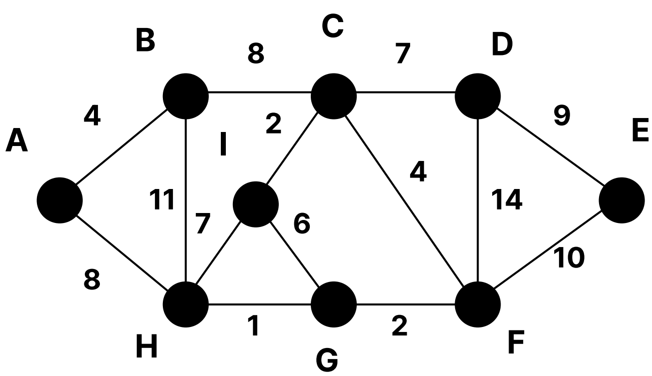
Week 14 (today)

- Minimum single source shortest path (SSSP)
- Kruskal and Prim
  Week 15
- (SSSP) (APSP) Review By Request
- Dijkstra and Bellman-Ford
- Floyd Warshall
  Week 16
- Exam
  Week 17
- Exam Review

Final Exam Topics:

1. Disjoint Sets
2. Binary Search Tree (except delete)
3. Graphs - BFS / DFS / MSP
4. SSSP
5. ASSP
6. One Question from Midterm



Weighted Graphs

Edges have weights (we will only be using integer values).

- Some sort of "cost"
- Unweighted graphs are assumed that all edges have a weight of 1.

Some weights may have negative weight edges.

NOTE: All paths have finite distance

How might we keep track of this?

1. Adjacency List (with some sort of pairing for adjacent and weight)
2. Adjacency Matrix (What do you put for slots that are not connected?)

Minimum Spanning Tree

A minimum weight subgraph in which all nodes are collected.

- A tree that spans the entire graph for the lowest (minimum) cost.

We might want to convert a graph to a tree by discarding edges that we don't need (ie edges that create cycles).

For weighted graphs, we want to try to discard edges with the lowest possible cost.

NOTE: There may not be a unique solution!

Greedy Algorithms

1. Kruskal's Algorithm $O(E \log V)$
2. Prim's Algorithm $O(E \log V)$ with binary heap + adjacency list
3. Prim's Algorithm $O(E + \log V)$ with Fibonacci heap + adjacency list

Basic Idea:

Let the solution set $A = \{\}$

```
while A is not MST
  find an edge (u,v) that is safe for A
  A = A union (u,v)
```

"Safe Edge" is part of the solution.

Finding A Safe Edge

A cut $(S, V - S)$ (AKA a partition)

- Is a partition of the nodes of the graph into two disjoint sets S and $V - S$
- Every node is exactly in one of the two sets. Not in both or neither

edge(u, v) either crosses the cut or it *respects* the cut. One of them will be the edge with the minimum weight (which is the one we will most likely be interested in)

Kruskal's Algorithm

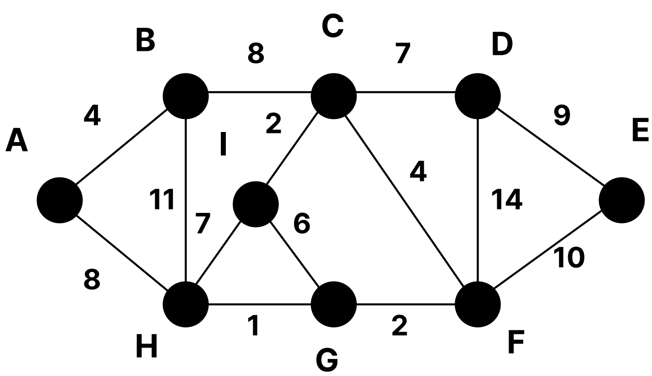
Requires a disjoint set (union find) data structure.

For graph $G(V, E)$:

```
A = {}  // set of edges (our MST)
DisjointSet ds;

for every vertex v in the set V in G(V,E)
        ds.makeSet(v)
        sort E by non decreasing weights
        for each edge(u,v) from E (by weight) in G(V,E)
                if (findSet(u) != findSet(v))
                        ds.union(u,v)
                        A = A union (u,v)
```

Example Graph:



With A being an empty solution set:

- Consider the edges in increasing order by weight
- If the two incident vertices are not in the same set we union them

For our case:

1. Start with edge of weight 1. H and G are not of the same set, so we union them.
2. Looking at the edge between G and F (of weight 2), we notice that G and F are in different sets, so we connect them.
3. The edge between I and C has a weight of two... so we union them.

4. If the next lowest weight contains 2 vertices that are in the same set, we move on.

We end up with the parings:

u	v	weight
h	g	1
g	i	2
i	c	2
a	b	4
c	f	4
c	d	7
b	c	8
d	e	9

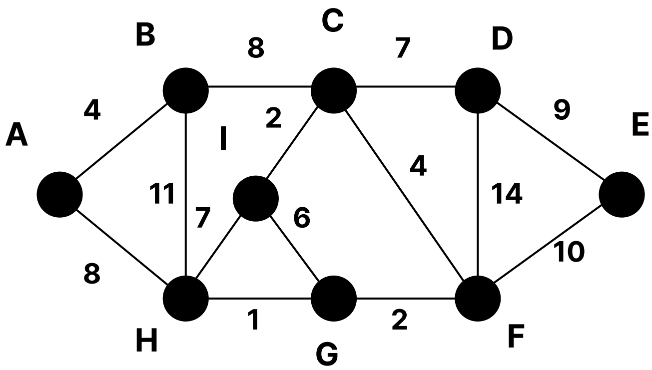
Total cost of our MST: 37

Prim's Algorithm

Uses a minimum (distance) priority queue behind the scenes

```
Prims(Graph G, vertex start)
        for each vertex v in V
                d[v] = INF
                p[v] = null
                PQ.insert(v)
        PQ.decreaseKey(start, 0)
        d[v] = 0
        while (!PQ.empty())
                u = PQ.extractMin()
                for each vertex v in adjacentcylist[u]
                        if (PQ.contains(v) && weight(u,v) < d[v])
                                p[v] = u
                                d[v] = weight(u,v)
                                PQ.decreaseKey(v, weight(u,v))
```

Now let's simulate it:



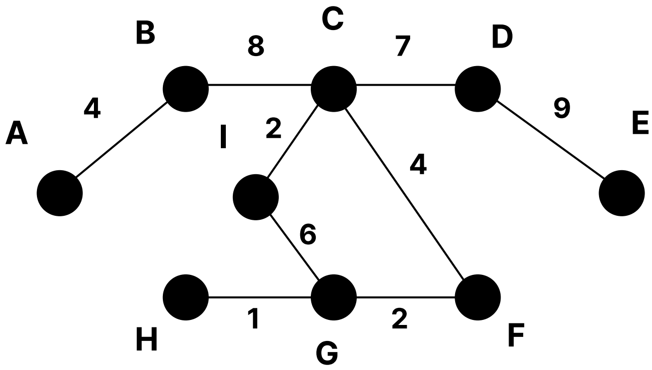
Steps:

1. Set every vertex to INF
2. Throw them into priority queue (no order currently)
3. Select a starting node (for this we do H)
4. Decrease key of H to 0
5. Enter while loop
    1. Adjacent: $\{a, g, i\}$
    2. is A in the PQ? Yes.
    3. Change parent $p[A] = H$
    4. Decrease key of A to 8
6. Eventually we find that the lowest value to G from H is 1. We then remove G from the PQ and finalize the edge.

Priority Queue:

1. $\{a\}$
2. $\{a, b\}$
3. $\{g, a, b\}$
4. $\{g, i, a, b\}$
5. $\{f, i, a, b\}$
6. $\{i, a, b\}$
7. $\{c, i, a, b\}$
8. $\{c, i, a, e, b, d\}$
9. $\{i, a, e, b, d\}$
10. $\{i, a, e, b, d\}$
11. $\{i, d, b, a, e\}$
12. $\{d, b, a, e\}$
13. $\{b, a, e\}$
14. $\{a, e\}$

15. $\{e\}$



u	v	weight
h	g	1
g	i	2
i	c	2
a	b	4
c	f	4
c	d	7
b	c	8
d	e	9

Total cost of our MST: 37

Single Source Shortest Path Algorithms

(Like breadth first search for weighted graphs)

Textbook has 3 algorithms

1. Dijkstra's $O(E \log V)$ with binary heap (could be faster with fibonacci heap $O(V \log V + E)$
2. Bellman-Ford $O(EV)$

Dijkstra's

- no negative weight edges
- uses PQ

Bellman-Ford

- allows negative weight edges
- detects negative weigh cycles

Common Code For Dijkstra and Bellman-Ford

Initialization:

```
Initialization(V,s)
        foreach vertex v in V
                d[v] = INF   // shortest path estimate
                p[v] = null  // shortest path tree
        d[s] = 0
```

Relax: Used to improve the (currently) best known path to a node.

```
Relax(u,v)
        if (d[v] > d[u] + weight(u,v))
                d[v] = d[u] + weight(u,v)
                p[v] = u
```

The main difference in these two algorithms are their strategies for relaxing (or when you relax)

- Finds most optimal edge to relax (one at a time)
- Bellman Ford relaxes everything over and over until everything is relaxed.

Dijkstra's

"Smarter" relaxation

```
Dijkstra(Graph g, Node s)
        Initialization(V,s)
        foreach node v in V
                PQ.insert(v)
        while(!PQ.empty())
                u = PQ.extractMin()
                foreach node v in adjacency_list[u]
                        Relax(u,v)
```

Bellman-Ford

Assumes worst case every time.

```
Bellman_Ford(Graph g, Node s)
        Initialization(V,s)
        // working part
        for (i = 0; i < |v| - 1; i++)  // for every node... kind of
                for each edge(u,v) in E
                        Relax(u,v)
        // error checking
        // try to relax some more
        for each edge(u,v) in E
                if (d[v] > d[u] + weight(u,v))
                        return error...
```