Question 1 Suppose all edge weights in a graph are integers in the range from 1 to $|V|$, how fast can you make Kruskal's algorithm run?

Kruskal's algorithm consists of two major considerations

① Sorting $\Rightarrow O(m \log m)$

(constructing a minimum priority queue based on edge weights, and removing lowest valued edge not yet considered)

② Merges/Finds $\Rightarrow O(n + m \log^* n)$

(order of growth for m union-find operations on a set of n objects using weighted QU + path compression)

$\Rightarrow$ Overall runtime is $O(m \log m)$ due to sorting

How can we improve?

As we know the edge weights are in a certain range, we can using sorting methods such as hash sort, bucket sort, or counting sort to sort edge weights in linear time! (we discussed these methods in CSC225 so I assume I don't have to explain more about this fact!)

This causes our bound to be based on the weighted QU data structure, which means our runtime overall is:

$$O(m \log^* n) \text{ now}$$

↑ theoretical runtime, real world is much lower!

Question 2 Given a MST for a weighted graph, G, suppose we delete an edge from G. Design an algorithm to find the modified MST. What is the runtime?

NewMST find (mst, graph, deleted edge)

if deleted edge is NOT in MST return original mst

Otherwise,

delete edge from MST and assign:

x ← tree from one side of deleted edge (from MST)

y ← tree from other side of deleted edge (from MST)

for each edge that connects x and y find minimum edge

add minimum edge to mst and return new mst

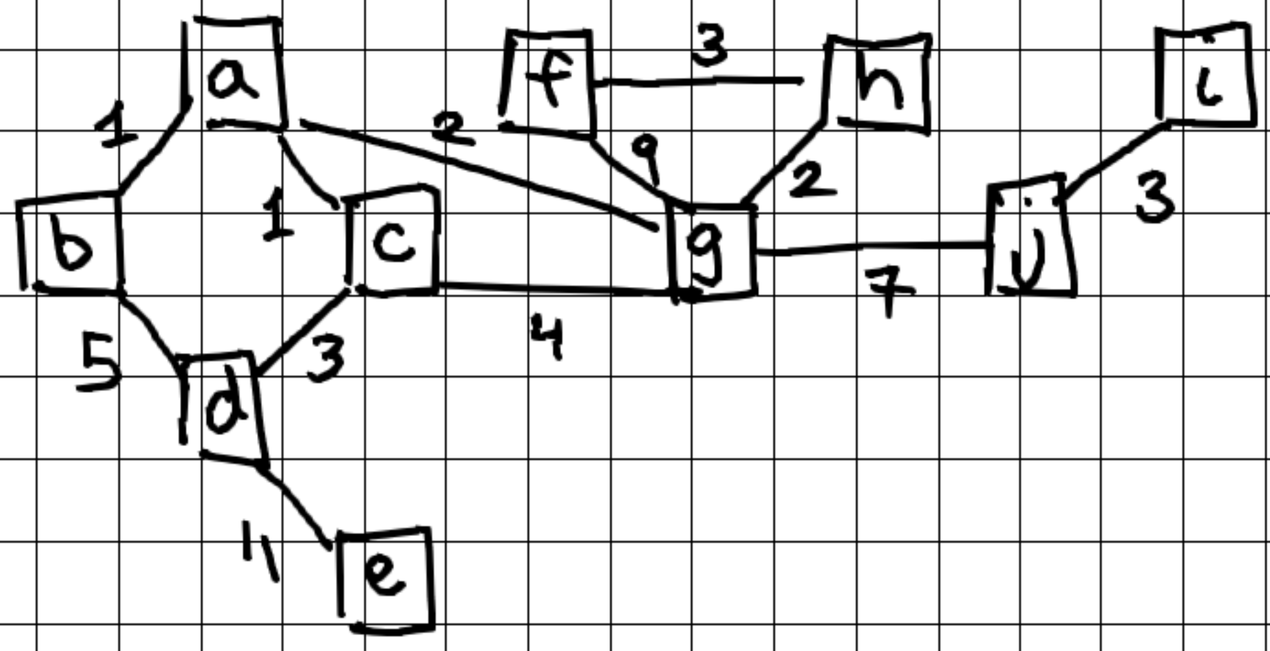
This algorithm is based on the cut property!

The runtime is defined by our for loop that searchs over all edges. In the case we are given basic data structures as input this would cause our runtime to be O(m). //

This could be improved if we are given more advanced data structures for the MST and graph.

(note: Why this is O(m) is because we iterate over all possible edges in the worst case, and do constant work to record the minimum of the edges that satisfy our requirements)

Question 3 Show Dijkstra's algorithm with the source node as "g" on the following graph



Note: Visit(u) occurs when popping out of min priority queue where D(u) is the key of the queue

I will state D(u) next to Visit(u) and that value is used to add to the weight of each edge from u to update D(neighbour) values

Initalization:

all D(u) values are set to $+\infty$, D(g) = 0

Visit(g): (D(g)=0)

D(f)= 9, D(h)=2, D(j)=7, D(a)=2, D(c) = 4

Closest vertex → a or h → choose a

Visit(a): (D(a)=2)

example of how D(u) updated

$\boxed{D(c) = D(a)+w(a,c) = 2+1=3}$, D(b) = 3

Closest vertex → h, chose h

Visit(h): (D(h)=2)

D(f) = 5,

closest vertex → b or c → choose b

Visit(b): (D(b)=3)

D(d) = 8,

closest vertex → c → choose c

Visit(c): (D(c)=3)

D(d)=6,

Visit closest vertex → f → choose f

Visit(f): (D(f)=5)

nothing to update

visit closest vertex → d → choose d

Visit(d): (D(d)=6)

D(e)=17,

visit closest vertex → j → choose j

Visit(j): (D(j)=7)

D(i)=10,

visit closest vertex → i → choose i

Visit(i): (D(i)=10)

nothing to update

choose closest vertex → e → choose e

Visit(e): (D(e)=17)

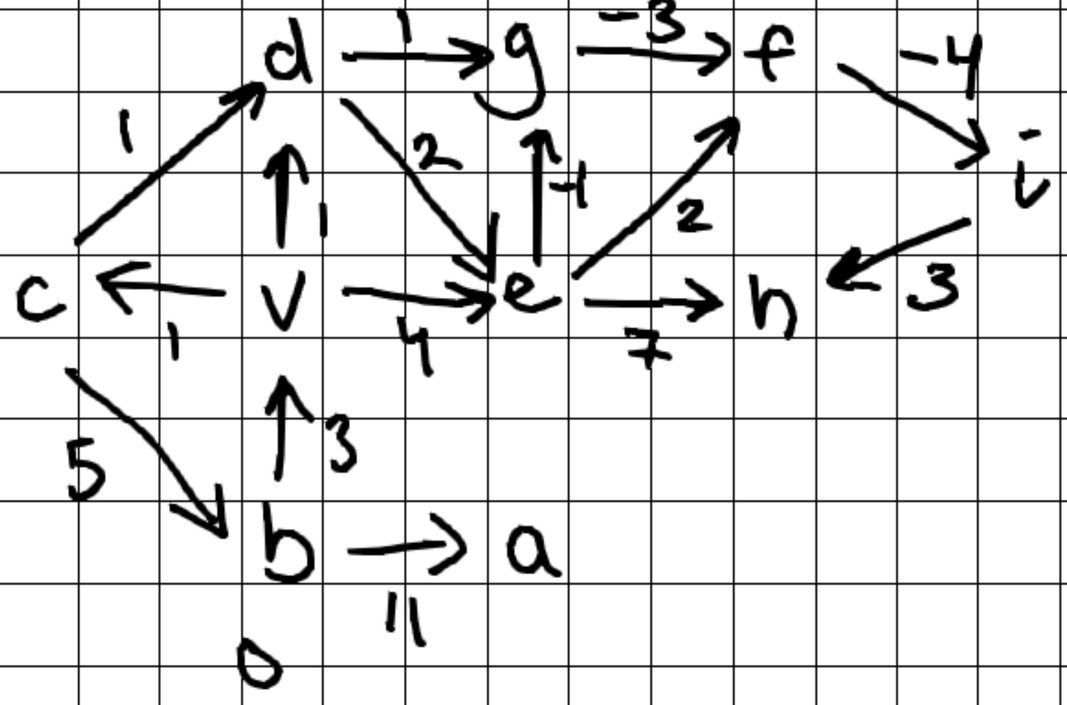
nothing to update

chose closest vertex → queue empty → done!

Final Values:

D(a)=2, D(b)=3, D(c)=3, D(d)=6, D(e)=17, D(f)=5
D(g)=0, D(h)=2, D(j)=7, D(i)=10

Question 4 Show Bellman-Ford's algorithm on the following graph with "b" as the source node. Considering lexicographical order of outbound edges.



Initialization: $D(b) = 0$, $D(u)$ for all other nodes $= +\infty$

Edges Ordered		Iteration 1	Iteration 2	Iteration 3	Iteration 4
1)	b → a	D(a) = 11	x	x	x
2)	b → v	D(v) = 3	x	x	x
3)	c → b	x	x	x	x
4)	c → d	x	x	x	x
5)	d → e	x	D(e) = 6	x	x
6)	d → g	x	D(g) = 5	x	x
7)	e → f	x	D(f) = 8	x	x
8)	e → g	x	x	x	x
9)	e → h	x	D(h) = 13	x	x
10)	f → i	x	D(i) = 4	D(i) = -2	x
11)	g → f	x	D(f) = 2	x	x
12)	i → h	x	D(h) = 7	D(h) = 1	x
13)	v → c	D(c) = 4	x	x	x
14)	v → d	D(d) = 4	x	x	x
15)	v → e	D(e) = 7	x	x	x

Iteration 4 has no edge relaxation ⇒ exit algorithm ✓

Final values:

D(a) = 11	D(f) = 2
D(b) = 0	D(g) = 5
D(c) = 4	D(h) = 1
D(d) = 4	D(i) = -2
d(e) = 6	D(v) = 3

Question 5 — Design an algorith for SSSP problem on DAGs in O(m+n) time

Idea: Topological Sort ensures a sort such that at any nodes children exist at a later index. So we sort DAG topologically from source and visit each node in that order. when we visit the node, we update a distence array of its adjacent edge if edge is relaxable.

DAGShortestPath (graph, source)

```
T ← topologicalSort(graph) # returns queue
D(source) ← 0  # distance array
D(u) for all other nodes ← +∞

while T is not empty
    cur ← T.pop()
    for each neighbour adjacent to cur
        if relaxable  # D(cur) + w(cur,neighbour) < D(neighbour)
            relax
return
```

Topological Sort is O(m+n) as it is just DFS the rest of our code iterates through each node, and then the neighbors of that node, which is O(m+n)

So overall this algorithm is O(m+n) //

A bit of explanation:

This algorithm was born from Bellman Fords. The reason why it runs quicker is because a DAG has a possible ordering that ensures you can always visit any nodes children later in that ordering. (Topological Ordering) Because of this, you only need to do one pass on this ordering instead of n-1 passes as it ensures that at any node, it's parent has already it's distance value updated from it's parent. (repeat infinitely until base case of source reached) This ensures the need for only one pass!

Proof wasn't asked for, so just gave informal explanation!