CS 580 Algorithm Design, Analysis, And Implementation
Fall 2021 Vassilis Zikas HW 4

Problem 1

Collaborators: List students that you have discussed problem 1 with: Vishnu Teja Narapareddy, Tulika Sureka
We can use the dynamic programming approach to solve this problem.

Subproblem:
opt[i][j] denotes the maximum points the team can earn after the i^{th} round if j^{th} person plays that round. Here i varies from 0 to n-1 and j varies from 0 to 2, mapping the given arrays s.t J $\longrightarrow$ 0, E $\longrightarrow$ 1, and C $\longrightarrow$ 2.
For returning the array of who pushed the button in every round, we can maintain three strings for the three 1-D dp's which we are populating simultaneously. Initialize three empty substrings path_sara, path_eliz and path_charles for each of the team players and then keep appending as per the cases discussed below.

Base Cases:
Choose based on the max score and store the respective path in the structure

$$opt[0][0] = max\left\{ \begin{array}{rl} 0, & path\_sara = path\_sara + \text{"\_"} \\ J[0], & path\_sara = path\_sara + \text{"J"} \end{array} \right\}$$

$$opt[0][1] = max\left\{ \begin{array}{rl} 0, & path\_eliz = path\_eliz + \text{"\_"} \\ E[0], & path\_eliz = path\_eliz + \text{"E"} \end{array} \right\}$$

$$opt[0][2] = max\left\{ \begin{array}{rl} 0, & path\_charles = path\_charles + \text{"\_"} \\ C[0], & path\_charles = path\_charles + \text{"C"} \end{array} \right\}$$

Recursion:
Choose based on the max score and store the respective path in the structure

$$opt[i][0] = max\left\{ \begin{array}{rl} opt[i-1][0], & path\_sara = path\_sara + \text{"\_"} \\ opt[i-1][1] + J[i], & path\_sara = path\_eliz + \text{"J"} \\ opt[i-1][2] + J[i], & path\_sara = path\_charles + \text{"J"} \\ opt[i-1][1], & path\_sara = path\_eliz + \text{"\_"} \\ opt[i-1][2], & path\_sara = path\_charles + \text{"\_"} \end{array} \right\}$$

Similarly, we can write for other two members.

$$opt[i][1] = max\left\{ \begin{array}{rl} opt[i-1][1], & path\_eliz = path\_eliz + \text{"\_"} \\ opt[i-1][2] + E[i], & path\_eliz = path\_charles + \text{"E"} \\ opt[i-1][0] + E[i], & path\_eliz = path\_sara + \text{"E"} \\ opt[i-1][2], & path\_eliz = path\_charles + \text{"\_"} \\ opt[i-1][0], & path\_eliz = path\_sara + \text{"\_"} \end{array} \right\}$$

$$opt[i][2] = max \begin{Bmatrix} opt[i-1][2], & path\_charles = path\_charles + "\_" \\ opt[i-1][0] + C[i], & path\_charles = path\_sara + "C" \\ opt[i-1][1] + C[i], & path\_charles = path\_eliz + "C" \\ opt[i-1][0], & path\_charles = path\_sara + "\_" \\ opt[i-1][1], & path\_charles = path\_eliz + "\_" \end{Bmatrix}$$

Final ans to be returned is: max(opt[n-1][0], opt[n-1][1], opt[n-1][2]) along with the path stored alongside that value in the matrix
The maximum of these three values will be the maximum points which the team can earn after n rounds.

Analysing TC:
We traverse through the given three arrays simultaneously once. So, the TC = O(n)
Analysing SC:
We store 3n values in the matrix in the algorithm. So, the space complexity = O(n)

Proof of correctness:
We can use induction to prove the correctness of the algorithm.
For base case (i=0), it is trivial to see that max(opt[0][0], opt[0][1], opt[0][2]) gives us the optimal value.
Now, let say that this is true for all values till i-1. We need to show that it holds true for the i^{th} round too.

opt[i][0] denotes the max value the team can achieve if Sara plays in the i^{th} round. For the recursion mentioned above, we see that it considers these five possibilities:

1. Sara also played in the $(i-1)^{th}$ round, so she can't play this round
2. max score till i^{th} round was achieved when Elizabeth played last round and now we add Sara's score to the current max score if it's positive
3. max score till i^{th} round was achieved when Charles played last round and now we add Sara's score to the current max score if it's positive
4. max score till i^{th} round was achieved when Elizabeth played last round and now we do not add Sara's score to the current max score if it's negative
5. max score till i^{th} round was achieved when Charles played last round and now we add Sara's score to the current max score if it's negative

In this way, all the possibilities are taken care by the given algorithm which could have maximised the score if Sara played in the last round.
Similarly, we can calculate the values for Elizabeth and Charles.
Now taking the max(opt[i-1][0], opt[i-1][1], opt[i-1][2]) gives us the max score which can be scored by the team.
Hence, proved.

Problem 2

Collaborators: List students that you have discussed problem 2 with: Vishnu Teja Narapareddy, Tulika Sureka
We can use the max-flow algorithm to solve this problem.

Construction:
First of all, we convert the given undirected graph G to a directed graph by adding another edge between same set of vertices and then giving them the respective direction, keeping their weights as it is, if any, given in the original graph. Let's call this transformed directed graph as H
Now, we create a source node, s', and connect it to the vertex v with the edge weight as 2. Also, we connect the vertices s and t with a sink node, t', with an edge weight of 1.
Afterwards, we split each vertex in graph H into two vertices, let say v_{in} and v_{out}, and connect these split vertices by an edge weight of 1. The other edges in graph H would be transformed as follows: edge (a,b) in graph H would correspond to the edge (a_{out}, b_{in}) in the new graph. Let say this graph is J. In the graph J, paths are edge disjoint if and only if they are vertex disjoint because of the splitting of vertices done by us.

Algorithm:
Now, we run the ford-fulkerson algorithm to get the max flow in the graph J. If the max flow algorithm gives the value as 2, then, we can say that there exists a path from s to t that passes through v in which none of the vertices are repeated.

Analysing TC:
In the construction of the graph, we go through the vertices and edges, so the time taken is O(V+E). The ford fulkerson algorithm takes the time O(Ef), where f is the max flow possible in the graph, which is 2 in our case. Hence, the total time complexity is O(V+Ef).

Proof of Correctness:
We can show that a set of edge disjoint paths in the graph J are also vertex disjoint. This is because after we split each vertex into v_{in} and v_{out} and joined them with an edge weight of 1, we are trying to force the flow to pass through that vertex only once. This is because if one of the augmenting paths chooses that vertex and flow passes through that, then, for other augmenting paths to be edge disjoint, flow can't reach to that same vertex's v_{in} as it would then lead to common edge (edge between v_{in} and v_{out}) for two flows in the graph which is not possible in the edge disjoint problem. Thus, it shows that set of edge disjoint paths in graph J are also vertex-disjoint.
Every path in the graph J can be represented as s' $\rightarrow v_{in} \rightarrow v_{out} \rightarrow \ldots\ldots. \rightarrow m_{in} \rightarrow m_{out} \rightarrow t$. It corresponds to the following path in the original graph s $\rightarrow$ v $\rightarrow \ldots\ldots.\rightarrow$ m $\rightarrow t$.
This shows that set of vertex disjoint paths in the original graph corresponds to a set of vertex disjoint paths in the graph J.
Hence, we say that set of vertex disjoint paths in the graph J corresponds to a set of vertex disjoint paths in original graph.

Problem 3

Collaborators: List students that you have discussed problem 3 with: Vishnu Teja Naraparedy, Tulika Sureka

(a) Modelling max flow as linear program by defining variables for each s-t path:
Let us denote the set of all possible simple paths from s to t by P. Now, consider the variable x_p, which denotes the flow which is going along the path p in the total s-t flow.
Our objective functions becomes this, which we need to maximise:

$$\sum_{p \in P} x_p$$

Following are the constraints for the given objective function:

$$\sum_{p \in P:(u,v) \in p} x_p \leq C(u,v) \qquad \forall (u,v) \in E$$

$$x_p \geq 0 \qquad \forall p \in P$$

The first constraint condition denotes that sum of all flows (considering all the paths) along an edge cannot be greater than the capacity of that edge.
The second constraint condition denotes that flow along any path p is non-negative.
There can be exponentially many possible paths from s to t and thus the number of variables and constraints which we have are exponential too.

(b) Dual of the above formulated LP looks like this:
We have one variable $x_{u,v}$ for every edge (u,v) ∈ E. It basically tell us whether that edge is in cut or not. C(u,v) denotes the capacity of the edge (u,v)
Our objective function would be this, which needs to be minimised:

$$\sum_{(u,v) \in E} C(u,v) x_{u,v}$$

Following are the constraints for the given objective function:

$$\sum_{(u,v) \in p} x_{u,v} \geq 1 \qquad \forall p \in P$$

$$x_{u,v} \geq 0 \qquad \forall (u,v) \in E$$

The first constraint condition denotes that the separation between s and t along any path is alteast greater than or equal to 1. This basically stands for the number of cuts possible in that path. It means that for each path atleast one edge must be in the cut
The second constraint condition denotes the possibility that whether that edge is present in the cut or not.
We can interpret this as LP relaxation of the min-cut problem.

Problem 4

Collaborators: List students that you have discussed problem 4 with:None
Yes

Problem 5

Collaborators: List students that you have discussed problem 5 with: Vishnu Teja Narapareddy, Tulika Sureka

We can use the dynamic programming approach to solve this problem.
Our objective is to repair the damaged parts of the road using minimal area being covered in the patches used for repairing, given the constraint that we can use atmost P patches, rectangular in shape.
Let say the road is given to us in the form of 2-dimensional matrix, road[2][n], where damaged parts are represented by 0 and the good parts are represented by 1.

SubProblem:
opt[i][j] has a structure which denotes the minimum area covered using 'i' patches for a road of length upto 'j' units, along with the dimensions of the last patch. The constraints for i are $i \in [0,P]$ and j are $j \in [0,n]$.

Recursion:
At every step, we have two options when we try to increase the length of road by 1 unit:

1. We use the last patch and try to extend it if the new column has a damaged portion
2. We start a new patch to cover the damaged portion in the new column

For this approach to work, we need to store the coordinates of our patch so that we know about the orientation of the last patch. It will help us to see how much area needs to be covered while extending the length of the road covered by 1 unit. The structure used to store the patches is as follows:
Let m be left column from where the patch starts and n be the ending column for the patch. We also need to take into account the orientation of the patch. So, we store another integer to take care of that. Let say, field 'o' denotes the orientation of the patch. If the patch covers just the first row, we store the value 0 in the orientation field. If it covers just the second row, we store the value 1 in the orientation field. If it covers both the rows, we store the value 2 in the orientation field.
This orientation of the last patch, along with the orientation of the new column being added in the road is required to get the overall dimension and orientation of the updated patch afterwards.

So, we have the following cases:

1. if road[0][j-1]==1 and road[1][j-1]==1 // new column is good
   opt[i][j] = opt[i][j-1] // no need to extend the patch or start a new patch, just use the value for the last column
2. if road[0][j-1]==0 and road[1][j-1]==1 // upper element in new column is damaged
   This means the orientation of the new column is 0 as only the upper element is damaged in the column being added
   (a) extending the last patch
      i. orientation of last patch is 0:
         last_patch = opt[i][j-1].patch
         opt[i][j].area = opt[i-1][last_patch.m - 1].area + (j-last_patch.m)
         opt[i][j].patch = last_patch.m,j,0
      ii. orientation of last patch is not 0:
         last_patch = opt[i][j-1].patch
         opt[i][j].area = opt[i-1][last_patch.m - 1].area + 2*(j-last_patch.m)
         opt[i][j].patch = last_patch.m,j,0

   Take the min area value for the two cases discussed above

(b) starting a new patch
opt[i][j].area = opt[i-1][j-1].area + 1 // no need to take into account the orientation of last patch
opt[i][j].patch = (j-1,j,0) // start col, end col, orientation

Take the min area value for the two cases discussed above (extending the last patch, starting a new patch)

3. if road[0][j-1]==1 and road[1][j-1]==0 // lower element in new column is good
This means the orientation of the new column is 1 as only the lower element is damaged in the column being added

    (a) extending the last patch

        i. orientation of last patch is 1:
        last_patch = opt[i][j-1].patch
        opt[i][j].area = opt[i-1][last_patch.m - 1].area + (j-last_patch.m)
        opt[i][j].patch = last_patch.m,j,1

        ii. orientation of last patch is not 1:
        last_patch = opt[i][j-1].patch
        opt[i][j].area = opt[i-1][last_patch.m - 1].area + 2*(j-last_patch.m)
        opt[i][j].patch = last_patch.m,j,1

    Take the min area value for the two cases discussed above

    (b) starting a new patch
    opt[i][j].area = opt[i-1][j-1].area + 1 // no need to take into account the orientation of last patch
    opt[i][j].patch = (j-1,j,1) // start col, end col, orientation

Take the min area value for the two cases discussed above (extending the last patch, starting a new patch)

4. if road[0][j-1]==0 and road[1][j-1]==0 // new column is fully damaged
This means the orientation of the new column is 2 as both the elements are damaged in the column being added

    (a) extending the last patch

        i. orientation of last patch does not matter in this scenario:
        last_patch = opt[i][j-1].patch
        opt[i][j].area = opt[i-1][last_patch.m - 1].area + 2*(j-last_patch.m)
        opt[i][j].patch = last_patch.m,j,2

    (b) starting a new patch
    opt[i][j].area = opt[i-1][j-1].area + 2 // no need to take into account the orientation of last patch
    opt[i][j].patch = (j-1,j,2) // start col, end col, orientation

Take the min area value for the two cases discussed above (extending the last patch, starting a new patch)

Base Case:
For j $\in$ [0,n]:
opt[0][j].area = 0 // area that can be covered is zero when there is no patch
opt[0][j].patch = null, null, null
for i $\in$ [0,P]:
opt[i][0].area = 0 // area to be covered is zero when there is no road

opt[i][0].patch = null, null, null

For the recursive cases, we make a check that if the last patch was null, we can't extend the patch and have to go for the case - starting a new patch.

The result to our problem is opt[P][N] as it would give us the min area which would be required by P patches to cover the damaged portion of road of length N.

Analysing TC:
We are traversing through the complete matrix of size (P+1)(N+1). So, the time complexity for the overall problem becomes O(NP).
Analysing SC:
We require to

Proof of Correctness:
We can use strong induction to prove the correctness of the algorithm.

For base cases (N=0 or P=0), it is trivial to observe that we would not be covering any road area through patches, so their value is 0.

Now, let say our algorithm gives the optimum area value for all the values till opt[i][j]. We need to show that the algorithm provides the optimal value for opt[i][j] too.

opt[i][j] denotes the min area to be covered using i patches up til j column. So we can see that for the area to be minimised, we have two choices, either to extend the last patch used or to start a new patch from this column.

1. we extend the patch
   We need to find the area required to cover till the start of the last patch using i-1 patches and then add the new area after extension of the patch as described for each of the possible orientations in the algorithm.
   We already know that all the previous computed values are optimal. So, we get the corresponding value from the matrix and add the new computed area.
2. we start a new patch
   We know the area required to cover the road uptil j-1 column using i-1 patches (already computed before). We also know that it is optimal (using strong induction).
   We add the new patch area corresponding to the orientation of the column being added to this value.

Finally, we take the min of the two values from the possible cases discussed above. As those values were computed using the optimal values till the previous point, we can say that the area value which we get for this block in the matrix is also optimal. This ensures that opt[i][j] also stores the optimal value. The result would be stored in the cell opt[P][N], which will give us the minimum area to be covered using P patches for a 2-lane road of length N units. Hence, proved.