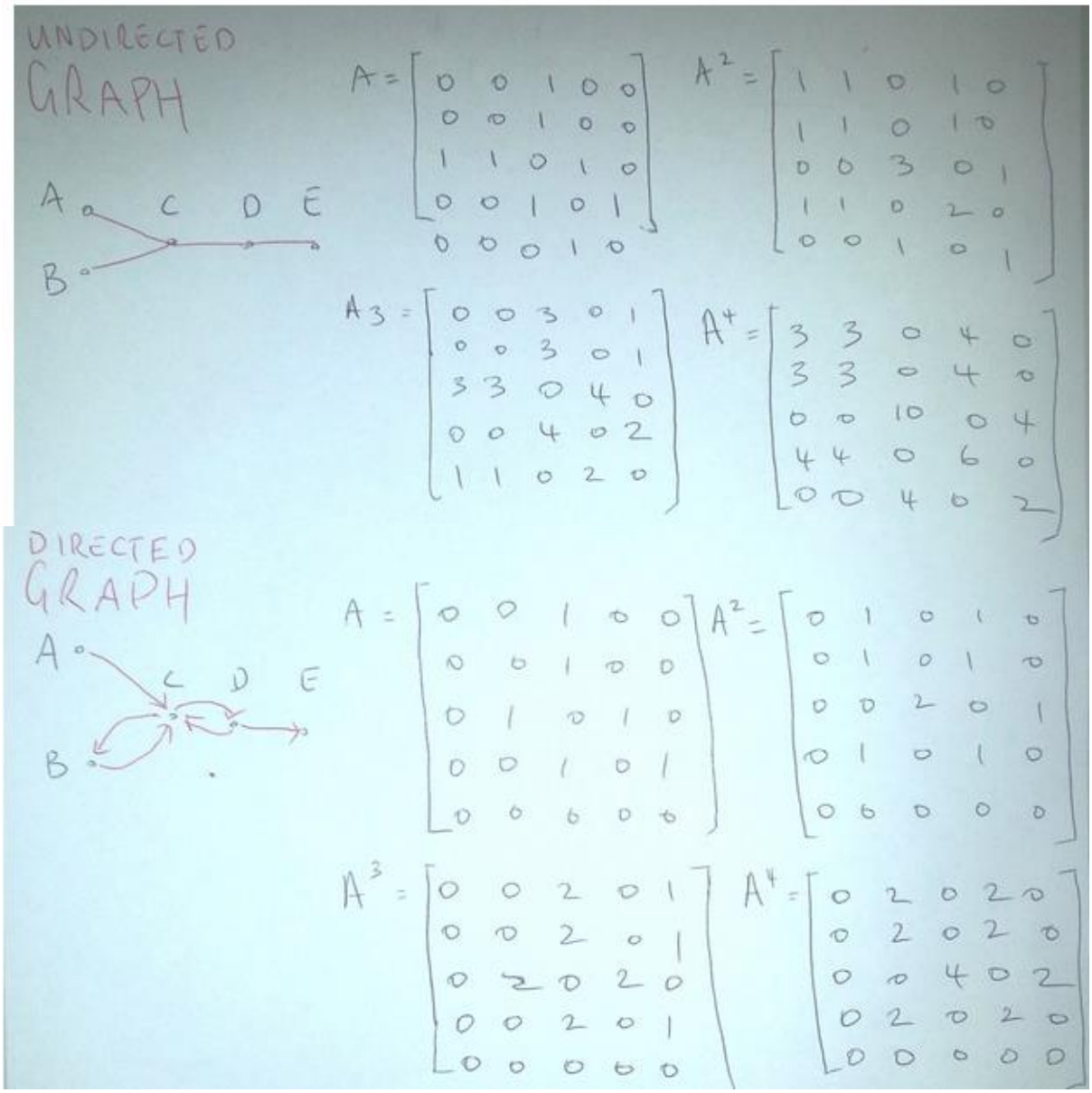
# FIT2004 S1_2015 Tute Week11

**Question one -**




**Question two**

a. revise bellman ford algorithm
b. discuss correctness for handling negative weight cycles
c. run bellman ford for the 5 vertex graph provided

a)The Bellman-ford algorithm functions by considering every path of length k from a single source node; in each successive iteration we consider k+1 and then k+2 edge paths until such time as we have considered every path up to a path length equal to the number of vertices in the graph. For clarity that's the number of edges included as opposed the to cost of that path.

For any particular iteration, we consider if we can reach some vertex v from u by accessing some new vertex k in between; should that new path be cheaper, we update the existing path to use vertex k

b)
as for why bellman ford is correct when it determines there to be a negative weight cycle in the graph. Once the algorithm proper has terminated, we will have considered every path of length up to and including each and every vertex. The cycle detection part suggests that if we can (after including all paths up to v edges) find a cheaper way of getting to a particular vertex we must have a cycle. The reason this is correct is that if we have included every vertex in our path thus far and we can find a cheaper way of reaching a vertex then we are revisiting a previously found vertex later with less cost after travelling through the entire graph once; this suggests a negative weight cycle as otherwise we would expect the total to be no less than the current cost. having more than v vertices in a path means we can traverse through a negative weight cycle (as that will always be preferenced over other paths as it will always be less) and considering paths beyond the number of vertices includes negative weight cycles where the whole graph is one huge cycle (and of course having a cycle larger than one including every vertex is impossible by definition)

| c) | i0 | i1 | i2 | i3 | i4 |
|---|---|---|---|---|---|
| s | 0 | 0 | 0 | 0 | 0 |
| u | inf | 5 | 4 | 4 | 4 |
| v | inf | 8 | 8 | 8 | 6 |
| w | inf | inf | 8 | 7 | 7 |
| x | inf | inf | inf | 7 | 6 |

taking one more iteration we see we can reach u in 2, this is cheaper than the current estimate for u and hence we have a negative weight cycle making the problem of shortest path undefined

## Question Three

*Work out the time complexity of Kruskal's algorithm for Minimum Spanning Tree problem using the Union-Find data structure introduced in Week 10 lecture.*

Given that the set of edges is sorted or placed in a priority queue, we can access each minimum weight edge that isn't already part of the spanning forest in constant time O(1). This initial sorting procedure (or priority queue construction, same-same) can be performed in, at best, O(E log E) time.

Once this initial sort is complete, the actual algorithm can be completed by examining every edge and, for each, either adding it to the forest or discarding it if it creates a cycle. Discarding the edge is a constant time operation, but merging two trees in the forest takes O(UNION(x, y)) time.

Given that you have to iterate through the smaller tree on union to update its subset map, the naive suggestion is that time time taken is O(E * V). We can be more specific. Given that the subset being merged is smaller than the current set, this means that the union set size is at least double the smaller set size on every iteration.

|X| >= |Y|
therefore

2 * |Y| <= |UNION(X, Y)|

Given that the size of our set can be at most size V, this means that our smaller set can only be doubled (and therefore “merged” on union) at most log2(V) times.

This means the time complexity of the general algorithm loop is E * log(V).

Our total time complexity for the algorithm is max(O(E*logV), O(E*logE)). Given that E is at most V^2, this means logE = logV^2 = 2 logV. This means O(E*logE) = O(E*logV^2) = O(E*2*log(V)) = O(E*logV).

The time complexity for Kruskal’s algorithm is O(E*logV).

**Question Four**

*Reason why kruskal's algorithm is correct.*

Given that we start with a set of spanning trees of |V| = 1, these spanning trees must be minimum spanning trees. With this base case in mind, the reasoning behind why Kruskal’s is correct is the same as the generic proof given in Lecture 10.

A quick summary of the proof by contradiction:

Let a minimum spanning tree T contain lightest edge <x,y> connecting two other minimum spanning tree cut sets.

Let minimum spanning tree T’ contain the same two cut sets, with a different (not lightest) edge <a,b> connecting the two cut sets.

Assume a spanning tree can be found by connecting two cut sets with an edge that is *not* the lightest edge. If this is true, W(T’) <= W(T).

With the above definitions in mind:

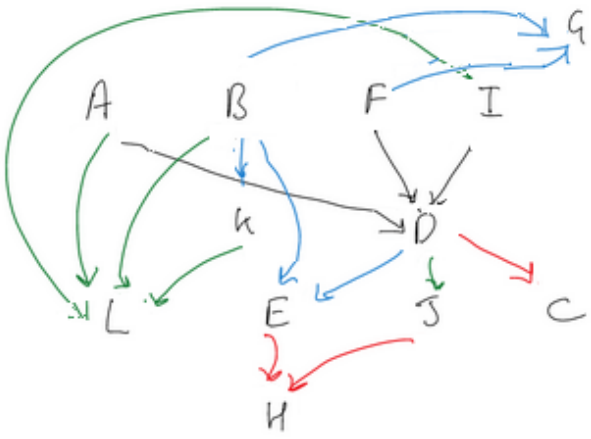
W(<x,y>) < W(<a,b)
so
W(<a,b>) - W(<x,y) > 0.

From here, we can describe the weight of T’ as:
W(T’) = W(T) - W(<x,y>) + W(<a,b>)
but this means that
W(T’) >= W(T)
which contradicts our premise. Therefore adding the lightest edge at every step must result in a minimum spanning tree.

In Kruskal’s, we’re adding edges across a whole graph, one at a time, starting with the lightest.

**Question Five**

A FIT 1040 —
B MAT 1830 —
C FIT 3042 D
D FIT 1008 A, I, F
E FIT 2004 D, B
F FIT 1029 —
G FIT 2014 F, B
H FIT 3036 E, J
I FIT 1002 —
J FIT 3140 D
K MAT 2003 B
L FIT 3139 I, A, B, K




topological
↑ precedence constraints
"courses in a Uni curriculum"

1st Year ; 2nd Year ; 3rd Year *

[A B F I ; D K G E ; C L J H]

choose based on unit number
and "year" 1008 first year
2004 second year etc.