DMFS - Problem Set 4

pwn274

March 2022

1

1.1 Find strongly connected

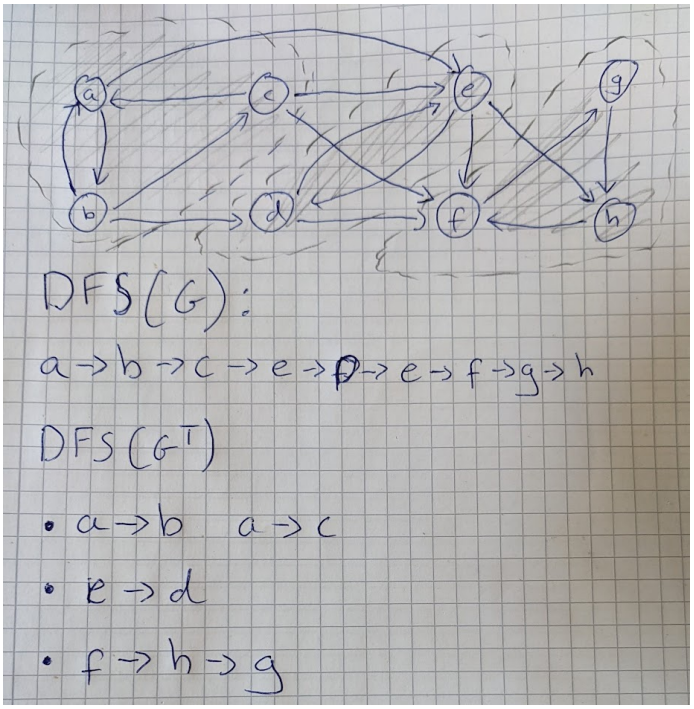

Figure 1: Strongly Connected

The first pass simply requires running a Depth First Search, noting down the order. Here I get:

$$a \to b \to c \to e \to d \to f \to g \to h \tag{1}$$

Now, reversing the direction of all edges, I run a second Depth First Search pass by hand, this time going in the opposite direction of the order I closed them off in the first pass. This gives the following three trees (threes?):

$$\{a, b, c\} \text{ , } \{e, d\} \text{ and } \{f, g, h\} \tag{2}$$

1.2

I did not have time to make this

2

2.1 Generate a minimum spanning tree by running Kruskal's algorithm by hand on the graph

Starting from an empty set A, the edges are added in the following order. I am lazy so the "$-||-$" short hand simply means "the same as last".

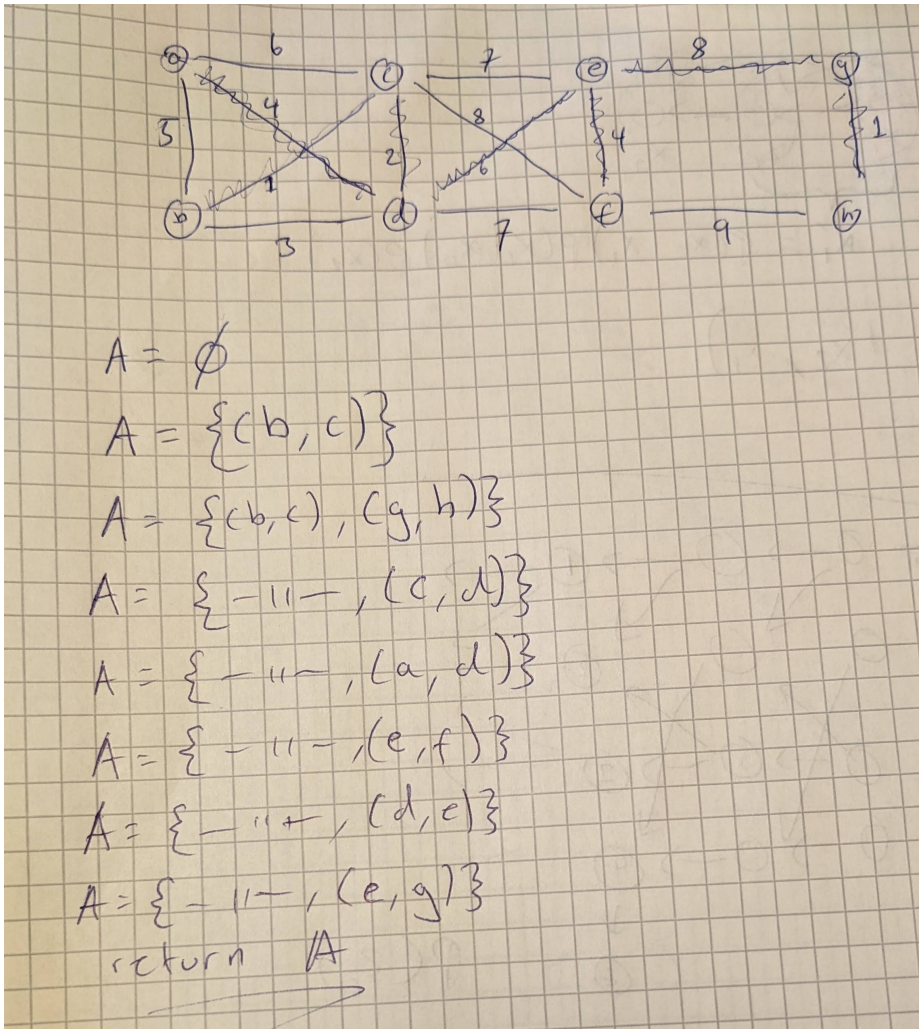

Figure 2: Minimum Spanning Trees by Kruskal

2.2

I am not sure what the argument here would be

2.3 Vertex is strictly higher

This is a lot easier to prove, as I can simply come with a simple counterexample: If a node only has two edges (one being strictly higher than the other) but connects two parts of the graph, we can see that both edges needs to be utilized even though one is unambiguously 'the highest' incident from the node. See figure 3 where both the edge of weight k and $> k$ are utilized:

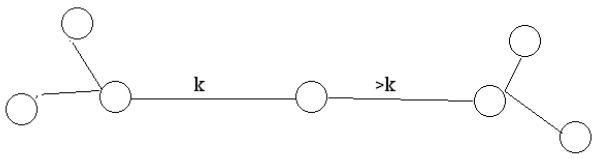

Figure 3: Simple counterexample

2.4

Since the actual weights are not taken into account once the algorithm is running (it simply does comparisons) adding a constant to all elements does not change the sorting order, which means that T would stay a minimum spanning tree. Another way to see this is that the number of edges in a minimum spanning tree is constant, making the total sum of any path higher by the same amount ($(N-1)C$).

3

3.1 Run Dijkstra by hand

I start out by initializing all all distances to be ∞. I then run the algorithm updating the distances as follows:

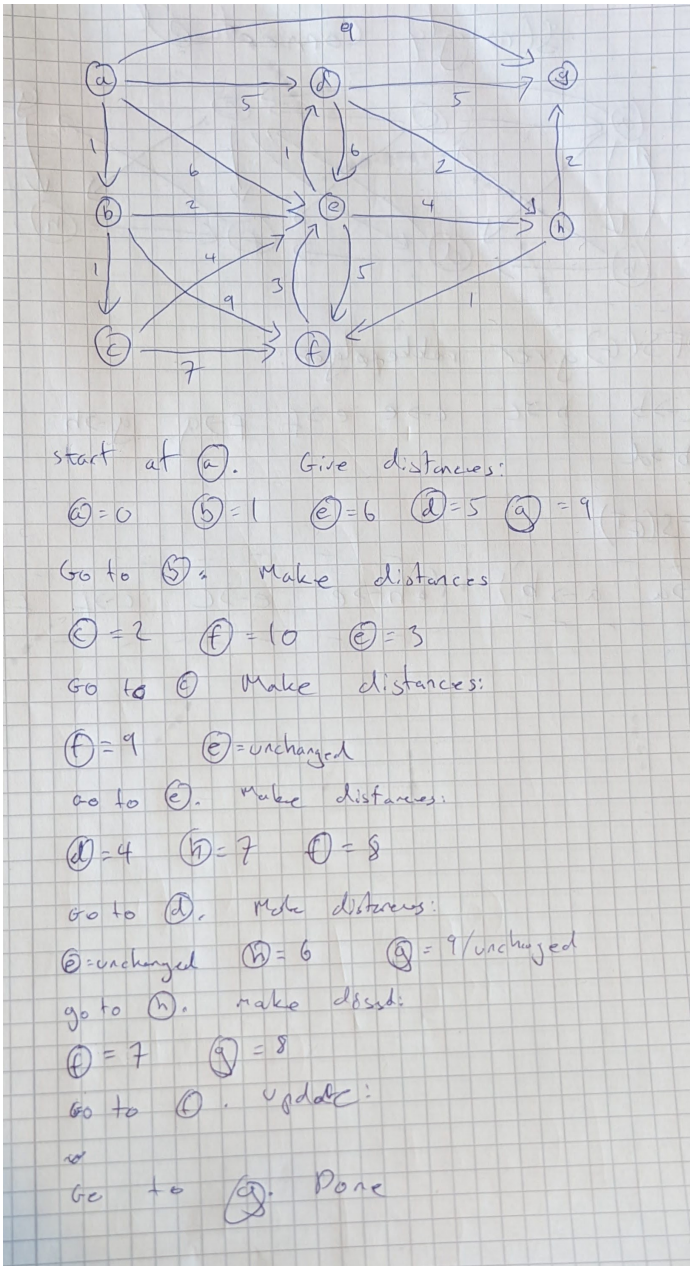

Figure 4: Dijsktra

3.2 Additive constant

An additive constant would absolutely change something. Having read ahead, this kind of has the effect of the last question - multiple hops gives multiple instances of the constant, making the algorithm prefer using fewer edges. As this happens for any two paths with the same weight before but a different number of edges, coming up with a counter example is trivial.

3.3 Multiplicative constant

On the other hand, multiplying by a constant does not change the optimal path. There are multiple ways to see why this is. My initial idea was:
When comparing two paths in an inequality, they each consist of a sum of edges. Basic math tells us that we can pull any positive multiplicative constants outside parentheses, dividing by it on both sides of the inequality without flipping the sign. Therefore, multiplying every edge will never change which path is better between any two nodes.

3.4 Few hops

Here, simply adding 1 to every edge will grant us the algorithm we want. This can this can either be done on a first pass changing our runtime from $\Theta((V+E)\log V)$ to $\Theta((V+E)\log V + E)$. It can also be done during the iterations by redefining the "relaxing" algorithm to check against the weights + 1. This would not change the asymptotic running time.