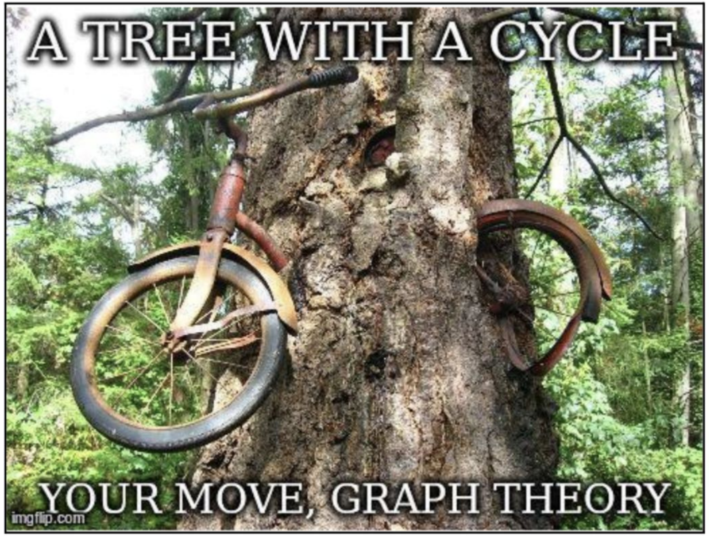
CS 1332R WEEK 14



Dijkstra's

Minimum Spanning Tree

Prim's

Kruskal's

ANNOUNCEMENTS

-

REFRESHER: Graphs

- ❑ We define a graph **(G)** by its **vertices** and **edges**.
- ❑ We define an edge **(e)** by the two vertices it connects **(u, v)** and its weight **(w)**.

V = the set of vertices

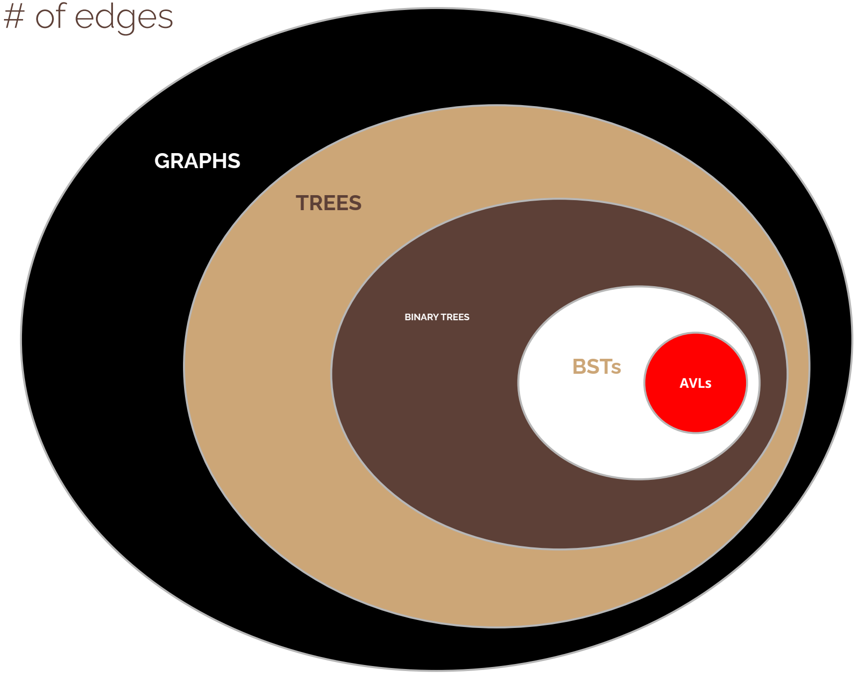
|V| = the # of vertices

E = the set of edges

|E| = the # of edges

RELEVANT TERMINOLOGY

ORDER(G) = |V|
SIZE(G) = |E|
Indegree(v) = # of edges going into a vertex
Outdegree(v) = # of edges going out of a vertex
Adjacent = describes two vertices that are connected by an edge
Connected Graph = every vertex has a path to every other vertex
Unconnected Graph = not all vertices have a path to every other vertex
Directed Graph = edges have a direction, they are distinct ordered pairs
Undirected Graph = edges do not have a direction, e(u, v) is the same as e(v, u)
Subgraph = a graph G' such that V' is a subset of V and E' is a subset of E
Cycle = a path with the same start and end vertex
Acyclic Graph = a graph containing no cycles
Tree = a minimally connected, acyclic graph



REFRESHER: Graphs

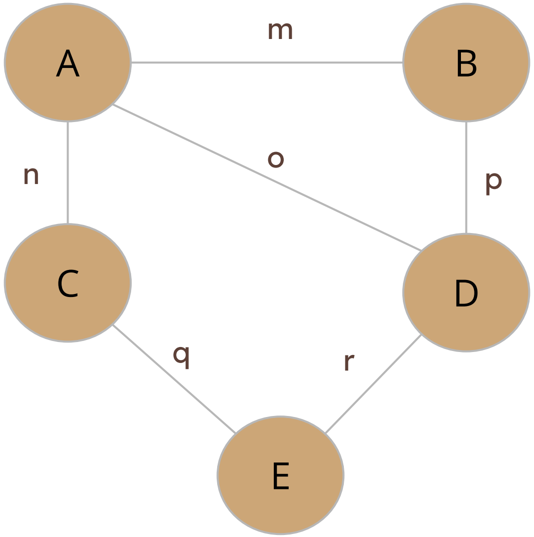
- We define a graph **(G)** by its **vertices** and **edges**.
- We define an edge **(e)** by the two vertices it connects **(u, v)** and its weight **(w)**.

V = the set of vertices

|V| = the # of vertices

E = the set of edges

|E| = the # of edges



HOW WE REPRESENT A GRAPH

1. Adjacency Matrix
   a. |V| x |V| 2D array
   b. Space: O(|V|^2)
   c. Matrix[v1, v2] = edge(v1, v2, w)

2. Adjacency List (***used in your homework***)
   a. Map of each vertex to its incident edges
   b. Space: O(|V| + |E|)
   c. In your homework, we map each vertex to a list of <u>VertexDistance</u> objects. The VertexDistance contains an adjacent vertex and the distance to that adjacent vertex, a.k.a the weight of the edge connecting the two vertices.

3. Edge Set (***used in your homework***)
   a. A set of all edges in the graph
   b. Space: O(|E|)

Dijkstra's Shortest Path Algorithm

- **PURPOSE**: Finding the shortest path between a start vertex and all other vertices in a graph with non-negative edge weights.

INTUITION: Dijkstra's is BFS generalized for a weighted graph.

STRUCTURES WE NEED

- **PriorityQueue<VertexDistance<T>> :** *a priority queue of VertexDistance objects ordered by the distance, which is the cumulative distance from the start vertex to the vertex*
- **Set<Vertex<T>>** : *a **visited set** containing vertices we have found the shortest path to*
- **Map<Vertex<T>, Integer>** : *a **distance map** of each vertex in the graph to its shortest distance to the start vertex*

← **This is our final solution.**

Dijkstra's: Implementation

STRUCTURES

- **PriorityQueue<VertexDistance<T>>** : *a priority queue of VertexDistance objects ordered by the distance, which is the cumulative distance from the start vertex to the vertex*
- **Set<Vertex<T>>** : *a **visited set** containing vertices we have found the shortest path to*
- **Map<Vertex<T>, Integer>** : *a **distance map** of each vertex in the graph to its shortest distance to the start vertex*

ALGORITHM

```
initialize s as start vertex
initialize pq, visitedSet, distanceMap
for all v in G, initialize distanceMap to +inf
pq.enqueue((s, 0))
while pq is not empty and visitedSet is not full:
    (u, d1) = pq.dequeue()
    if u is not in visitedSet:
        visitedSet.add(u)
        distanceMap.put(u, d1)
        for all (w, d2) adjacent to u and w not in visitedSet:
            pq.enqueue((w, d1 + d2))
```

2 termination conditions

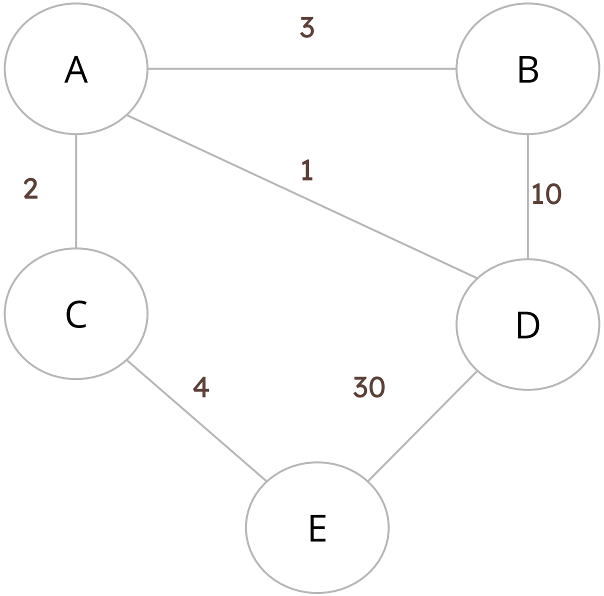
a vertex is considered visited once it has been dequeued from pq - GREEDY ALGORITHM

VertexDistance objects hold the <u>cumulative</u> distance to w

How do we get all VertexDistance objects adjacent to u?

`graph.getAdjList(u)`

Dijkstra's: Practice



DIAGRAMMING SETUP

DISTANCE MAP

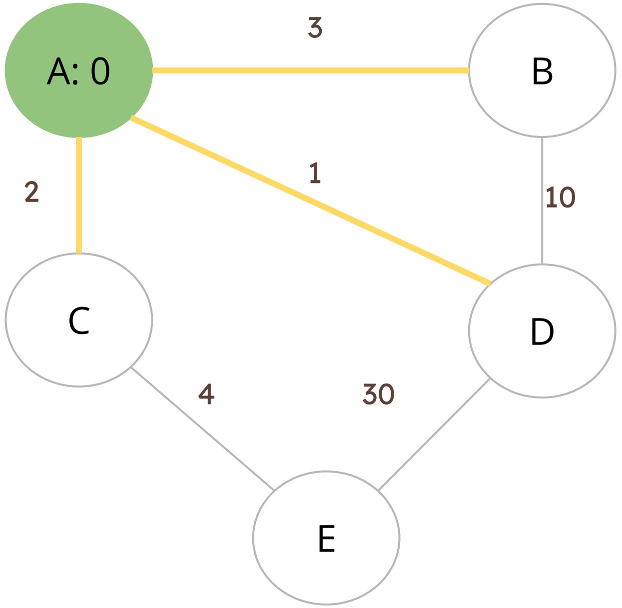
A	inf
B	inf
C	inf
D	inf
E	inf

VISITED SET

PRIORITY QUEUE

(A, 0)

Dijkstra's: Practice



DIAGRAMMING SETUP

DISTANCE MAP

A	0
B	inf
C	inf
D	inf
E	inf

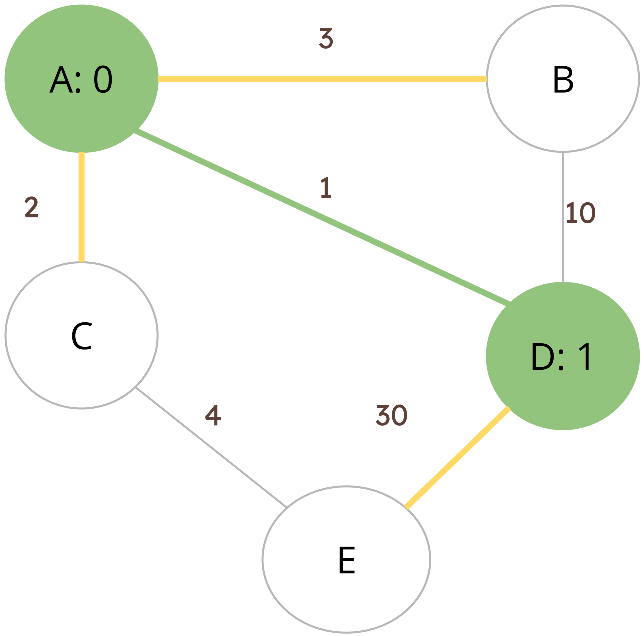
VISITED SET

A

PRIORITY QUEUE

~~(A, 0)~~
(B, 3)
(C, 2)
(D, 1)

Dijkstra's: Practice



DIAGRAMMING SETUP

DISTANCE MAP

A	0
B	inf
C	inf
D	1
E	inf

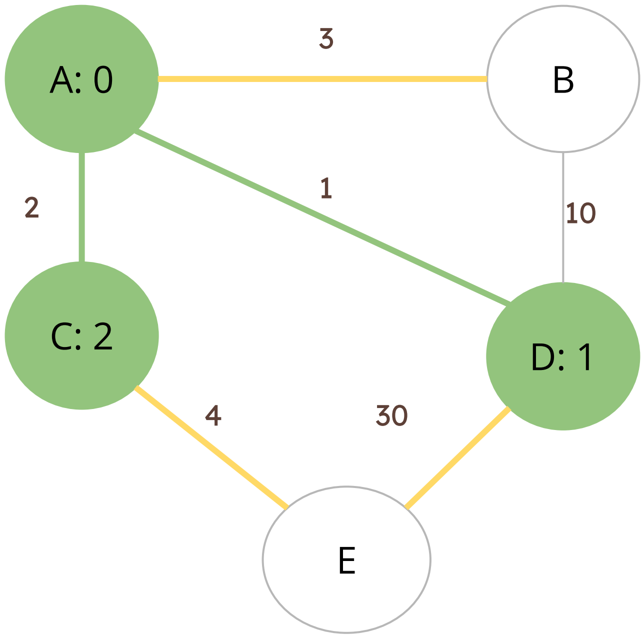
VISITED SET

A
D

PRIORITY QUEUE

~~(A, 0)~~
(B, 3)
(C, 2)
~~(D, 1)~~
(B, 11)
(E, 31)

Dijkstra's: Practice



DIAGRAMMING SETUP

DISTANCE MAP

A	0
B	inf
C	2
D	1
E	inf

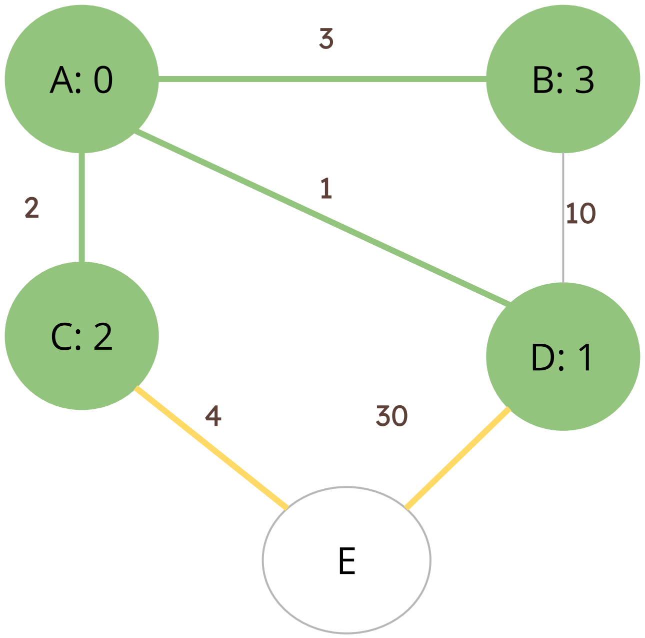
PRIORITY QUEUE

~~(A, 0)~~
(B, 3)
~~(C, 2)~~
~~(D, 1)~~
(B, 11)
(E, 31)
(E, 6)

VISITED SET

A
D
C

Dijkstra's: Practice



DIAGRAMMING SETUP

DISTANCE MAP

A	0
B	3
C	2
D	1
E	inf

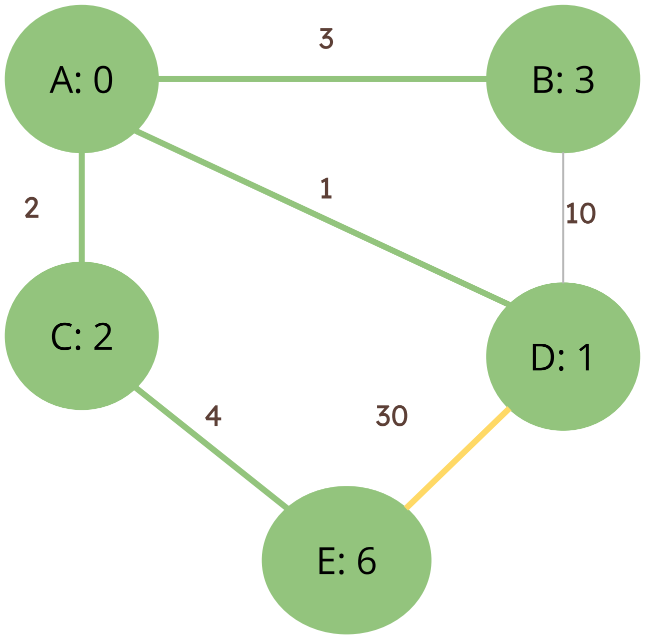
PRIORITY QUEUE

~~(A, 0)~~
~~(B, 3)~~
~~(C, 2)~~
~~(D, 1)~~
(B, 11)
(E, 31)
(E, 6)

VISITED SET

A
D
C
B

Dijkstra's: Practice



DIAGRAMMING SETUP

DISTANCE MAP

A	0
B	3
C	2
D	1
E	6

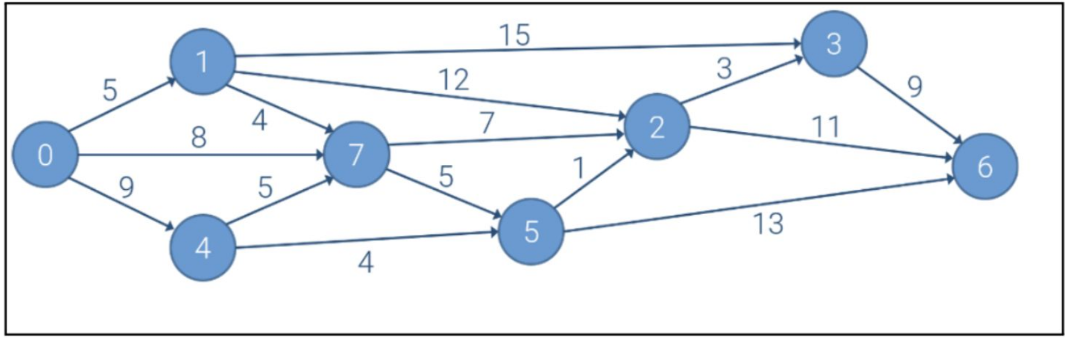
VISITED SET

A
D
C
B
E

PRIORITY QUEUE

~~(A, 0)~~
~~(B, 3)~~
~~(C, 2)~~
~~(D, 1)~~
(B, 11)
(E, 31)
~~(E, 6)~~

Dijkstra's: Practice



Vertex	Distance
0	0
1	5
2	14
3	17
4	9
5	13
6	25
7	8

DIAGRAMMING SETUP

DISTANCE MAP

0	inf
1	inf
2	inf
3	inf
4	inf
5	inf
6	inf
7	inf

PRIORITY QUEUE

(0, 0)

VISITED SET

Dijkstra's: Efficiencies

STRUCTURES

- **PriorityQueue<VertexDistance<T>>** : *a priority queue of VertexDistance objects ordered by the distance, which is the cumulative distance from the start vertex to the vertex*
- **Set<Vertex<T>>** : *a **visited set** containing vertices we have found the shortest path to*
- **Map<Vertex<T>, Integer>** : *a **distance map** of each vertex in the graph to its shortest distance to the start vertex*

ALGORITHM

```
initialize s as start vertex
initialize pq, visitedSet, distanceMap
for all v in G, initialize distanceMap to +inf
pq.enqueue((s, 0))
while pq is not empty and visitedSet is not full:
      (u, d1) = pq.dequeue()
      if u is not in visitedSet:
            visitedSet.add(u)
            distanceMap.put(u, d1)
            for all (w, d2) adjacent to u and w not in visitedSet:
                  pq.enqueue((w, d1 + d2))
```

- ❑ **Time: O(|E|log|E|)** - removing from the priority queue |E| times
- ❑ **Space: O(|E|)** - priority queue could contain all edges, we are guaranteed to not “reuse” edges

Minimum Spanning Trees (MST)

MORE TERMINOLOGY

Subgraph = a graph G'(V', E') is a subgraph of G(V, E) if V' and E' are subsets of V and E
Connected Graph = every vertex can reach every other vertex

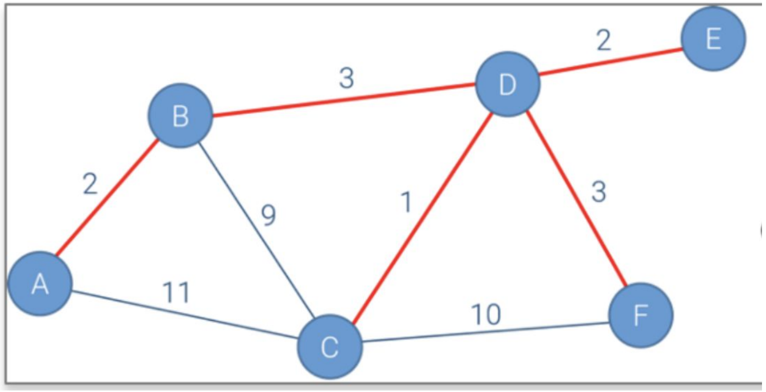
Tree = a connected, acyclic graph

Spanning Tree = a subgraph of an undirected graph containing all vertices and edges

Minimum Spanning Tree = a spanning tree of minimum edge weight - NOT UNIQUE

Can a disconnected graph have a spanning tree?

NO



|V| = 6
|E| = 8
How many undirected edges does the MST have?

An MST always has |V| - 1 undirected edges and 2 * (|V| - 1) directed edges.

Prim's

- **PURPOSE**: Finding the minimum spanning tree of an undirected graph given a starting vertex.

INTUITION: Prim's is Dijkstra's with a priority queue of edges.

STRUCTURES WE NEED

- **PriorityQueue<Edge<T>> :** *a priority queue of edges in the graph*
- **Set<Vertex<T>>** : *a **visited set** containing vertices already* ~~*added to the minimum spanning tree*~~
- **Set<Edge<T>>** : *an **edge set** representing our minimum spanning tree*

← **This is our final solution.**

Prim's: Implementation

STRUCTURES

- **PriorityQueue<Edge<T>>** : *a priority queue of edges in the graph*
- **Set<Vertex<T>>** : *a **visited set** containing vertices already added to the minimum spanning tree*
- **Set<Edge<T>>** : *an **edge set** representing our minimum spanning tree*

ALGORITHM

```
initialize s as start vertex
initialize pq, visitedSet, mst
visitedSet.add(s)
for all e(s, v) in G:
    pq.enqueue(e(s,v))
while pq is not empty and visitedSet and mst is not full:
    e(u, w) = pq.dequeue()
    if w is not in visitedSet:
        visitedSet.add(w)
        mst.add(e(u, w))
        for all e(w, x) and x not in visitedSet:
            pq.enqueue(e(w, x))
```

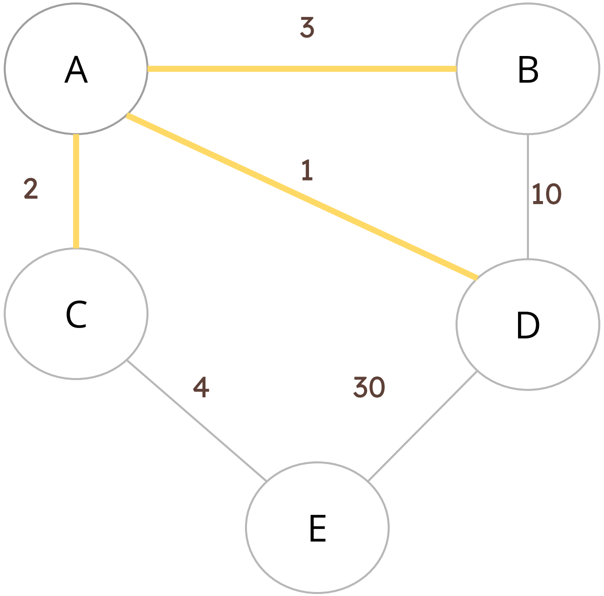
3 termination conditions

a vertex is considered visited once it has been dequeued from pq - GREEDY ALGORITHM

How do we know that the MST is valid after we exit the while loop?

If the size of the MST is equal to |V| - 1 or 2 * (|V| - 1) for undirected edges

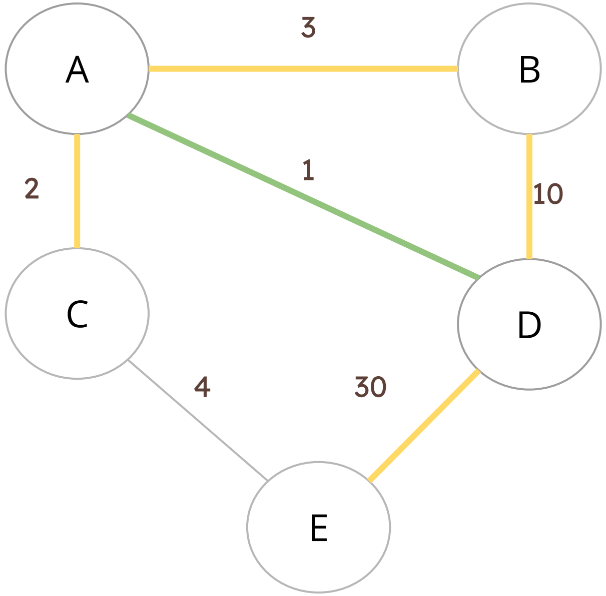
Prim's: Practice



DIAGRAMMING SETUP

PRIORITY QUEUE	VISITED SET	EDGE SET
(A, B, 3)	A	
(A, C, 2)		
(A, D, 1)		

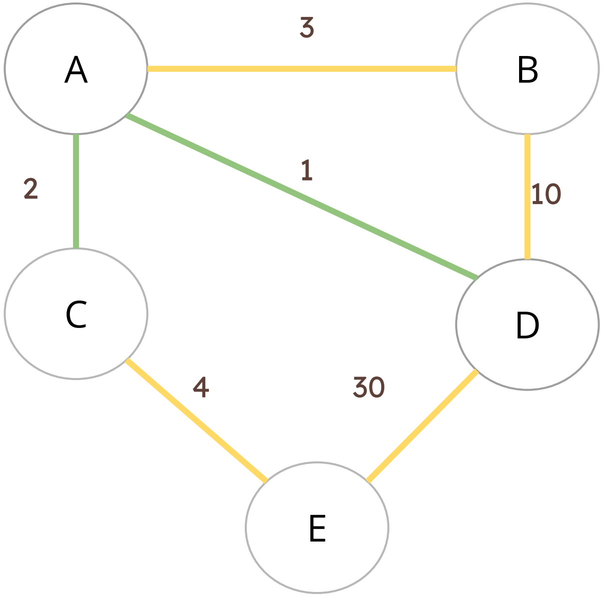
Prim's: Practice



DIAGRAMMING SETUP

PRIORITY QUEUE	VISITED SET	EDGE SET
(A, B, 3)	A	(A, D, 1)
(A, C, 2)	D	
~~(A, D, 1)~~		
(D, B, 10)		
(D, E, 30)		

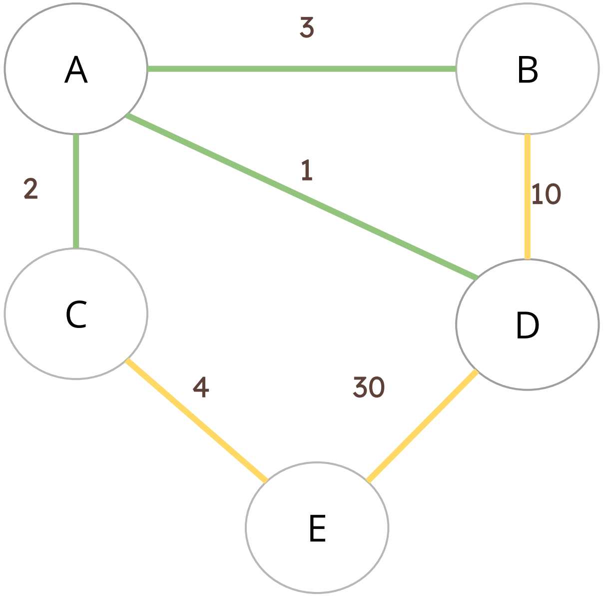
Prim's: Practice



DIAGRAMMING SETUP

PRIORITY QUEUE	VISITED SET	EDGE SET
(A, B, 3)	A	(A, D, 1)
~~(A, C, 2)~~	D	(A, C, 2)
~~(A, D, 1)~~	C	
(D, B, 10)		
(D, E, 30)		
(C, E, 4)		

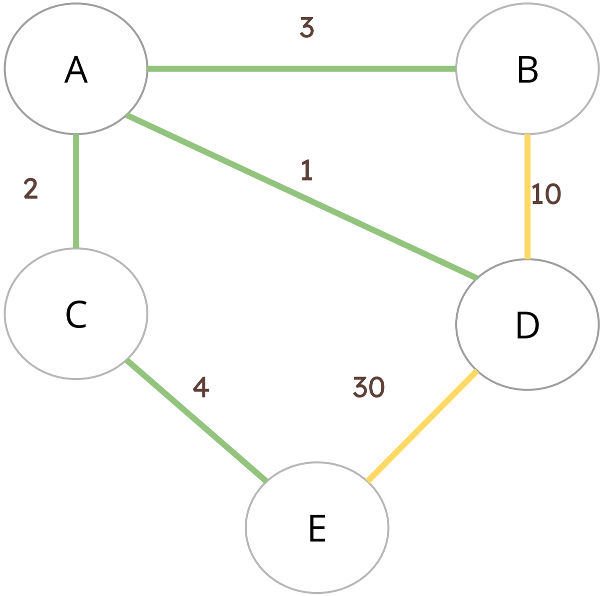
Prim's: Practice



DIAGRAMMING SETUP

PRIORITY QUEUE	VISITED SET	EDGE SET
~~(A, B, 3)~~	A	(A, D, 1)
~~(A, C, 2)~~	D	(A, C, 2)
~~(A, D, 1)~~	C	(A, B, 3)
(D, B, 10)	B	
(D, E, 30)		
(C, E, 4)		

Prim's: Practice



DIAGRAMMING SETUP

PRIORITY QUEUE	VISITED SET	EDGE SET
~~(A, B, 3)~~	A	(A, D, 1)
~~(A, C, 2)~~	D	(A, C, 2)
~~(A, D, 1)~~	C	(A, B, 3)
(D, B, 10)	B	(C, E, 4)
(D, E, 30)	E	
~~(C, E, 4)~~		

Prim's: Efficiencies

STRUCTURES

- **PriorityQueue<Edge<T>>** : *a priority queue of edges in the graph*
- **Set<Vertex<T>>** : *a **visited set** containing vertices already added to the minimum spanning tree*
- **Set<Edge<T>>** : *an **edge set** representing our minimum spanning tree*

ALGORITHM

```
initialize s as start vertex
initialize pq, visitedSet, mst
for all e(s, v) in G:
      pq.enqueue(e(s,v))
while pq is not empty and visitedSet and mst is not full:
      e(u, w) = pq.dequeue()
      if w is not in visitedSet:
            visitedSet.add(w)
            mst.add(e(u, w))
            for all e(w, x) and x not in visitedSet:
                  pq.enqueue(e(w, x))
```

- ❑ **Time:** $O(|E|\log|E|)$ - removing from the priority queue |E| times
- ❑ **Space:** $O(|E|)$ - priority queue could contain all edges, we are guaranteed to not "reuse" edges

Kruskal's

- **PURPOSE**: Finding the minimum spanning tree of an undirected graph.

INTUITION: Dequeuing from a Priority Queue of edges.

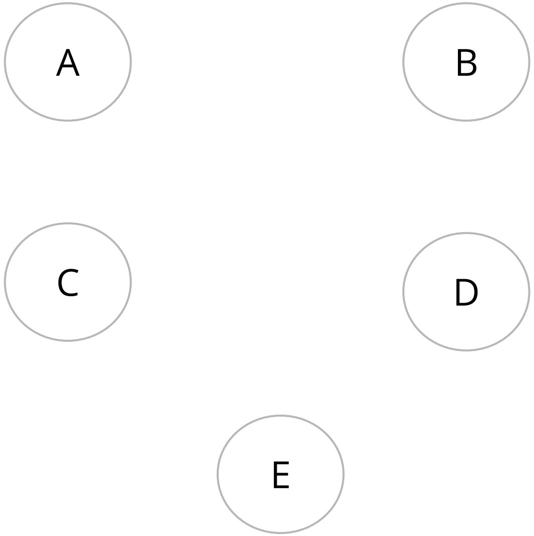
STRUCTURES WE NEED

- **PriorityQueue<Edge<T>> :** *a priority queue of **<u>all</u>** edges in the graph*
- **DisjointSet<Vertex<T>>**: *a **disjoint set** containing vertices ~~already added to the minimum spanning tree~~*
- **Set<Edge<T>>** : *an **edge set** representing our minimum spanning tree*

← **This is our final solution.**

Disjoint Set ADT

- In Kruskal's, using a visited set of vertices does not work. The disjoint set tells us if there is already an edge connecting two vertices in the minimum spanning tree. ***<u>All operations are O(1).</u>***
- **Keeps tracks of vertices that are already connected, visualized as "clusters"**



A constructor taking in a set of vertices

→ {{A}, {B}, {C}, {D}, {E}}

Vertex<T> find(Vertex<T> v)

void union(Vertex<T> v1, Vertex<T> v2)

Don't worry about the implementation, just know how to use it.

Kruskal's: Implementation

STRUCTURES

- **PriorityQueue<Edge<T>>** : *a priority queue of **all** edges in the graph*
- **DisjointSet<Vertex<T>>**: *a **disjoint set** containing vertices already added to the minimum spanning tree*
- **Set<Edge<T>>** : *an **edge set** representing our minimum spanning tree*

Final Step Missing?

ALGORITHM

2 termination conditions

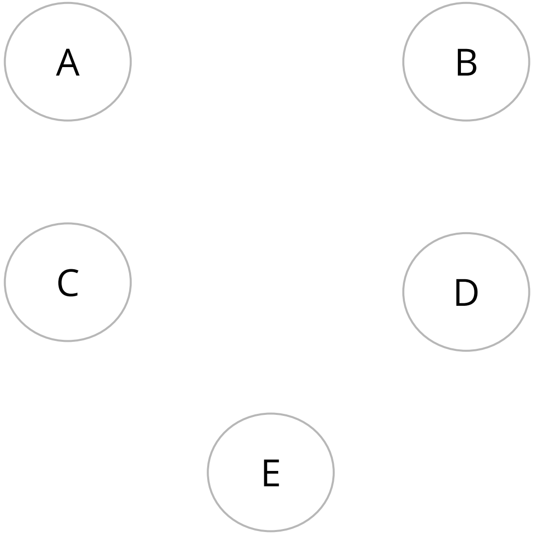
an edge is considered visited once it has been dequeued from pq - GREEDY ALGORITHM

```
initialize disjointSet, mst
initialize pq of all edges in G
while pq is not empty and mst is not full:
    e(u, w) = pq.dequeue()
    if u and w are not in the same cluster:
        mst.add(e(u, w))
        disjointSet.union(u, w)

if mst is not full:
    return null (the graph is disconnected and does not have a valid mst)

return mst
```

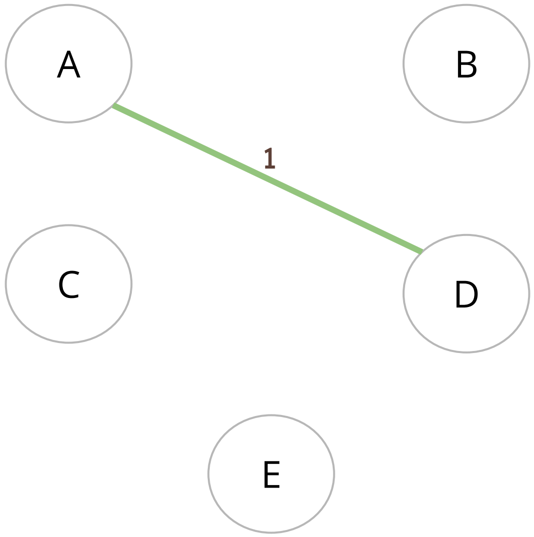
Kruskal's: Practice



DIAGRAMMING SETUP

PRIORITY QUEUE	DISJOINT SET	EDGE SET
(A, D, 1)	{A}	
(A, C, 2)	{B}	
(A, B, 3)	{C}	
(C, E, 4)	{D}	
(B, D, 10)	{E}	
(D, E, 30)		

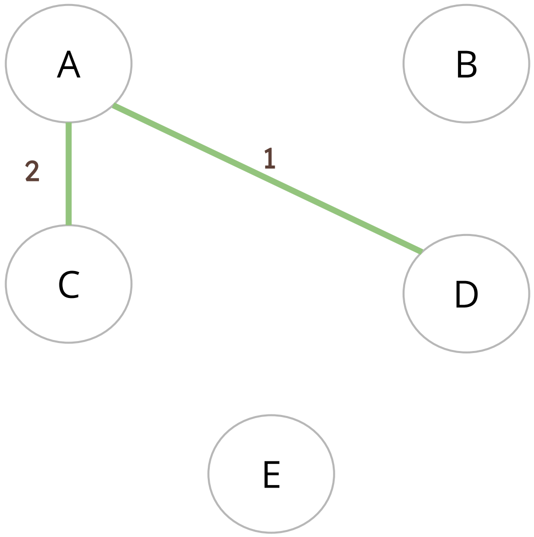
Kruskal's: Practice



DIAGRAMMING SETUP

PRIORITY QUEUE	DISJOINT SET	EDGE SET
~~(A, D, 1)~~	{A, D}	(A, D, 1)
(A, C, 2)		
(A, B, 3)	{B}	
(C, E, 4)		
(B, D, 10)	{C}	
(D, E, 30)		
	{E}	

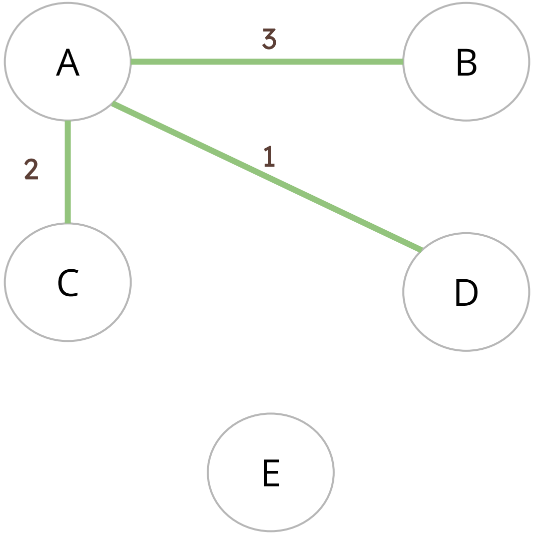
Kruskal's: Practice



DIAGRAMMING SETUP

PRIORITY QUEUE	DISJOINT SET	EDGE SET
~~(A, D, 1)~~	{A, D, C}	(A, D, 1)
~~(A, C, 2)~~		(A, C, 2)
(A, B, 3)	{B}	
(C, E, 4)		
(B, D, 10)	{E}	
(D, E, 30)		

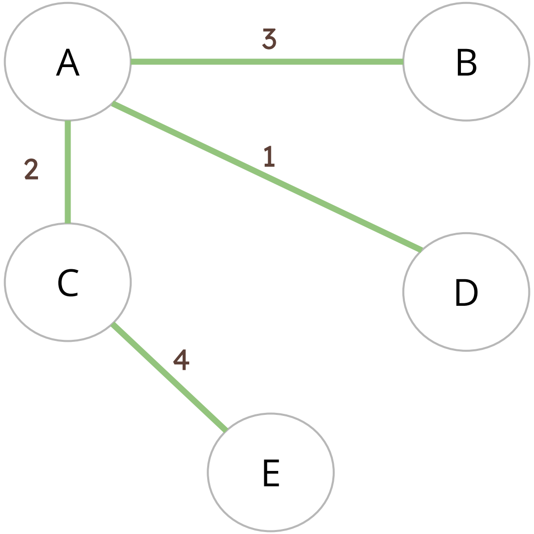
Kruskal's: Practice



DIAGRAMMING SETUP

PRIORITY QUEUE	DISJOINT SET	EDGE SET
~~(A, D, 1)~~	{A, D, C, B}	(A, D, 1)
~~(A, C, 2)~~		(A, C, 2)
~~(A, B, 3)~~	{E}	(A, B, 3)
(C, E, 4)		
(B, D, 10)		
(D, E, 30)		

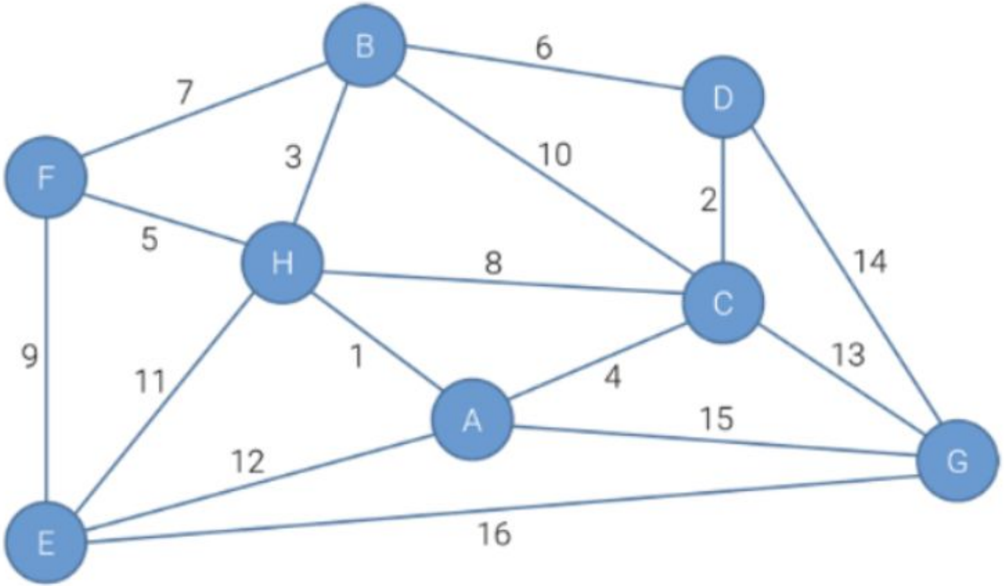
Kruskal's: Practice



DIAGRAMMING SETUP

PRIORITY QUEUE	DISJOINT SET	EDGE SET
~~(A, D, 1)~~	{A, D, C, B, E}	(A, D, 1)
~~(A, C, 2)~~		(A, C, 2)
~~(A, B, 3)~~		(A, B, 3)
~~(C, E, 4)~~		(C, E, 4)
(B, D, 10)		
(D, E, 30)		

Kruskal's: Practice



DIAGRAMMING SETUP

PRIORITY QUEUE

DISJOINT SET

{A}
{B}
{C}
{D}
{E}
{F}
{G}
{H}

EDGE SET

(A, H, 1)
(C, D, 2)
(H, B, 3)
(A, C, 4)
(H, F, 5)
(F, E, 9)
(C, G, 13)

Kruskal's: Efficiencies

STRUCTURES

- **PriorityQueue<Edge<T>> :** *a priority queue of* ***all*** *edges in the graph*
- **DisjointSet<Vertex<T>>**: *a* ***disjoint set*** *containing vertices already added to the minimum spanning tree*
- **Set<Edge<T>>** : *an* ***edge set*** *representing our minimum spanning tree*

ALGORITHM

```
initialize disjointSet, mst
initialize pq of all edges in G
while pq is not empty and mst is not full:
    e(u, w) = pq.dequeue()
    if u and w are not in the same cluster:
        mst.add(e(u, w))
        disjointSet.union(u, w)
```

2 termination conditions

an edge is considered visited once it has been dequeued from pq - GREEDY ALGORITHM

- ❑ **Time: O(|E|log|E|)** (+ O(|E|)) - removing from the priority queue |E| times + buildHeap
- ❑ **Space: O(|E|)** - priority queue

LEETCODE PROBLEMS

[1584. Minimum Cost to Connect All Points]()

[1091. Shortest Path In A Binary Matrix]()

[2642. Design Graph with Shortest Path Calculator]()

Any questions?

Name
Office Hours
Contact

Name
Office Hours
Contact

Let us know if there is anything specific you want out of recitation!