

Assignment – 7

Question 1: SpanTree.java

Verbal Description:

This program finds the minimum spanning tree in the given graph. If the minimum spanning tree is found then the total weight is the output, else it will output -1 which indicates no minimum spanning tree is found. Minimum Spanning Tree is a subset to a connected graph G, where all the edges are connected but the subset does not contain any cycle. This implies that if a graph G has N nodes then the minimum spanning tree will have at most N-1 edges.

The minimum spanning tree will have a total weight less than or equal to the weight of every other spanning tree.

For this program, Merge-sort was implemented to sort the edges. After that Kruskal's algorithm which is also known as Union-find algorithm was implemented to find the minimum spanning tree. Apart from that, LinkedList and Queue was also implemented.

Kruskal's algorithm uses a greedy approach which implies that after the graph has been created it has to be sorted in decreasing order based on the weight of each edge.

If the given edge has value 1 associated with it, then unify it and compute their weight.

In the next step, check the remaining edges and if that can be added to the minimum spanning tree then add it as long as it does not create a cycle and if they belong to the same set.

If it does not belong to the same set, then it is unified and again the total weight is computed.

Once all the edges have been checked the total computed weight of the edges belonging to the minimum spanning tree is the required output.

Pseudo-Code:

```
kruskal_algorithm():  
    weight_edge = 0  
  
    check if the graph is connected:  
        if the graph contains more than one graph:  
            output -1  
  
    for j = 0 till total no. of edges:  
  
        combine edge[u,v] if kruskal_set does not contain a cycle:  
            weight_edge = weight_edge + edges[j].weight  
  
        if kruskal_algorithm contains a cycle:  
            return -1  
  
    sort the edges array in increasing order  
    for i = 0 till total no. of edges:  
  
        combine edge[u,v] if kruskal_set does not contain a cycle:  
            weight_edge = weight_edge + edges[i].weight  
  
    Return the weight_edge
```

Proof of Correctness:

Assignment – 7

The algorithm makes sure to exclude the edges that will create a cycle in the graph. Due to the implementation of the Kruskal's method, which is fairly precise, the portion of the algorithm that computes the weight at each iteration is precise as well.

All the edges added in the minimum spanning tree have their weight added as well, so in the end the total sum of the weight is the output.

Running Time Estimate: $O(m \log n)$

Brief Reasoning:

This algorithm runs in $O(m \log n)$ time complexity.

This is due to the fact that the remaining time complexities are dominated by the Kruskal's algorithm which takes $(n \log n)$ and the edges' weight-based Merge-Sort sorting takes $O(m \log n)$.

Question 2: NegativeCycle.java

Verbal Description:

The bellman-ford algorithm was used to solve this problem. Because it can deal with negative weights. In the presented graph, this task seeks to locate a negative weight cycle. If one is discovered, it will produce YES; otherwise, it will produce NO.

Heart of the problem is computed using:

$\text{vertex_dist}[v]$ = the total sum of the weights of the shortest path from start vertex to the end vertex.
The vertex_dist array is initialized to 100000 and then updated later.

In the case when n is the entire number of vertices, the bellman-ford algorithm will execute $n-1$ times.
The maximum number of iterations needed to locate a negative cycle in a graph is $n-1$.

The inner for-loop runs m times where m is the total number of edges in a graph.

A boolean variable is used to keep track of whether the vertex_dist value should be updated or not,

if cost found is greater than the one found during the last iteration then it will break out of the inner for-loop. During each iteration the vertex_dist array is updated using:

```
for edge  $u \rightarrow v$ 
    if (  $\text{vertex\_dist}[u] > \text{vertex\_dist}[v] + \text{cost}[u, v]$  ) :
         $\text{vertex\_dist}[u] = \text{vertex\_dist}[v] + \text{cost}[u, v]$ 
```

Pseudo-Code:

Bellman-ford algorithm:

Assignment – 7

cost[] -> weights of all the edges

vertex_dist[] -> keeps track of the total cost taken to reach each edge

edge -> total number of edges

vertices -> total number of nodes

hasNegativeCycle = false

```
for n = 0 to vertices-1:
    track = false
    for m = 0 to edges:
        if (vertex_dist[u] > vertex_dist[v] + cost[u,v]):
            vertex_dist[u] = vertex_dist[v] + cost[u,v]
            track = true
    if track == false:
        break
```

```
for i = 0 to edge:
    if (vertex_dist[u] > vertex_dist[v] + cost[u,v]):
        hasNegativeCycle = true
```

```
if hasNegativeCycle == true:
    print "YES"
```

```
if hasNegativeCycle == false:
    print "NO"
```

Proof of Correctness:

To ensure that every vertex is updated at least once, this technique assures that it will only take n-1 iterations to move from the start vertex to the end vertex.

Bellman-ford algorithm requires only n-1 iterations.

Running Time Estimate:

n -> outer for-loop

m -> inner for-loop

$O(n*m)$

Brief Reasoning:

This algorithm has a nested for-loop where the outer loop runs n-1 times because that is the number of iterations required to update all the edges at least once.

For the command line input: $O(n)$ n->total number of edges Nested for loop for bellman-ford algorithm.

Question 3:

Assignment - 7

Edmonds - Karp Algorithm

This is a maximum flow algorithm and is an implementation of Ford - Fulkerson method that uses BFS for finding augmenting paths.

For this algorithm, in every iteration the shortest augmentation path is chosen.

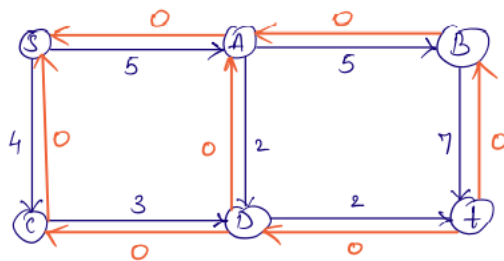
Run-time complexity for Edmonds-Karp algorithm is $O(m^2)$.

This is because the dependency on C has been completely removed.

Iteration : 1

$s \rightarrow$ start node

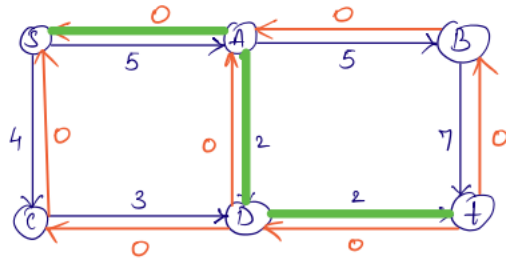
$t \rightarrow$ destination node



Current Residual Graph

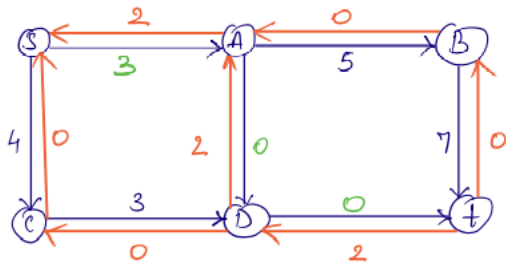
Assignment - 7

Augmenting path taken by the algorithm: $S \rightarrow A \rightarrow D \rightarrow t$



$$S \xrightarrow{5} A \xrightarrow{2} D \xrightarrow{2} t$$

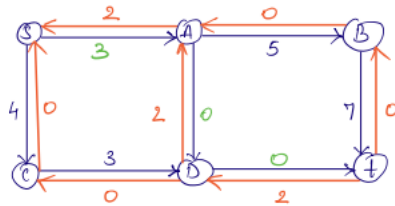
\therefore The maximum flow for this iteration will be 2.



Residual graph after the augmented path was chosen.

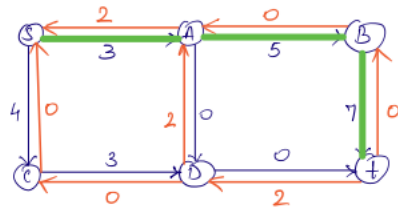
Assignment – 7

Iteration : 2



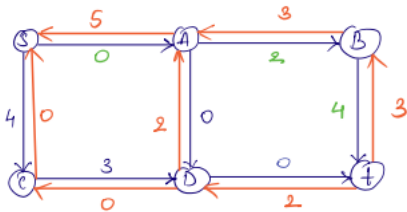
Current Residual Graph

Augmenting path taken by the algorithm : $S \rightarrow A \rightarrow B \rightarrow T$



$$S \xrightarrow{3} A \xrightarrow{5} B \xrightarrow{7} T$$

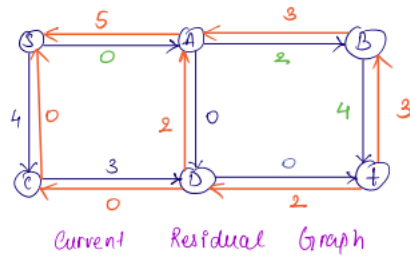
\therefore The maximum flow for this iteration will be 3



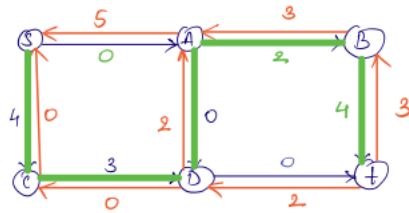
Residual graph after the augmented path was chosen.

Assignment – 7

Iteration : 3

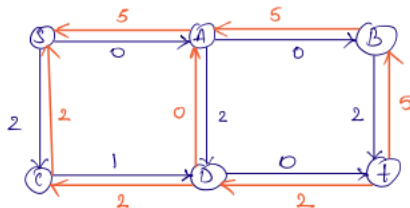


Augmenting path taken by the algorithm : $S \rightarrow C \rightarrow D \rightarrow A \rightarrow B \rightarrow t$



$$S \xrightarrow{4} C \xrightarrow{3} D \xrightarrow{2} A \xrightarrow{2} B \xrightarrow{4} t$$

\therefore The maximum flow for this iteration will be 2.
This graph uses a backward edge $D \rightarrow A : (D, A)$.



Question 4:

Verbal Description:

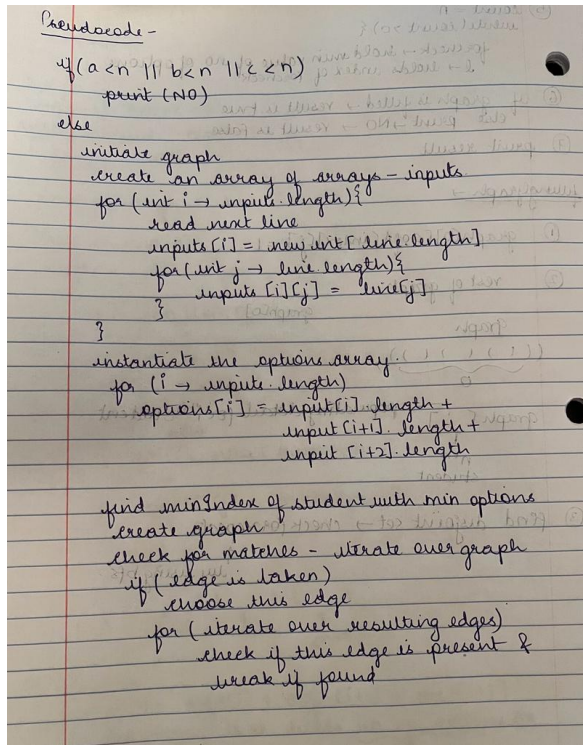
The goal of the question is to find out whether it is possible to get every child dressed in his/her chosen pair of hats, mittens and jackets keeping in mind the fact that no two children can have the same clothing.

Assignment – 7

The code is our best attempt to do the same using the concept of graph. We have tried to create a graph of choices of every child such that they are distinct. We have used the concept of adjacency matrix in order to keep the preferences of each child separate. Using this array of arrays, we have attempted to create a graph which blocks the node if it is a choice of that particular child (of that particular iteration).

Lastly the code checks for repeats in the graph created by using BFS i.e., visiting every node. If no repeats, then the code returns a YES followed by the nodes of the graph which are the possible combinations.

Pseudo-Code:



```
Pseudocode -
if (a < n || b < n || c < n)
    print (NO)
else
    initialize graph
    create an array of arrays - inputs
    for (int i = 0; i < inputs.length; i++)
        read next line
        inputs[i] = new int[inputs[i].length]
        for (int j = 0; j < inputs[i].length; j++)
            inputs[i][j] = inputs[i][j]
    }
    instantiate the options array
    for (i = 0; i < inputs.length; i++)
        options[i] = inputs[i].length +
                    inputs[i+1].length +
                    inputs[i+2].length
    find minIndex of student with min options
    create graph
    check for matches - iterate over graph
    if (edge is taken)
        choose this edge
        for (iterate over resulting edges)
            check if this edge is present &
            break if found
```

Proof of Correctness:

The code discussed is correct because a graph of possible combinations for the i th student is created. The graph itself is created such that it is checked with the current edge for repeats. If this edge is found, we halt because it means that this edge is seen before – hence we skip it. The graph is filled for every index according to the order – hats, mittens and jackets for each i – i.e., each child thereby preserving their preferences. In the end, the created graph is traversed using BFS to find possible combinations for a child. If such a traversal is possible, we print the combination along with a YES.

Running Time Estimate: $O(n^2(a + b + c))$

Brief Reasoning:

Two for loops to iterate over entire graph that run through the length of the graph – so n^2

Assignment – 7

Iterate over the resulting edges – which can change according to the input of hats/mittens/jackets – so
 $a+b+c$

Question 5:

Q1 : Longest Path

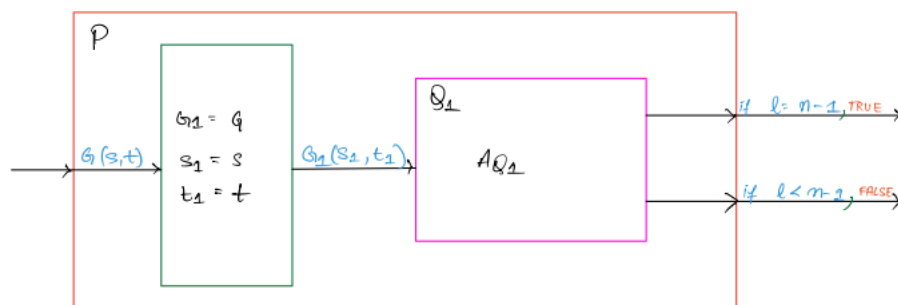
Input :

$$\begin{cases} G_1 = G(\forall w_{s_i} = w_{t_i}) \\ s_1 = s \\ t_1 = t \end{cases}$$

The input for Q1 remains the same as P.
This is because both are unweighted and undirected graph G and the two vertices s and t.

Output : In case of Q2, the result l of Q2, the value of 'l' is checked with the value of $(n-1)$.
If $l = (n-1)$ the machine will output TRUE
(which means that the graph G(s,t) contains a Hamiltonian path from s to t.)

If $l < (n-1)$ the machine will output FALSE.
(which means that the graph G(s,t) does not contain a Hamiltonian path from s to t.)



Poly-reduction for PE Q1

Poly-Reducible bound's correctness of Q1 with P.

Assignment – 7

If a longest path of length $(n-1)$ exists, it passes through each vertex. Since there must exist at least one vertex that cannot be reached without being visited twice, the longest path will not pass through all vertices if its length is less than $(n-1)$. Other paths that are shorter than the longest path would also not be able to reach all vertices if the longest path is unable to

do so. This demonstrates that there is no path connecting all vertices between s and t .

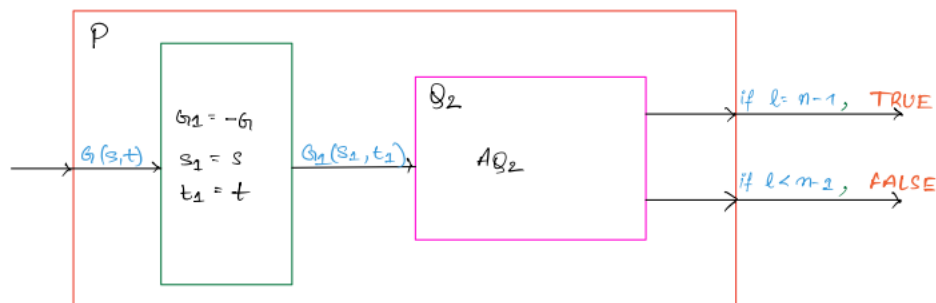
Q_2 : Shortest Path with Negative Weights

Input :
$$\begin{cases} G_1 = -G (\forall w_{i,j} = -w_{i,j}) \\ s_1 = s \\ t_1 = t \end{cases}$$

Construct graph G_1 , where G_1 is derived from G by changing every weight to its negation.

Output : In case of Q_2 , the result l of Q_2 , the value of ' l ' is checked with the value of $(n-1)$.
If $l = (n-1)$ the machine will output TRUE
(which means that the graph $G(s,t)$ contains a Hamiltonian path from s to t .)

If $l < (n-1)$ the machine will output FALSE.
(which means that the graph $G(s,t)$ does not contain a Hamiltonian path from s to t .)



Poly-reduction for $P \in Q_2$