ECE385: Homework 04

Farbod Mohammadzadeh
(1008360462)
Connor Ludwig
(1008333028)

24 Novermber 2023

11 Pages

Problem 1

(a)

G and H share the same set of minimum spanning trees. First it must be proved that any minimum spanning tree for G is also a minimum spanning tree for H and then it must be proved that any minimum spanning tree for H is also a minimum spanning tree for G.

Let $w_g(graph)$ be the summation of all weights of a tree in the graph G, and $w_h(graph)$ be the summation of all weights of a tree in the graph H. Let T be a minimum spanning tree for G

Proof:

Recall, by properties of a MST, an MST for G or H will have $V-1$ edges. Because each edge in H has a weight that is 1 greater than the weight in G, $w_h(T) = w_g(T) + (V-1)$ **ATAC:** Assume there is a minimum spanning tree A for the graph H that is not a minimum spanning tree for G. By definition, $w_g(A) > w_g(T)$ and $w_h(A) \leq w_h(T)$.

$$w_g(A) > w_g(T) \tag{1}$$
$$w_h(A) - (V-1) > w_h(T) - (V-1) \tag{2}$$

The $-(V-1)$ terms cancel:

$$w_h(A) > w_h(T) \tag{3}$$

But by definition:

$$w_h(A) \leq w_h(T) \tag{4}$$

Contradiction! If A is not a minimum spanning tree of G, it also cannot be a minimum spanning tree for H.

ATAC: Assume there is a minimum spanning tree B for the graph G that is not a minimum spanning tree for H. By definition, $w_h(B) > w_h(T)$ and $w_g(B) \leq w_g(T)$.

$$w_h(B) > w_h(T) \tag{5}$$
$$w_g(B) + (V-1) > w_g(T) + (V-1) \tag{6}$$

The $+(V-1)$ terms cancel:

$$w_g(B) > w_g(T) \tag{7}$$

But by definition:

$$w_g(B) \leq w_g(T) \tag{8}$$

Contradiction! If B is not a minimum spanning tree of H, it also cannot be a minimum spanning tree for G.

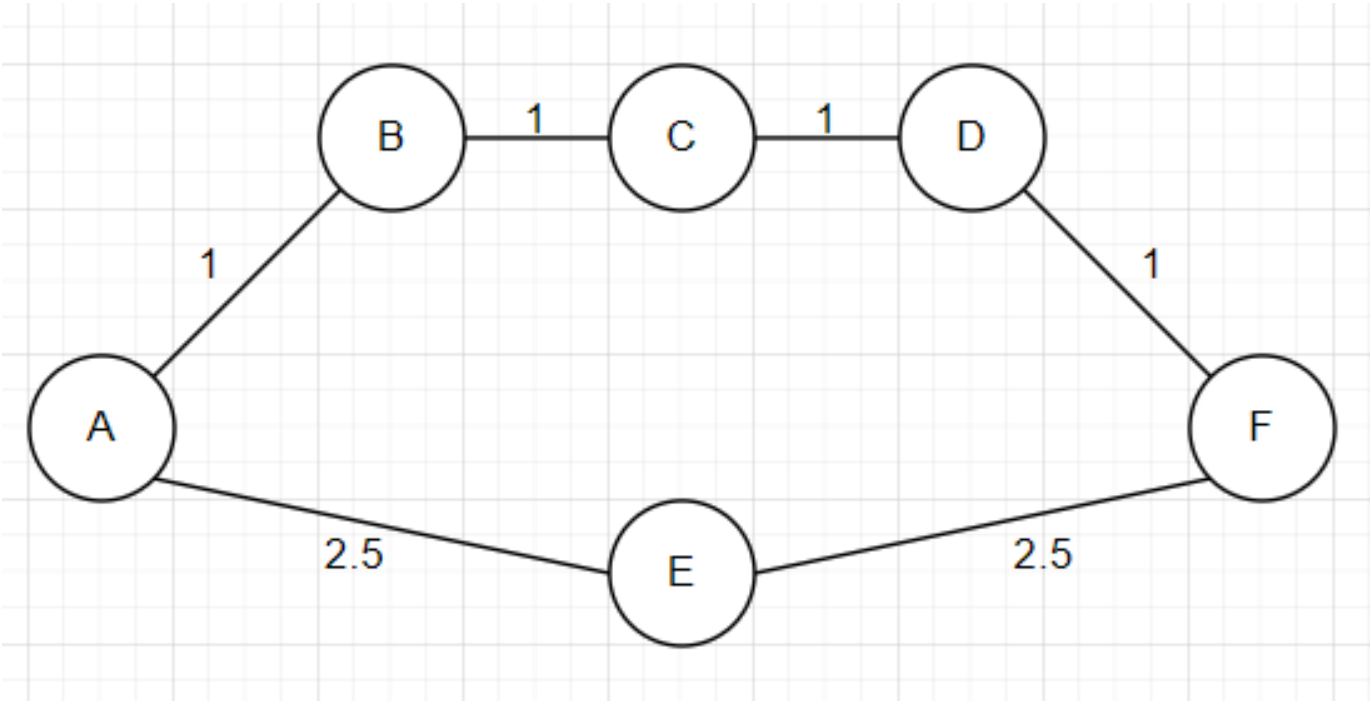
This means that there cannot be any graph that is a minimum spanning tree for H but not G or a minimum spanning tree for G but not H.

Therefore, G and H share the same set of minimum spanning trees.

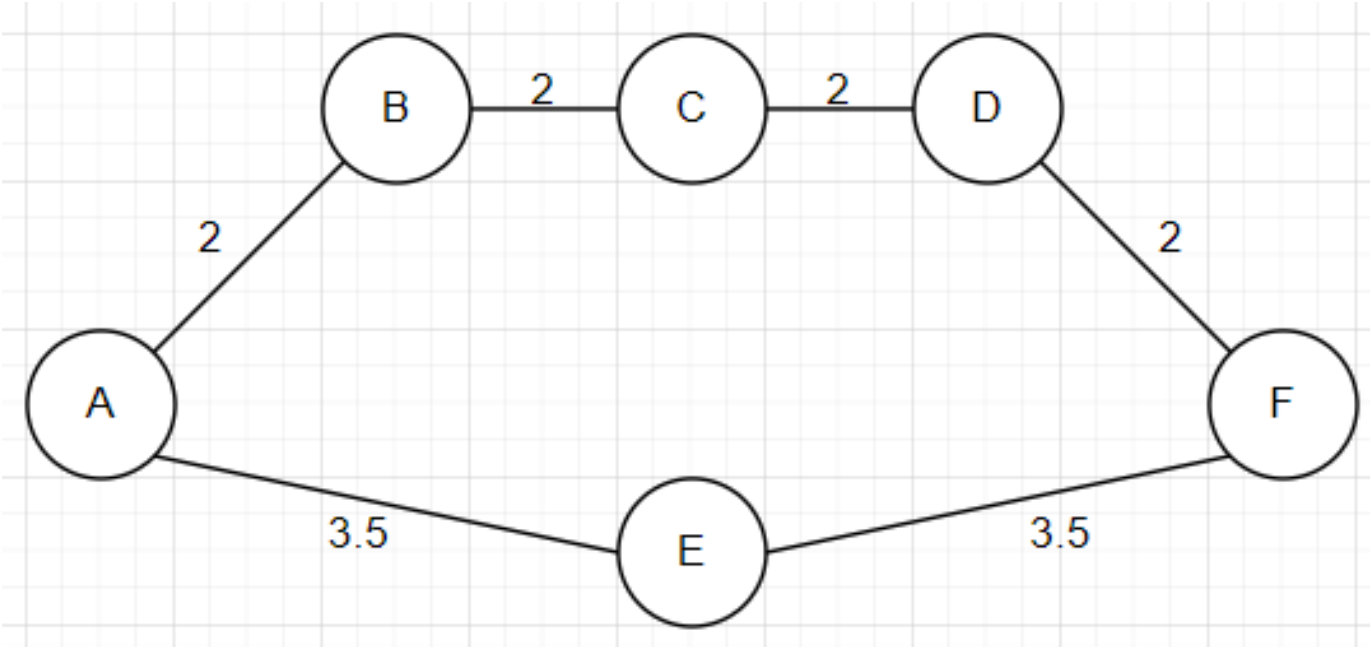
(b)

G and H do not always share the same shortest path between all pairs of vertices.

Consider the following graph G



The shortest path from A $\rightarrow$ F is: A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D $\rightarrow$ F, having a total weight of 4. This same graph in H becomes:



Now the path A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D $\rightarrow$ F has weight 8, while the path A $\rightarrow$ E $\rightarrow$ F has weight 7. This means that the shortest path A $\rightarrow$ F is A $\rightarrow$ E $\rightarrow$ F. This counter example proves that G and H do not always share the same shortest path between all pairs of vertices.

Problem 2

Based on the proof in chapter 22.3 of the textbook we know that Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph in

$$O((V + E)\lg V)$$

With this algorithm we can calculate a $minWeight$ dictionary that will give us the minimum weight required to get from any North American node to the connection node and the minimum weight required to get from any European node to the connection node.

We can then combine those values with the weight of the v_{E1} to get:

$$query(v_{NA}, v_E) = minWeight(v_{NA}) + weight(v_{E1}) + minWeight(v_E)$$

This operation will compute in $O(1)$ since all the values are pre-computed.

Problem 3

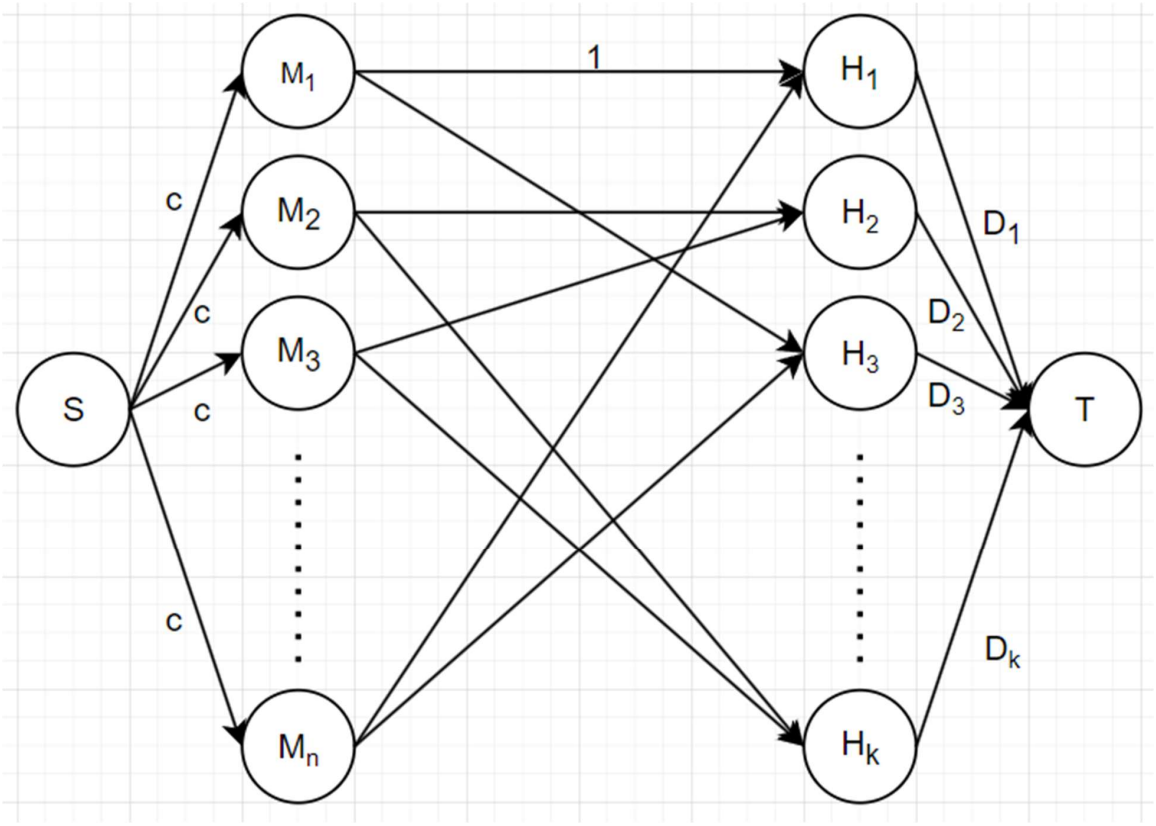
Algorithm 1 Currency Exchange

```
1: function ADD-RATE(FromCurr,ToCurr,Rate,AdjList)
2:     AdjList[FromCurr].append( (ToCurr, Rate) )
3:     AdjList[ToCurr].append( (FromCurr, 1/Rate) )
4: end function
5: function RATE-SEARCH(InCurr)
6:     if InCurr == OutCurr then
7:         return 1
8:     end if
9:     for (PossibleCurr, Rate) in G[InCurr]  do
10:        if PossibleCurr not in visited then
11:            result = RATE-SEARCH(PossibleCurr)
12:            if result not null then
13:                return Rate * result                    ▷ Conversion found
14:            end if
15:        end if
16:    end for
17:    return null                                         ▷ No conversion found
18: end function
19: function FIND-CONVERSION(FromCurr,ToCurr,Rates)
20:    AdjList = {∅}
21:    for (curr1, curr2, rate) in Rates do
22:        ADD-RATE(curr1, curr2, rate,AdjList)
23:    end for
24:    visited = [∅]
25:    conversion = RATE-SEARCH(FromCurr)
26:    if conversion not null then
27:        return conversion
28:    else
29:        return "conversion not possible"
30:    end if
31: end function
```

Problem 4

Problem 4

Consider the following graph (all numbers on edges represent the capacity of the edge, not the flow):



M_i represents doctor i, and H_j represents vacation/holiday j. The flow of this graph is the vacation/holiday days that can be worked. Each doctor can only work at most c vacation days, so each edge connecting S to a doctor node M_i has capacity c. Each doctor can only work 1 day per vacation period, so each edge from a doctor node M_i to a vacation node H_j has capacity 1. Note that the amount of connections between a doctor node M_i and the set of vacation nodes H, is related to the doctor's availability. A doctor node will only flow into vacation nodes if the doctor has availability on that vacation. Each edge connecting vacation node H_j to T has capacity D_j where D_j is the number of days in that vacation period. The flow from H_j to T represents the number of days in that vacation period that can be covered by a doctor. If the flow into T is equal to N, it means that all vacation days were able to be covered.

Pseudocode Algorithm (Ford-Fulkerson):

#Setting up graph, this code will create the graph pictured on the previous page

create graph G=(V,E)

add source node S to G

add sink node T to G

add nodes M_i to G for all i ∈ n

add nodes H_j to G for all j ∈ k

add edge (S, M_i) with capacity c to G for all i ∈ n

add edge (H_j, T) with capacity D_j to G for all j ∈ k

for every node M_i do

    for every vacation period H_j that doctor M_i is available for do

        add edge (M_i, H_j) with capacity 1 to G

#Solving part of algorithm, this code will maximize the flow given the pictured graph:

for each edge (u,v) ∈ E do

    (u, v).f = 0

while there exists an s, t path p in G_f do

    $c_f(p)$ = min{ $c_f(u,v)$: (u,v) ∈ p do

    for each edge (u,v) ∈ p do

        if (u,v) ∈ E then

            (u,v).f += $c_f(p)$

        else

            (v,u).f -= $c_f(p)$

#checking if solution is valid:

maxFlow = sum((H_j, T).f) for all j ∈ k

if maxflow ≥ N

    return “yes”

else

    return “no”

Runtime Analysis:

It was noted in class that the run time of this algorithm is: $O(VE^2)$

The size of V and E can be calculated as:

There is one vertex for each doctor, one for each vacation period, a sink and a source:

$$O(V) = O(n + k + 2) = O(n + k)$$

There is an edge connecting each doctor to the source, one connecting each vacation period to the sink, and then up to k edges for each doctor to connect a doctor to all vacation periods they are available for, and since there are n doctors:

$$O(E) = O(n + nk + k) = O(nk)$$

Putting this all together, the runtime is:

$$O(VE^2) = O\big((n + k)(n^2k^2)\big) = O(n^3k^2 + n^2k^3)$$

Problem 5

(a)

To analyze the SkewedHeapMerge operations at node x, we can denote $w(x)$ as the weight and $c(x)$ as the amortized cost.

We will assign each operation a cost, and if the computed cost is lower then it can be stored as credit on nodes and can be used to pay for other operations.

Initializing weights: at first all weights are set to 1.

Perform SkewedHeapMerge: This process is centered around swapping the left and right children along the rightest path of the two heaps.

Therefore, when a node is merged with another node we can update the weights to reflect that. We also set a cost of \$1 for a swap operation. Additionally, we deposit a penalty amount anytime we move a heavy child to a right child position.

The total cost of the SkewedHeapMerge operation is the sum of the actual costs incurred during the swaps.

To prove that the amortized time is $O(\log_2 n)$, we need to show that the cost is $O(n\log n)$

Analysis on the root to leaf path:

(b)

Insert Operation: When we insert a node, we initially set its weight to 1. The insertion process may involve merging subtrees.The amortized cost for the insert operation is $O(\log_2 n)$ due to the SkewedHeapMerge operation.

DeleteMin Operation: Deleting the minimum element involves merging its left and right children. The amortized cost for the delete operation is $O(\log_2 n)$ due to the SkewedHeapMerge operation.