CSCI-GA.1170-001 Problem 11-1

Ankit Sati

TOTAL POINTS

36 / 21

QUESTION 1

1 Minimum Spanning Trees **36 / 21**

part b

✓ **+ 1.5 pts** *correct for case i=0*

✓ **+ 3.5 pts** *correct for case i > 0*

part d

✓ **+ 1 pts** *part i) w(i, j) > w(i, c[i])*

✓ **+ 1 pts** *part i) w(i, j) < w(i, c[ij])*

✓ **+ 1 pts** *part i) w(i, j) = w(i, c[i])*

✓ **+ 2 pts** *part ii) correct*

✓ **+ 1 pts** *part iii) concluding T' contains P*

part h

✓ **+ 2 pts** *Iterate on P*

✓ **+ 2 pts** *Update weights*

Homework 11-1

(a) Assumption - $\mathcal{Z}$ exist

where $\mathcal{Z}$ is any arbitary cycle in P. with V_0 as the smallest among all V_i in this cycle.

Let $X(V_0)$ denote the edge of V_0 in P.

$\therefore$ $X(V_0)$ gives minimum edge of V_0

Consider 2 cases. $\therefore$

(i) Case 1 → All weights are equal for all edges in P.

Hence, for $\mathcal{Z}$ to exist

if $X(V_0) = (V_0, V_i)$

then $X(V_1) = (V_1, V_2)$ and $w(V_0, V_1) > w(V_1, V_2)$

otherwise $X(V_1)$ would be (V_0, V_1) and cycle would break here itself

Similarly, $X(V_2)$ have to be (V_2, V_3)

$\vdots$

$X(V_{k-1}) = (V_{k-1}, V_k)$

and for.

$X(V_k) = (V_k, V_0)$

$\Rightarrow w(V_0, V_1) > w(V_1, V_2) > \cdots > w(V_{k-1}, V_k) > w(V_k, V_0)$

However if $w(V_0, V_1) > w(V_k, V_0)$

then $X(V_0)$ should be (V_k, V_0)

and (V_0, V_1) will not exist.

(ii) Case 2. ⇒ Any two or more consecutive weights are equal.

Assume the equal weight edges be (v_{j-1}, v_j) & (v_j, v_{j+1})

$\therefore \quad X(v_0) = (v_0, v_1)$

$X(v_1) = (v_1, v_2)$

$\vdots$

$X(v_{j-1}) = (v_{j-1}, v_j)$

$X(v_j) = (v_{j-1}, v_j)$

as $\omega(v_{j-1}, v_j) = \omega(v_j, v_{j+1})$ and $v_{j-1} < v_{j+1}$

So we choose (v_{j-1}, v_j) over (v_j, v_{j+1})

and the cycle breaker as.

$X(v_{j+1}) = (v_{j+1}, v_{j+2})$

For $j = k$;

$\omega(v_k, v_0) = \omega(v_k, v_{k-1})$

then we choose (v_k, v_0) and not (v_k, v_{k-1}) as $v_0 < v_{k-1}$

Hence cycle breaker at (v_k, v_{k-1})

Hence both the cases show that our assumption is wrong.

⇒ 2 does not exist

Conclusion Proved.

(ii) Claim : " If an addition of an edge $e = (v, v)$ to P causes a cycle Z, then e has the maximum weight among the edges in Z"

→ Proof :-

Let $x(v)$ denote minimum edge of v in P.

(i) Case 1 : $i = 0$

v_0 is the node that either shares or does not have its minimum edge in Z.

* when v_0 shares its minimum edge.

So $x(v_0) = (v_0, v_1)$

$x(v_1) = (v_1, v_0)$

$x(v_2) = (v_2, v_1)$

$\vdots$

$x(v_k) = (v_k, v_{k-1})$

and $e = (v_k, v_0)$

So we can say that.

$\Rightarrow w(v_0, v_1) < w(v_1, v_2) < \cdots < w(v_{k-1}, v_k)$ and $w(v_k, v_0) > w(v_{k-1}, v_k)$

as otherwise minimum edge of v_k would have been (v_k, v_0) if.

$w(v_k, v_0) < w(v_{k-1}, v_k)$

$\therefore w(v_k, v_0) > w(v_{k-1}, v_k) > \cdots > w(v_1, v_2) > w(v_0, v_1)$

$\therefore$ Claim holds true in this case.

* when v_0 does not leave it minimum edge. i.e

$$x(v_1) = (v_1, v_0)$$
$$x(v_2) = (v_2, v_1)$$
$$\vdots$$
$$x(v_k) = (v_k, v_{k-1})$$

& $e = (v_k, v_0)$

$\Rightarrow w(v_0, v_1) < w(v_1, v_2) < \dots < w(v_{k-1}, v_k)$ —— (1).

and $w(v_k, v_0) > w(v_{k-1}, v_k)$ —— (2).

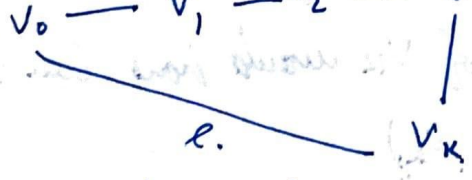
as otherwise minimum edge of v_k would have been (v_k, v_0) if $w(v_k, v_0) < w(v_{k-1}, v_k)$.

From (1) & 2 above,

$w(v_k, v_0) > w(v_{k-1}, v_k) > \dots > w(v_1, v_2) \geq w(v_0, v_1)$

Therefore the claim is true.

(ii) Case 2. $i > 0$.



$\Rightarrow$ when v_i shares its minimum edge:

$\therefore x(v_i) = (v_i, v_{i-1})$

where $w(v_i, v_{i-1}) < w(v_i, v_{i+1})$

$\therefore w(v_0, v_1) > w(v_1, v_2) > w(v_2, v_3) \dots > w(v_{i-1}, v_i)$ —— (3)

(1)

and now,

$x(v_{i+1}) = (v_{i+1}, v_i)$

$\vdots$

$x(v_k) = (v_k, v_{k-1})$

$\Rightarrow \omega(v_k, v_{k-1}) \geq \dots \geq \omega(v_{i+1}, v_i)$ ---- (2).

For edge $e(v_k, v_0)$

$\omega(v_k, v_0) > \omega(v_0, v_1)$ and $\omega(v_k, v_0) > \omega(v_k, v_{k-1})$ } ---- (3) these have to be true.

otherwise edge e would be the minimum edge for v_0 or v_k.

From the above equation (i) (ii) & (iii)

edge e has the maximum weight.

$\Rightarrow$ when v_i does not have its minimum edge in 2.

$\Rightarrow x(v_{i-1}) = (v_{i-1}, v_i)$

& $x(v_{i+1}) = (v_i, v_{i+1})$

$\vdots$

$x(v_2) = (v_2, v_3)$

$x(v_1) = (v_1, v_2)$

$x(v_0) = (v_0, v_1)$

$\Rightarrow \omega(v_0, v_1) > \omega(v_1, v_2) \dots > \omega(v_{i-1}, v_i)$ ---- (1).

and $x(v_{i+1}) = (v_{i+1}, v_i)$

$\vdots$

$x(v_k) = (v_k, v_{k-1})$

$\Rightarrow \omega(v_k, v_{k-1}) \geq \dots \geq \omega(v_{i+1}, v_i)$ ---- (2).

Now with the edge e.

$\omega(V_k, V_0) > \omega(V_k, V_{k-1})$ & $\omega(V_k, V_0) > \omega(V_0, V_1)$ → (3) This has to be true otherwise 'e' would be the minimum edge for V_k or V_0.

From above i, ii & iii

edge e has the maximum weight.

⇒ From all three cases, edge e has the maximum weight.

This shows that our claim is true.

(c) Assumption → C[1 ..... n] is initialized with -1 P = set(). (empty set)

Algorithm:-

Compute (G):

```
1.  For v in G.V:                          ┐
2.      temp = ∞                           │
3.      For i in G.Adj[v]:                 │
4.          if ω(v,i) < temp:              │  O(|V| + |E|)
5.              temp = ω(v,i)              │
6.                  C[v] = i.              ┘
7.      Sum = 0                            ┐
8.      For v in G.V.                      │
9.          if C[v] == -1:                 │
10.             Continue                   │  O(|V|).
11.         else if v == C[C[v]]:          │
12.             C[C[v]] = -1               │
13.         Sum += ω(v, C[v])              ┘
```

Cont →

14. P. append (v, c[v])
15. RETURN P, Sum

Correctness of Algorithm:

First we generate C by travelling each & every vertex & then each edge of that vertex

② We update C values accordingly when we find an edge with a weight less than the edge made by endpoint stored in C for that vertex.

③ This happens in $O(|V| + |E|)$ as we visit each edge correctly atleast once.

Now, we generate P by ~~taking~~ traversing C. when an edge is repeated, its endpoint c. value is the vertex i.e.

i.e. $c[x] = y$.

$c[y] = x$.

To avoid duplicate edges, we check this condition on every edge & set $c[c[y]] = -1$. this way we visit but the condition fails. and we skip it $c[u] = -1$.

Rest all other times, we just keep adding to the sum & appending to P.

Runtime $= O(|V| + |E|)$ { complete in linear time }

first loop runs in $O(|V| + |E|)$ & 2nd loop runs in $O(V)$ other operations take constant time.

(d) Proove: Resulting T' is an MST

(d) We need to consider all 3 cases.

① Case 1 : $w(i,j) = w(i, c[i])$

It does not change the MST in any ways as the weight also remains same.

⇒ If T was an MST, then T' after swapping will be an MST.

② Case 2: $w(i,j) > w(i, c[i])$

→ Here, we actually improve the MST after swapping.

→ If T was an MST, then after swapping T' will always be an MST, as the weights of that edges have infact decreased.

③ Case 3: $w(i,j) < w(i, c[i])$

→ In this case is not possible, as accordingly to defination of $c[i]$, it should store the endpoint with the minimum-weight.

→ It is not possible as $j = c[i]$ which is case 1, hence resulting tree is an MST.

⇒ From 3 cases, we have proved that resulting tree is an MST.

To Proove → swap never removes some $[k, c[k])$ for $k < i$.

Proof → This is indeed true as the line "if $j < i$ then return SWAP (j, T', c)" takes care of cases where some $(k, c[k])$ might have been removed.

⇒ Here $j = k$, and we call SWAP (k, T', c) in which we swap (k, e) with $(k, c[k])$ if $(k, c[k])$ don't $\in T'$.

~~if edge $(k, c[k])$ has been removed.~~

(k, x) is the i^{th} edge in path $1 \to c[1]$.

→ Hence this overall swap never removes any $(1, c[1])$ for $1 < k < i$.

Therefore this is proved.

Finally
To prove → the Algorithm iterates all over the vertices and checks if $(i, c[i])$ is part of T.

If it exist we perform the swap in a way that resulting T has the edge $(i, c[i])$

→ In this way we iterate & check for all vertices and finally return T' which indeed contain all of P.

(e) P has sampled from G.

∴ for each $\in$ G.V, there is an edge in P.

→ According to equivalence relation returns (u, v). Then we can define an equivalence class $[v]$, where u & v will belong to one equivalence class only or they are " Equivalent modulo P"

→ So two vertices (v & v) are reduced to one equivalent class.

→ If we consider the edge case where each equivalence class has two vertices, then the total number of equivalence classes will be exactly half of number of vertices

i.e. $|V'| = \frac{|V|}{2}$ in this case.

→ Now, if equivalence class has more than 2 vertices, the total number of equivalence classes will surely be less than $\frac{|V|}{2}$.

→ Like so, any equivalent class can have multiple vertices with minimum being two. Hence. $|V'| < \frac{|V|}{2}$ for these cases.

Concluding $|V'| \leq \frac{V}{2}$ where $V' = \{[v] \;.\; v \in V\}$

Proved.

(f) $V' = \{ [v] : v \in V \}$

$E' = \{ ([u],[v]) : \text{weight} = \min \{ w(u',v') : v' \in [u], v' \in [v], (u',v') \in E \} \}$

Algorithm

```
Compute COST ( ) ;
1   Cost = 0
2   Sum = 0
3   while |G-V| > 1 :
4         P, Sum = COMPUTE P (G)
5               Cost + = Sum
6                  G = COLLAPSE G( G, P)
7     RETURN Cost.
```

Correctness :-

COMPUTE P will generate set of edges with minimum weight. essentially these are returning equivalent classes with atleast 2 vertices with connection shown in P.

Now COLLAPSE G will try to return a graph with V', E' as defined above such that it will return one representative for each equivalent class and all its edges $\in E'$ with other representatives of equivalent classes if its exists.

Since our MST has to join these parts if possible, the new graph G' will be again ran on COMPUTE P & COLLAPSE G all the way until we are left with one vertex.

All the while we keep adding sum of each P to our total cost, & finally return it.

//.

(g) Assumption :- Runtime of COLLAPSE G_i = $O(m+n)$ and runtime of COMPUTE = $O(m+n)$. from c

∴ Runtime inside each interation of outer loop = $O(m+n)$

From part (e), we know.

$|V'| \leq \frac{|V|}{2}$.

i.e number of vertices get halved atleast in each iteration.
Taking worst case, $(|V'| = \frac{|V|}{2})$.

After each iteration, number of vertices get halved.

⇒ Number of iteration = $\log(|V|) = \log n$.

⇒ Total Runtime of COMPUTECOST()
= $O((m+n)\log n)$

(h) Generate v':

→ Modifying the DFS-VISIT algorithm such that it checks if $u, v, \in P$ like in.

for each $u \in G_{adj}[v]$:
if $u.color == WHITE$ & $(u, v) \in P$.

→ On calling DFS with the above modification we can get a tree for each equivalence class on each return from DFS-VISIT from DFS.

→ Since the modification is in constant time, runtime for this is same as DFS. $O(m+n)$.

Fill in V. equivalence.

1. Iterate through each node in all the equivalence. class trees and find the minimum of each class by keeping a temporary variable.

2. Now we iterate through each tree again & just fill in. the equivalence. with the value from temp variable.

3. This is done by a tree traversal algorithm with slight modifications happening in constant time.

∴ Runtime $O(V) = O(n)$.

→ Generate E'

① We keep the temp variable that stores the min weight.

② We iterate through each. vertex & its neighbours & check if they belong to different equivalent classes. If they do then we store that classes. edge E' with that weight as. $w(u,v)$

③ Now, if those classes. edge $([u]\ [v])$ already. exists in E' then we check. the stored weight with the new. weight & update accordingly.

④ This may we. add edges to E' in runtime $O(m+n)$

Total Runtime =. $O(m+n) + O(n) + O(m+n)$

$= O(m+n)$

1 Minimum Spanning Trees 36 / 21

part b

✓ **+ 1.5 pts** *correct for case i=0*

✓ **+ 3.5 pts** *correct for case i > 0*

part d

✓ **+ 1 pts** *part i) w(i, j) > w(i, c[i])*

✓ **+ 1 pts** *part i) w(i, j) < w(i, c[i])*

✓ **+ 1 pts** *part i) w(i, j) = w(i, c[i])*

✓ **+ 2 pts** *part ii) correct*

✓ **+ 1 pts** *part iii) concluding T' contains P*

part h

✓ **+ 2 pts** *Iterate on P*

✓ **+ 2 pts** *Update weights*