Assignment 4

1. Algorithm: -
   i) Run DFS on the graph G, making sure every vertex is visited at least once and make note of the finish time of every node
   ii) Reverse every edge in the graph and set r(v) of every vertex as to itself.
   iii) Start running DFS on the reversed Graph according to the decreasing order of the finish times of vertex
   iv) Now for every depth-first forest create a set of nodes present in the forest (connected components) and keep note of maximum influential vertex *l(u)* of each set
   (consider this as r(v) of that set).
   (step iv will return us acyclic graph of connected components with the maximum *l(u)* on each)
   v) Replace all sets of connected components *c~* with a new vertex *v~* with *r(v~)* as max influential vertex in that component and create a new graph G~.
   vi) Run Topological sort on the new graph G~ formed in step 5 and compare the r(v) of every vertex with its predecessor.
   If $r(V_{child})$ is smaller than $r(V_{parent})$
   then $r(V_{child}) = r(V_{parent})$
   else keep $r(V_{child})$ as it is.
   vi) For all nodes v~ representing a connected component (c~) set $r(v_i)$ as a r(v~) for all $v_i \in c$~
   vii) Return all the vertices from the graph G~.

Analysis: –
- Step 1 runs a DFS hence complexity of that step will be $\Theta(m + n)$ where m and n represents edges and vertices respectively.
- Step 2 requires traversing through all the vertex once and thus its time complexity will be $\Theta(n)$
- Step 3 again runs DFS on the reversed graph so again time required will be $\Theta(m + n)$
- Step 4 & 5 traverse through all vertex and replaces all connected vertices with single node and calculate r(v~), so this step will take maximum $\Theta(n)$ time.
- Step 6 will visit every vertex in pre-requisite manner so running time would be $\Theta(m + n)$
- Step 7 can maximum take $\Theta(n)$ time as it will end up going through all nodes in the worst case.
- Thus, overall running time complexity of this algorithm would be ***$\Theta(m + n)$***

Correctness: -

Lemma1:

r(v) for a strongly connected component is the vertex with of maximum influence l(v) among all vertices present in that component.

Proof: Proof by contradiction.

r(v) is the vertex that has largest influence among all vertices reachable from v in graph G. …***Given***

Let us assume that there exists a vertex in a strongly connected component whose r(v) doesn`t have the maximum influence in the connected components.

For Graph G(V, E)

$\{ \exists\, v \in V$ for which $r(v) = v`$ and $\exists\, v`` \in V$ such that $l(v`) < l(v``) \}$

Now according to the definition of connected components, it is a set of vertices in which every vertex is reachable from every other vertex in that set. Thus, v`` will be reachable from every other vertex in V

and vice-a-versa. So, v`` is also reachable from the point v and as per the given definition of r(v) it should reflect the maximum influence vertex which is v``.
This contradicts our assumption and hence lemma is proved.

Lemma2:

In a graph G(V, E), if a vertex x ∈ V reachable from a vertex y ∈ V (there is path from y to x), then in graph G^T, we can reach to the vertex y from x.

Proof: Proof by induction.

Let us assume that the lemma two works when there are at max m vertices between source and destination vertex such that 0 < m < n-1

Base Case –

When m = 0

- This means there is no path between destination and source node, thus even in case of (transpose) reversed graph there will not be any path between source and destination.

When m = 1

- This means that vertex u has a direct edge from vertex V, in case of a transposed graph the edge will be reversed and now vertex V will have a direct edge coming from U.

(G) V → U (G^T) V ← U

Induction Step –

When i = m+1

- We know that our lemma works when i = m thus in a graph G^T (#vertices = m) there is a path from U to V and in graph G there is a path that goes from V to U … (1)
- Now let us add one more vertex U~ (our new destination) at the end of the graph in the G thus now number of vertices between first and last node is m + 1.
  We do this by adding a direct edge from our previous destination U to the new destination U~. Let us call out new graph as G~.
- When we reverse the graph to $G\tilde{}^T$, we have an edge going from U~ to U … (2)
- From (1) we know that there is a path from U to V and from (2) we have an edge from U~ to U
  So, we have a path from U~ to V
- Thus, this proves our lemma.

Corollary of lemma 2:

- r(v) denote the vertex in G that has the largest influence among all vertices reachable from v in G
- From second lemma r(v) will denote the vertex in G^T that has the maximum influence among all source vertex that could reach to v in G^T.

Solution Proof:

- Step 6 of algorithm runs topological sort and checks the maximum influence source vertex that from which we can reach to other vertices in the graph and propagate the largest r(V_{parent})
- From first lemma, all vertices in a connected component will have same value of r(v) which will be the maximum influential source vertex from which we can reach till v.
- Thus, from above two statement we can say that our algorithm is correct.

2. Solution: -

Logic –

- If a graph G consists of a vertex that has a back edge to its parent vertex then the there is a cycle in that graph, similarly if while finding a path from source to destination we happen to reach to the source vertex again then the path contains cycle.
- Change vertex structure to save details of vertices travelled so far in it call it as a route. Basically, this will save the parent details for every vertex and every time you visit a new vertex you save the path taken to reach to that vertex.
  The route will consist of list of vertices.
- A vertex is considered as a loop vertex if it can reach to its parent vertex.

  X ⟶ Y

  For an edge e = connecting vertex {x, y}, vertex x is a loop vertex if route of x contains y
- Maintain a list to save all the loop vertex(loopVertices).
  In above example loopVertices will have 'x' in it.
- Return True, if route of a destination vertex has any vertex that is present in loopVertices.

Procedure –

DetermineCertainCycle (G, SourceVertex v, DestinationVertex u):

```
Stack <Node> s = new Stack ()
HashSet <Node> loopVertices = new HashSet < Node> ()
list <Node> visitedVertices = new ArrayList<loopVertices> ()
s.push(v)

/*   This is sort of DFS from the source vertex v   */
while(! s.isEmpty()) {
    Node n = s.pop()
    forEach(ChildNode vn : children of n)   {
        pathTill = n.route
        pathTill.add(n)
        vn.route.add(pathTill)
        if (! visitedVertices contains vn)   {
            s.push(vn)
        }
        else if (visitedVertices contains vn  && n.route contains vn)  {
                loopVertices.add(n)
                if (vn == v) {
                   return True
                }
        }
        visitedVertices.add(n)
    }
}

if (! visitedVertices contains u)  {
    return False
}

/* Check whether route of destination vertex has a loop or not */

forEach(Node intermediateV : u.route)  {
    if (loopVertices containsKey(intermediateV))   {
        return True
    }
}

return False
```

Analysis –

- First while loop is nothing but DFS from source vertex v, thus the running time of first loop will be $\Theta (m + n)$ where m and n represents the number of edges and vertices respectively.
- The forEach loop will go through each vertex in the route of the destination node and check whether it presents in the list of loop vertices or not. We can do the search operation (contains) in constant time by making use of some data structure like Hash Set instead of list. In worst-case destination node will have all remaining nodes in route thus, the running time of the second loop will be $\Theta (n - 1)$ where n represents the number of vertices in the graph.
- Therefore, ultimately the total running time complexity of the algorithm will be $\boldsymbol{\Theta (m + n)}$

Correctness –

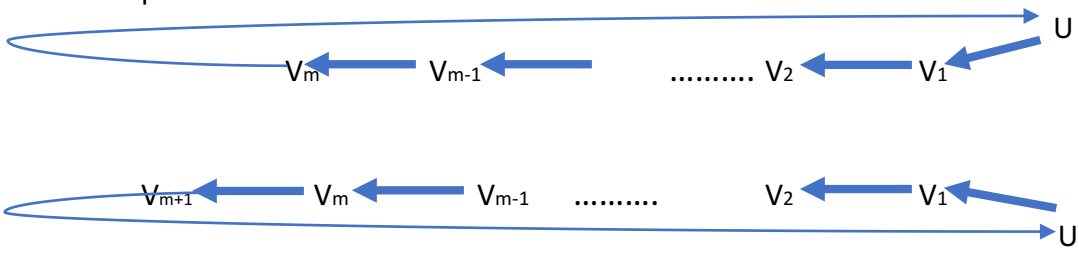
Lemma1:

If in a graph G, there exists a back edge from a child vertex to one of its ascendants vertex then the graph contains cycle.

Proof: Proof by induction.

- Let us assume that the lemma is true for child depth 'm',
  that is lemma holds True if there is a back edge after m vertices where 0 < m < n
- Base Case –
  Let i be an integer value such that 0 < i < m
  When m = 0
  When there are no child nodes at that time there won`t be any back edge so graph will be acyclic.
  When m =1

- Induction Step –
  Suppose the lemma works when i = m; now suppose we remove the back edge from the V_m and point it to a new node V_{m+1}.



  If suppose V_{m+1} has a back edge that goes to U then again, we end up having a cyclic graph containing U, $V_1, V_2, \ldots, V_m, V_{m+1}$ vertices.
- Thus, from the induction proof our lemma is correct.

Solution Proof:

- From lemma, it is proved that by keeping track of the all ascendant vertices (that is the vertices travelled through to reach to current vertex) of a current vertex and by checking whether the current vertex goes to its ascendant we can find out the availability of cycle in path of source and destination.

Alternative Solution –

- Run DFS from source, save all the vertex as v~, return false if destination not visited.
- Reverse all the edges between vertices present between v~.
- Run DFS again from the destination vertex and save all vertex reached as v~~.
- Now create a new graph G using vertices v~~ and all edges between those edges.
- Using the finish time algorithm find if the graph has a loop or not. If present return true or false. (Complexity and correctness of this algorithm will be same as that of above one.)

3. Solution: -

Logic –

i) For every process calculate t_i/wt_i and save it in an array list
ii) Sort the list in the increasing order t_i/wt_i
iii) Schedule the process according to the sorted list (process with minimum t_i/wt_i is executed first)
//Step 4 Is Not Required.
iv) Calculate the total weighted finish time for the schedule formed in step 3
v) Return the schedule formed in step 3.

Analysis –

- First step will take $\Theta(n)$
- Second step is sorting and that takes $O(nlogn)$ time
- Step 3 is scheduling of process which again takes $\Theta(n)$ time.
- Thus, the overall solution of this problem will be $(nlogn)$

Correctness –

Lemma1:

There exists an optimal solution that contains activity with smallest ratio of t_i/wt_i.

Proof:

Suppose the optimal solution OPT1 contains an $i^{\sim} \in$ processes and not i which has lower time by weight ratio.

Let us create a new optimal solution OPT2 by replacing the process $i^{\sim}$ by i.

$OPT2 = [\{ OPT1\} - \{i^{\sim}\} + \{i\}]$

As we know, $t_i/wt_i < t_{i^{\sim}}/wt_{i^{\sim}}$

Thus {OPT2} < {OPT1} and hence { OPT2 } will also be an optimal solution.

Lemma2:

Optimal Substructure property proof.

Proof:

By Induction on the number of processes in the input.

- Base Case –

When n = 0

When there is no process at that time the time required will be zero.

- Induction Step –

Let`s say that our solution is correct n < m & m > 0

Now we need to prove our algorithm returns optimal solution for n=m

Suppose for input i with m activities our algorithm returns set S.

Suppose there is a process i with least ratio of time by weight.

By Lemma, ∃ optimal solution that contains i.

Consider $I^{\sim} = I - \{ i \}$

Note that OPT – {i} is optimal for $I^{\sim}$.

Our solution for $I^{\sim} = S - \{i\}$

$|S - \{i\}| = |OPT - \{i\}|$

$|S| = |OPT|$

4. Solution: -

Logic –

i) Depending on whether the number of characters are even or odd, add a dead character Ø with frequency 0. Add dead character if total characters are even.
ii) Sort all the characters according to the increasing order of their frequencies.
iii) Run Huffman encoding algorithm by choosing three characters with minimum frequency at a time instead of two at a time like binary encoding.

Procedure –

```
HuffmanTrinary (List characters):

charac deadChar = new charac("Ø", 0)

UnionSet<charac> unionChars = new UnionSet()

if (characters.length % 2 == 0)   {
    characters.append(deadChar)
}

forEach(charac ch :characters) {
       unionChars.add(ch)
}

Map <Codes> encodes = new HashMap ()

int compare(charac c1, charac c2)  {
    return c1.freq < c2.freq ? -1 : 1
}

While (sortedcharacters.lenght > 3)  {
       saveValueOfEachInMap(unionChars, getLeastThree(sortedcharacters)
       Charac newCh = MergeToNewCharac(removeLeastThree(sortedcharacters))
       sortedcharacters.add(newCh)
}

return getIndividualEncodesFrom(unionChars)
```

Analysis –

- First loop will have complexity of $\Theta(n*\log(n))$
- Even second loop will have complexity of $O(n*\log(n))$
- Thus, overall running complexity of the algorithm $O(n*\log(n))$

Correctness –

Lemma1:

Let *x, y* and *z* are the characters with lowest frequency then there exists a solution in which all three of them will be siblings.

Proof:

- Let T~ is an optimal solution that has w instead of x in the lowest frequency siblings and frequency of *w* is larger than frequency of *x*
- Now let`s try to swap the *w* & x and create a new Tree T~~
- There the total cost required for new Tree will be as follows –

$$C(T\tilde{}\tilde{}) = C(T\tilde{}) + f(x)[\, d_T(w) - d_T(x)\,] + f(w)[\, d_T(x) - d_T(w)\,]$$
$$= C(T\tilde{}) + [\, f(x) - f(w)\,]*[\, d_T(w) - d_T(x)\,]$$

- Since frequency of *x* is less than *w* thus cost required by new tree will be less than previous tree.

$$C(T\tilde{}\tilde{}) <= C(T\tilde{})$$

- As, we said T~ is an optimal solution, T~~ will also be an optimal solution. Thus, we proved our lemma that there exists an optimal solution that has *x, y* and *z* will be siblings.

Lemma2:

Suppose T~ be an optimal tree which has *x, y* and *z* characters at lowest level. Then the T~ is said to be optimal iff tree $T^{\sim -3}$ without those three characters is also an optimal solution.

Proof: Proof by contradiction.

C(T~) is the most optimum solution tree for the given set of characters. Cost for this can be given by

$C(T^{\sim}) = \{ f(x)d_T(x) + f(y)d_T(y) + f(z)d_T(z) + C(T^{\sim -3}) \}$ …(1)

The cost of tree T~ is nothing but cost required for *x, y, z* and cost of remaining tree.

Now suppose, the optimal solution tree for characters other than *x, y* and *z* is T*,

then $C(T^*) < C(T^{\sim -3})$ …(2)

We can add the three characters *x, y* and *z* in the tree T* and create another solution. And that total cost would be $C(T^{*+3}) = \{ f(x)d_T(x) + f(y)d_T(y) + f(z)d_T(z) + C(T^*) \}$ …(3)

From (1), (2) & (3) we can say that, $C(T^{*+3}) < C(T^{\sim})$

This contradicts our assumption that C(T~) is the most optimum and hence to become an optimal solution even the substructure of the solution must be optimal.

Thus, $C(T^{\sim -3})$ should be the optimal solution for characters without *x, y* and *z*

5. Solution: -

Procedure –

```
SendInvitation(List people):

Map<person> invitationMapping = new HashMap<person> ()

forEach(person p : people) {
     invitationMapping.add(p)
}

forEach(person p : people) {
        if (p.knows < 4 or p.notKnows < 4){

           invitationMapping.remove(p)

           forEach(person rp : knows(p)){
                  rp.knows = rp.knows – 1
                  invitationMapping.add(rp)
           }
        }
}

return getInvitedPeople(invitationMapping)
```

Analysis –

- First loop will have complexity of $\Theta(n)$
- Even second loop will have complexity of $O(n*n)$ in the worst case
- Thus, in the worst case running time complexity of the algorithm will be $O(n^2)$

6(a). Solution: -

Logic –

i) For each edge in the graph, multiply the weight by -1
(make the weights negative such a way that maximum wt vertex becomes minimum & vice-versa).
ii) Apply Prims`s algorithm and create a minimum spanning tree
iii) Multiply each edge in the MST found in second step by -1
iv) Return the Tree formed in step 3.

Analysis –

- Complexity of algorithm will be equal to the complexity of the prim`s algorithm
- Thus, it will be O(nlogn + m)

6(b). Solution: -

Logic –

i) Find maximal spanning tree just like the one done in question 6(a) of this assignment and create a list of edges that are present in the mst
ii) Remove each edge e belonging to the MST found in first step form the given graph G
iii) Return list of edges remained in step 2

Analysis –

- Complexity of algorithm will be equal to the complexity of the prim`s algorithm

Thus, it will be O(nlogn + m)

Collaborators –

Jhalak Maheshwari
Sourabh Punja
Manish Yadav