkstra's Algorithm (1 of 2)

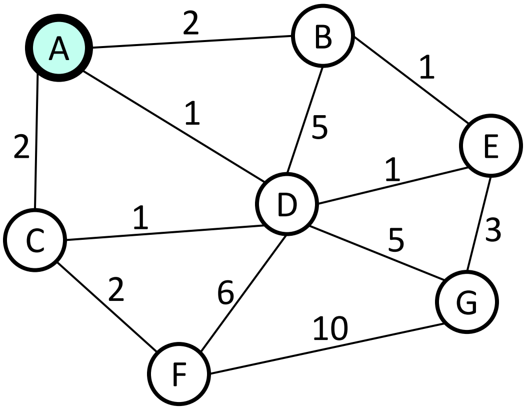
- Dijkstra's picks an unknown vertex with smallest *distance to the source*
  - ie, path weights
- Prim's picks an unknown vertex with smallest *distance to the known set*
  - i.e., edge weights
- Some differences in the initialization, but otherwise identical

Prim's Algorithm: Pseudocode

```
prims(Graph g) {
  foreach vertex v in g:
    v.distance = ∞
  start = g.getSomeArbitraryVertex()
  start.distance = 0

  mst = {}
  heap = buildHeap(g.vertices - {start})
  foreach vertex v in start.neighbors():
    v.distance = g.weight(start, v)
    v.previous = start
    heap.decreaseKey(v, v.distance)

  while (! heap.empty()):
    v = heap.deleteMin()
    mst.addEdge(v, v.previous)
    foreach edge (v, u) in g:
      d1 = v.distance
      d2 = u.distance
      if (d1 < d2):

        u.previous = v
}
```

```
dijkstra(Graph g, Vertex start) {
  foreach vertex v in g:
    v.distance = ∞

  start.distance = 0


  heap = buildHeap(g.vertices)





  while (! heap.empty()):
    v = heap.deleteMin()

    foreach edge (v, u) in g:
      d1 = v.dist + g.weight(v, u)
      d2 = u.dist
      if (d1 < d2):
        heap.decreaseKey(u, d1)
        u.previous = v
}
```

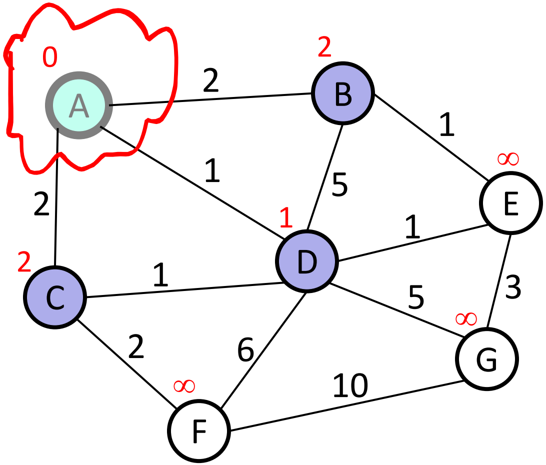
Prim's Algorithm: Example



Order Added to Known Set:

Vertex	Known?	Distance	Previous
A		∞	
B		∞	
C		∞	
D		∞	
E		∞	
F		∞	
G		∞	

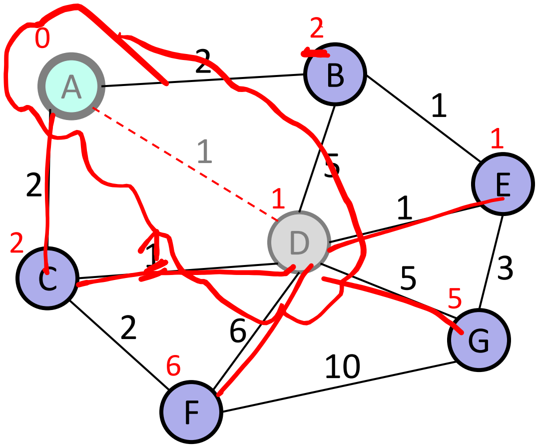
Prim's Algorithm: Example



Vertex	Known?	Distance	Previous
A	Y	0	\
B		2	A
C		2	A
D		1	A
E		∞	
F		∞	
G		∞	

<u>Order Added to Known Set:</u>
A

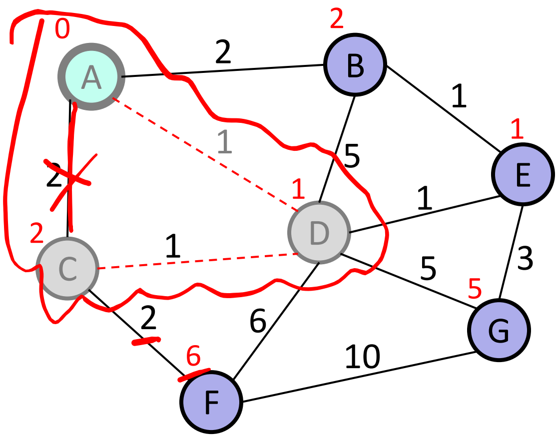
Prim's Algorithm: Example



Order Added to Known Set:
A, D

Vertex	Known?	Distance	Previous
A	Y	0	\
B		2	A
C		~~2~~ 1	D
D	Y	1	A
E		1	D
F		6	D
G		5	D

Prim's Algorithm: Example

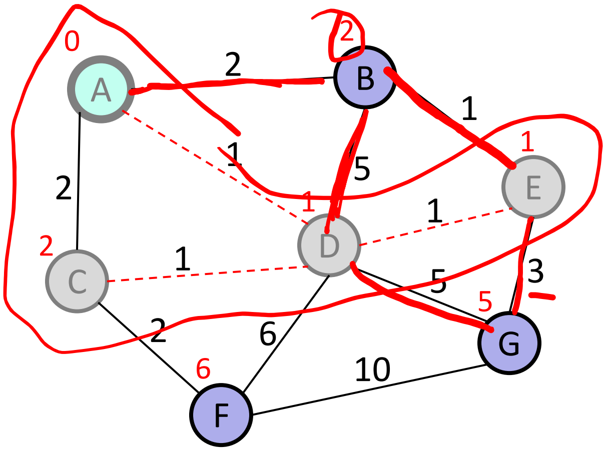


Order Added to Known Set:

A, D, C

Vertex	Known?	Distance	Previous
A	Y	0	\
B		2	A
C	Y	1	D
D	Y	1	A
E		1	D
F		2	C
G		5	D

Prim's Algorithm: Example

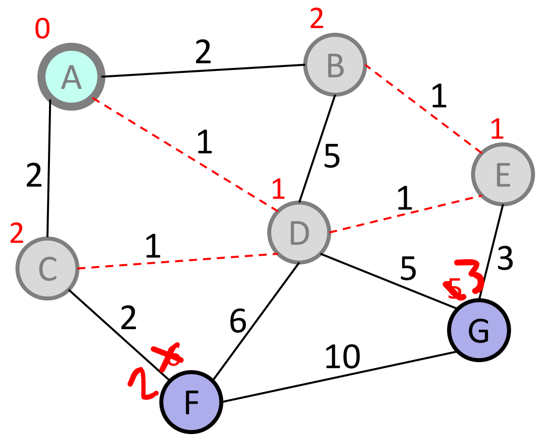


Vertex	Known?	Distance	Previous
A	Y	0	\
B		1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F		2	C
G		3	E

<u>Order Added to Known Set:</u>

A, D, C, E

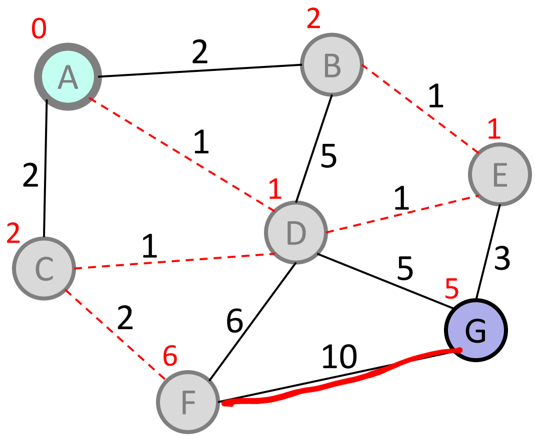
Prim's Algorithm: Example



Order Added to Known Set:
A, D, C, E, B

Vertex	Known?	Distance	Previous
A	Y	0	\
B	Y	1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F		2	C
G		3	E

Prim's Algorithm: Example

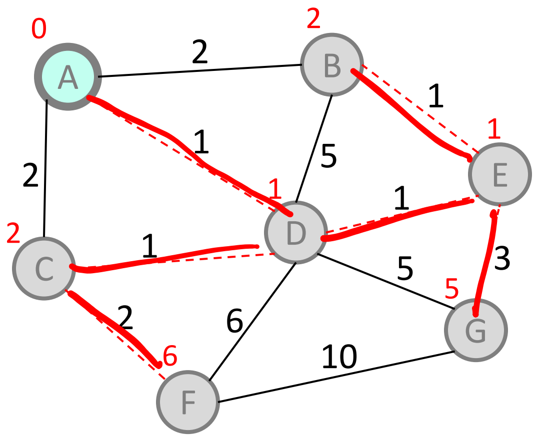


Vertex	Known?	Distance	Previous
A	Y	0	\
B	Y	1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F	Y	2	C
G		3	E

Order Added to Known Set:

A, D, C, E, B, F

Prim's Algorithm: Example



:D(one)

Total Cost: 9

Vertex	Known?	Distance	Previous
A	Y	0	\
B	Y	1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F	Y	2	C
G	Y	3	E

Order Added to Known Set:
A, D, C, E, B, F

Prim's Algorithm Visualizations

- Dijkstra's Visualization
  - https://www.youtube.com/watch?v=1oiQ0hrVwJk
  - Dijkstra's proceeds radially from its source, because it chooses edges by *path length from source*
- Prim's Visualization
  - https://www.youtube.com/watch?v=6uq0cQZOyoY
  - Prim's jumps around the MST-under-construction (the fringe), because it chooses edges by *edge weight* (there's no source)

Prim's Algorithm: Analysis

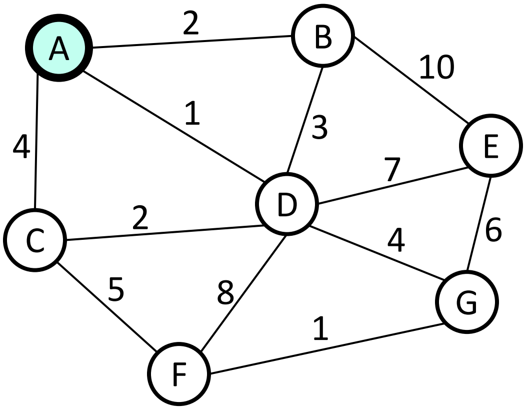
- Correctness:
  - A bit tricky to prove, but intuitively similar to Dijkstra
  - Proof on next slide, but left as an activity if you're curious

- Run-time:
  - Same as Dijkstra's! $O(|E|\log|V| + |V|\log|V|)$ using a priority queue
  - But since $E \in O(|V|^2)$, can also state as $O(|E|\log|V|)$

Prim's Algorithm: Correctness Proof

- ***Want to prove:*** If G is a connected, weighted graph with distinct edge weights, Prim's algorithm correctly finds an MST.
- ***Proof (credit: Stanford CS161, 13su); for more take CSE421!***
  - Let T be the spanning tree found by Prim's algorithm and T^* be the MST of G. We will prove $T = T^*$ by contradiction. Assume $T \neq T^*$. Therefore, $T - T^* \neq \emptyset$. Let (u, v) be any edge in $T - T^*$.
  - When (u, v) was added to T, it was the least-cost edge crossing some cut $(S, V - S)$. Since T^* is an MST, there must be a path from u to v in T^*. This path begins in S and ends in $V - S$, so there must be some edge (x, y) along that path where $x \in S$ and $y \in V - S$. Since (u, v) is the least- cost edge crossing $(S, V - S)$, we have $c(u, v) < c(x, y)$.
  - Let $T^{*\prime} = T^* \cup \{(u,v)\} - \{(x,y)\}$. Since (x,y) is on the cycle formed by adding (u, v), this means $T^{*\prime}$ is a spanning tree. However, $c(T^{*\prime}) = c(T^*) + c(u, v) - c(x, y) < c(T^*)$, contradicting that T^* is an MST.
  - We have reached a contradiction, so our assumption must have been wrong. Thus $T = T^*$, so T is an MST.

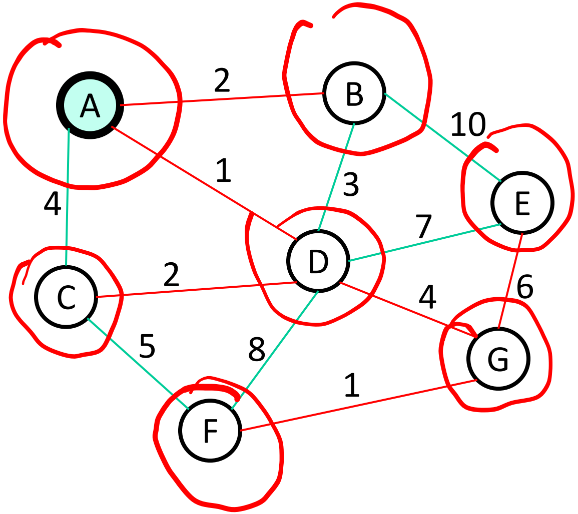
Exercise #2: Run through Prim's algorithm!



Vertex	Known?	Distance	Previous
A		∞	
B		∞	
C		∞	
D		∞	
E		∞	
F		∞	
G		∞	

Order Added to Known Set:

Exercise #2: Solution



Order Added to Known Set:

A, D, B, C, G, F, E

Vertex	Known?	Distance	Previous
A	Y	0	/
B	Y	2	A
C	Y	2	D
D	Y	1	A
E	Y	6	G
F	Y	4	G
G	Y	1	D

Lecture Outline

- Minimum Spanning Tree
  - Prim's Algorithm
  - **Kruskal's Algorithm**

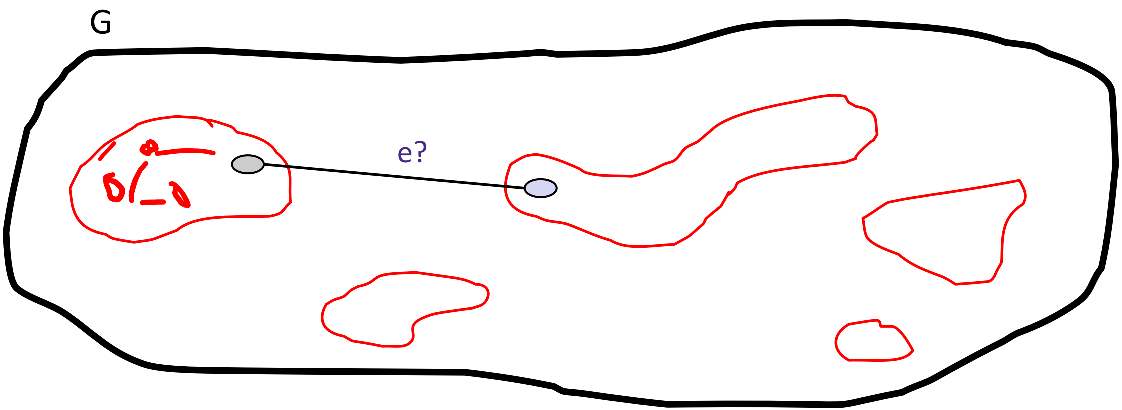
Kruskal's Algorithm: A Different Approach

- Prim's thinks vertex by vertex
  - Eg, add the closest vertex to the currently reachable set
- What if you think edge by edge instead?
  - Eg, start from the lightest edge; add it if it connects new things to each other (don't add it if it would create a cycle)

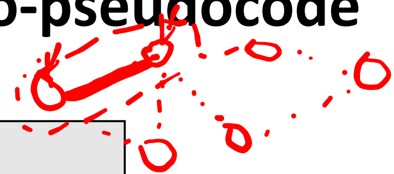
Kruskal's Algorithm

- *Intuition*: an edge-based greedy algorithm
  - Builds MST by greedily adding edges

- *Summary*: Start with a ***forest*** of MSTs, and successively connect them by adding edges; do not create a cycle



Kruskal's Algorithm: Pseudo-pseudocode



```
kruskals(Graph g) {
  edgesAccepted = 0
  mst = {}
  s = buildDisjointSets(g.vertices)
  edges = buildHeap(g.edges)

  while (edgesAccepted < NUM_VERTICES - 1):
    e = edges.deleteMin()
    u_id = s.find(e.u)
    v_id = s.find(e.v)
    if (u_id != v_id):
      mst.addEdge(e)
      s.unionSets(e.u, e.v)
      edgesAccepted++
}
```

Does this fit our 5-step pattern for a graph traversal?

What data structure is this?!?!

Aside: Disjoint Sets ADT

- The Disjoint Sets ADT has two operations:
  - Union
  - Find
  - AKA Union-Find ADT
- Applications include percolation theory (computational chemistry) and …. Kruskal's algorithm
- Simplifying assumptions
  - We can map elements to indices quickly
  - We know all the items in advance; they're all disconnected initially

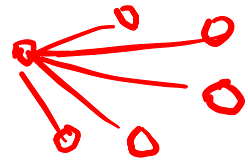
Disjoint Sets ADT

Disjoint Sets ADT. A collection of elements and sets of those elements.

- An element can only belong to a single set.
- Each set is identified by a unique id.
- Sets can be combined/ connected/ unioned.

- union(x, y): combines the set named x with the set named y; replaces x and y with (x ∪ y)
  - Given sets: {3,5,7} , {4,2,8}, {9}, {1,6}
    - Sets typically named after one of their elements
  - union(5,1) will union the set {3,5,7} with {1,6}
    - Result: {3,5,7,1,6} , {4,2,8}, {9}
  - Implementation: can be done in constant time
- find(e): gets the name of the element's set
  - Given sets: {3,5,7}, {4,2,8}, {9}, {1,6}
  - find(1) returns 1
  - find(7) returns 5
  - Implementation: can be amortized constant time with worst case O(logn) for an individual find operation

Kruskal's Algorithm: Pseudocode



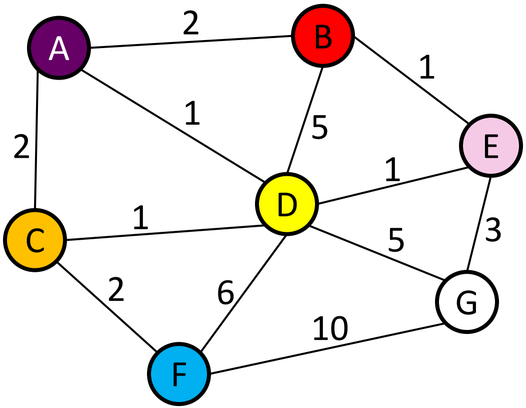
```
kruskals(Graph g) {
  edgesAccepted = 0
  mst = {}
  s = buildDisjointSets(g.vertices)
  edges = buildHeap(g.edges)

  while (edgesAccepted < NUM_VERTICES - 1):
    e = edges.deleteMin()
    u_id = s.find(e.u)
    v_id = s.find(e.v)
    if (u_id != v_id):
      mst.addEdge(e)
      s.unionSets(e.u, e.v)
      edgesAccepted++
}
```

- $|E|$ deleteMin()s
- $2|E|$ find()s
- $|V|$ union()s

Runtime: $|E|(\log|E| + 2\log|V| + 1) + |V|(1 + 1 + 1) \in O(|E|\log|V| + |E|\log|E|)$

Note: we know $|E| <= |V|^2$, so $\log|E| <= 2\log|V|$. Therefore, $|E|\log|V| + |E|\log|E| <= 3|E|\log|V|$, so the runtime can be simplified to $\mathbf{O(|E|\log|V|)}$

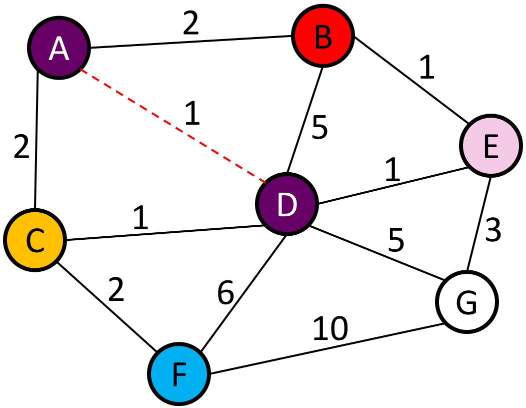
Kruskal's Algorithm: Example



Weight	Edges
1	(A,D), (C,D), (B,E), (D,E)
2	(A,B), (C,F), (A,C)
3	(E,G)
5	(D,G), (B,D)
6	(D,F)
10	(F,G)

MST:

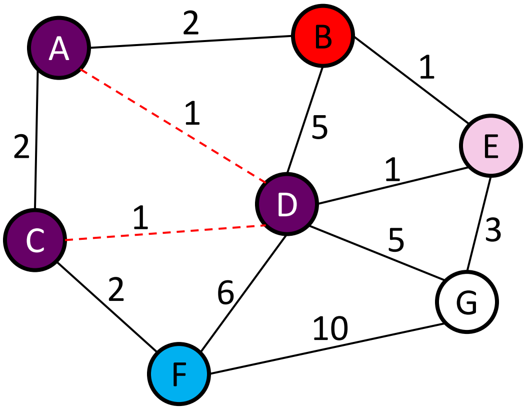
Kruskal's Algorithm: Example



Weight	Edges
1	*(A,D),* (C,D), (B,E), (D,E)
2	(A,B), (C,F), (A,C)
3	(E,G)
5	(D,G), (B,D)
6	(D,F)
10	(F,G)

MST:
(A, D)

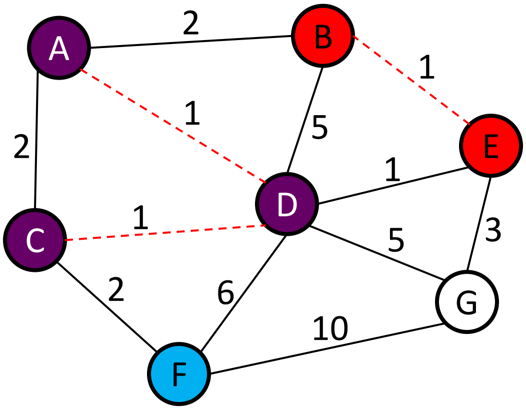
Kruskal's Algorithm: Example



Weight	Edges
1	*(A,D), (C,D),* (B,E), (D,E)
2	(A,B), (C,F), (A,C)
3	(E,G)
5	(D,G), (B,D)
6	(D,F)
10	(F,G)

MST:
(A, D), (C, D)

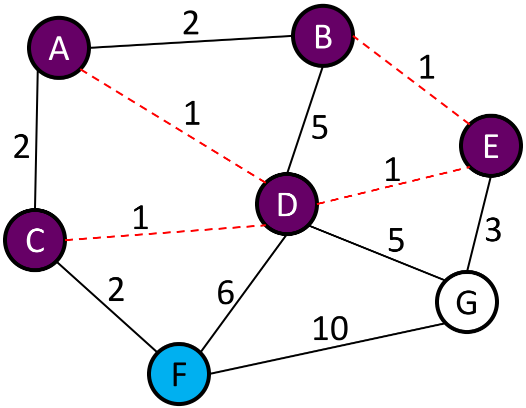
Kruskal's Algorithm: Example



Weight	Edges
1	*(A,D), (C,D), (B,E),* (D,E)
2	(A,B), (C,F), (A,C)
3	(E,G)
5	(D,G), (B,D)
6	(D,F)
10	(F,G)

MST:
(A, D), (C, D), (B, E)

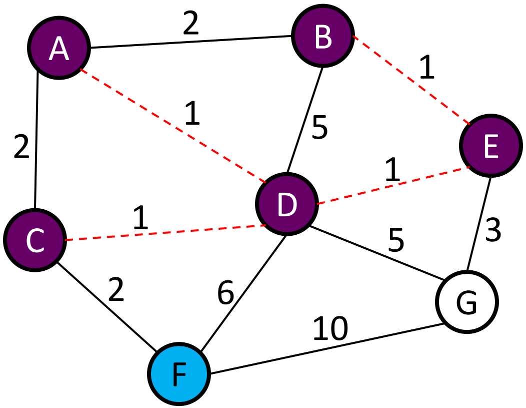
Kruskal's Algorithm: Example



Weight	Edges
1	*(A,D), (C,D), (B,E), (D,E)*
2	(A,B), (C,F), (A,C)
3	(E,G)
5	(D,G), (B,D)
6	(D,F)
10	(F,G)

MST:
(A, D), (C, D), (B, E), (D, E)

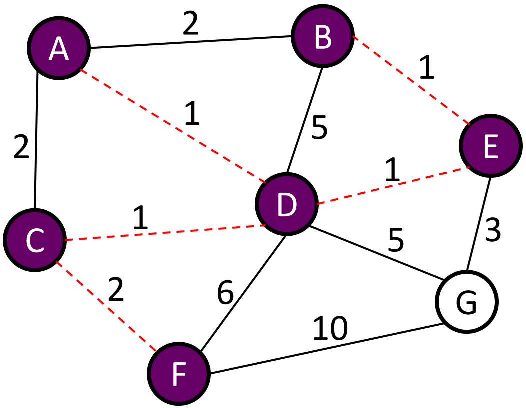
Kruskal's Algorithm: Example



Weight	Edges
1	*(A,D), (C,D), (B,E), (D,E)*
2	*(A,B),* (C,F), (A,C)
3	(E,G)
5	(D,G), (B,D)
6	(D,F)
10	(F,G)

<u>MST</u>:
(A, D), (C, D), (B, E), (D, E)

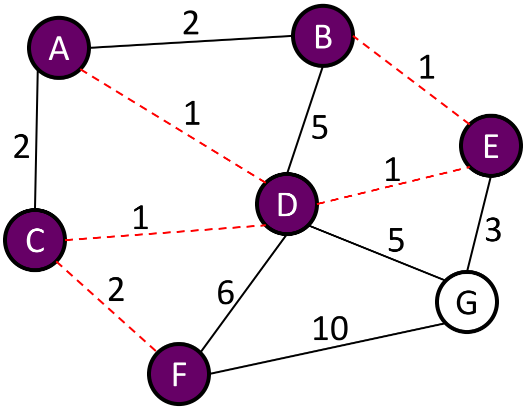
Kruskal's Algorithm: Example



Weight	Edges
1	*(A,D), (C,D), (B,E), (D,E)*
2	*(A,B), (C,F),* (A,C)
3	(E,G)
5	(D,G), (B,D)
6	(D,F)
10	(F,G)

MST:
(A, D), (C, D), (B, E), (D, E), (C, F)

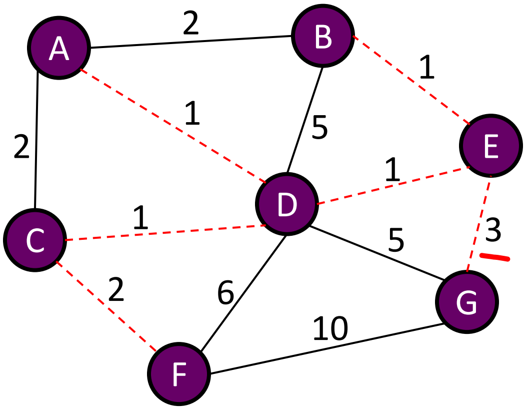
Kruskal's Algorithm: Example



Weight	Edges
1	*(A,D), (C,D), (B,E), (D,E)*
2	*(A,B), (C,F), (A,C)*
3	(E,G)
5	(D,G), (B,D)
6	(D,F)
10	(F,G)

MST:
(A, D), (C, D), (B, E), (D, E), (C, F)

Kruskal's Algorithm: Example



Yay!

Total Cost: 9

Weight	Edges
1	*(A,D), (C,D), (B,E), (D,E)*
2	*(A,B), (C,F), (A,C)*
3	*(E,G)*
5	(D,G), (B,D)
6	(D,F)
10	(F,G)

MST:
(A, D), (C, D), (B, E), (D, E), (C, F), (E, G)

Kruskal's Algorithm Visualizations

- Prim's Visualization
  - https://www.youtube.com/watch?v=6uq0cQZOyoY
  - Prim's jumps around the fringe, adding ~~edges~~ vertices by edge weight
- Kruskal's Visualization:
  - https://www.youtube.com/watch?v=ggLyKfBTABo
  - Kruskal's jumps around the graph – not just the fringe – because it chooses edges by edge weight independent of the "tree under construction"

Kruskal's Algorithm: Correctness

- Kruskal's algorithm is clever, simple, and efficient
  - But does it generate a minimum spanning tree?
- *First*: it generates a spanning tree
  - *Intuition*: Original graph was connected; we kept edges that didn't create a cycle
  - *Proof by contradiction*:
    - Suppose (u, v) is not in Kruskal's result
    - Then there's a path from u to v in the original graph with a *cheaper* edge we could add without creating a cycle
    - But Kruskal would have added that edge. **Contradiction!**
- *Second*: there is no spanning tree with lower total cost
  - Requires a more complex proof by Induction & Contradiction
  - Won't provide in a slide (relies on graph properties we won't cover)
  - Happy to prove in OH if you're curious; again, take CSE 421!

Summary

- Minimum Spanning Trees are a subset of the edges in an undirected connected graph

- Prim's looks a lot like the vertex-based graph traversals we've seen so far, except it uses *edge weight* instead of *path weight*
  - And since edge weights don't change during the algorithm's execution, we don't need a decreaseKey() operation

- Kruskal's is an edge-based graph traversal (which we haven't seen so far), but still uses *edge weight* to choose edges
  - Doesn't need decreaseKey() for the same reason
  - Needs an auxiliary ADT – the Disjoint Sets ADT – to speed up execution

up-tree