CS 580 Algorithm Design, Analysis, And Implementation
Fall 2021 Vassilis Zikas HW 3

Problem 1

Collaborators: List students that you have discussed problem 1 with: Vishnu Teja Narapareddy, Tulika Sureka

(a) The given graph G has c connected components and positive edge weights. Finding a subgraph of G such that it has n-c edges and the vertices which were connected earlier remain connected is equivalent to finding the spanning tree in each of the c components. This is because a spanning tree has v-1 edges and maintains the connectedness same as that of the graph and doesn't have any cycles. So, if we find the spanning tree for each the c components, we will get a spanning forest H, subgraph of graph G, with n-c edges and same connectedness as that in G.

So, for this problem, we have to find that spanning forest which has maximum sum of weights of the edges. This problem can be solved if we modify the given graph G. Basically, we iterate through all the edges and negate them, so now finding the minimum spanning tree for each of the c connected components and taking their sum would be same as that of finding the maximum weighted subgraph with the given constraints.
We can use Kruskal algorithm for finding the minimum spanning forest for the graph. Thus, we would get c spanning trees which is called a spanning forest, H, subgraph of G satisfying all the conditions given in the problem. Now, taking the sum of all these spanning trees and negating it again would give us the maximum weighted subgraph of graph G.

Proof of correctness:
Let say each component in the graph G has some vertices V_i. So if we find a spanning tree for that component, we will cover all the vertices for that component and get V_i -1 edges. So, after repeating this process for every component, we cover all the vertices and they remain connected with each other and the total number of edges would be n-c (upon summing V_i -1 edges for c components). We would get c spanning trees which is called a spanning forest.

Analysing TC:
Time complexity for the algorithm is same as that of the Kruskal algorithm. Overall TC is O(ElogV) where E is the number of the edges and V is the number of vertices in the graph G.
Analysing SC:
Space complexity of the algorithm is O(E+V)

(b) For this problem, we divide the task into 2 subtasks:
1. Find the longest sequence of 1's in the rows
2. Find the longest sequence of 1's in the column
For the subtask 1:
Let's divide the matrix into 4 subparts recursively till we reach individual elements. The base case would be individual elements in this case. Now while merging the elements, we store the longest sequence of 1's that can be found in that submatrix in the leftmost column if we traverse in the left direction and in the rightmost column of the submatrix if we traverse in the right direction. So basically, this would help us to merge the

subparts in O(n) time. Let say we get the subparts 1,2,3 and 4 on dividing the original matrix. We keep on updating the variable longest-sequence-row in every merge step to keep track of the longest sequence of 1's found till that point. There might be a possibility that when we merge subparts 1 and subparts 2, the middle portion gives us the longest sequence. So we keep on checking this value too in the merge step and update the variable longest-sequence-row if needed.
For updating the rightmost column while merging subparts 1 and 2, we compare the valuees in the rightmost column of subpart 1 and leftmost column of subpart 2:
1. If the subpart 2 has the value as n/2 (meaning all ones in that row), we update the rightmost value for that row in merged part with value of n/2 + value in rightmost column of subpart 1.
2. If the subpart 2 has the value less than n/2, we update the rightmost value for that row in merged part with value in leftmost column of subpart 2 + value in rightmost column of subpart 1. Compare this value with variable longest-sequence-row and update if required.
Similarly, we can update the values for leftmost column while merging subparts 1 and 2.

This would give the longest sequence of 1's in the n/2 * n matrix. Similarly, we combine subparts 3 and 4. This would also give the longest sequence of 1's in the n/2 * n matrix. As in this subtask, we were just looking for the longest sequence in terms of rows, so we don't do the merge step for subparts 1 and 3, subparts 2 and 4. In this way, we would have calculated the longest sequence of 1's in only in the rows.
Similarly, we can calculate it for columns by taking the transpose and applying the same procedure. Then, the max of these two variables, longest-sequence-row and longest-sequence-col, would be the answer to our problem.

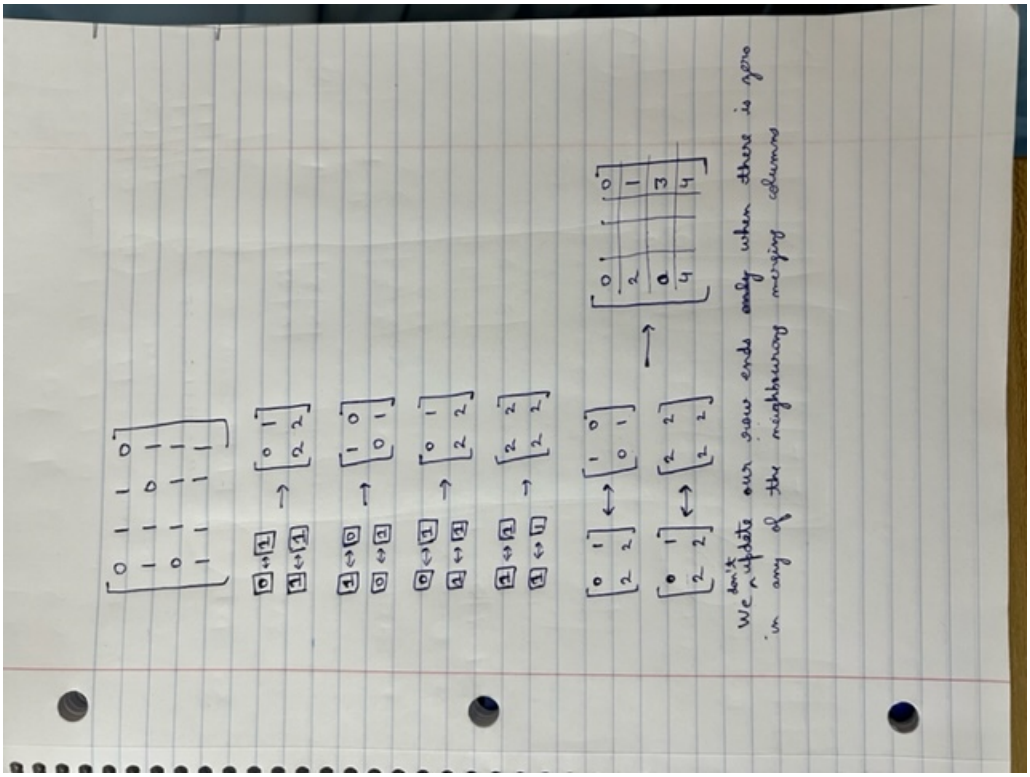

Figure 1: Simulation of Algorithm

Analysing the TC:

Time complexity would be : T(n^2) = 4T(n^2/4) + Θ(n)
Substituting n^2 by k, we get:
T(k) = 4T(k/4) + Θ($\sqrt{k}$)
From master theorem, we get TC = O(k),
which is equivalent to O(n^2)

Problem 2

Collaborators: List students that you have discussed problem 2 with: Vishnu Teja Narapareddy, Tulika Sureka

(a) The optimal algorithm would be the one in which Mario has to travel along the least edges in travelling from V_s to V_t so that he loses minimum number of lives. This process would ensure that he has maximal number of lives at the end.
As all the edges have negative weights of 1, we first iterate through all the edges and convert their weight from -1 to +1. After this, we apply BFS on graph G starting from V_s and this algorithm would give us the minimum path length to V_t. For calculating the lives left with Mario, we subtract this path length from the initial lives and thus obtain the lives still remaining with Mario.

Analysing TC:
First, we iterate through all the edges, so it takes O(E) time and then BFS algorithm takes O(V+E) time. So, the time complexity of the problem is O(V+E).
Analysing SC:
Space complexity of the given problem is same as that of BFS algorithm because we don't require any extra space storage. So, space complexity is O(V+E).

(b) The optimal algorithm would be the one in which Mario has to travel along the least weighted edges in travelling from V_s to V_t so that he loses minimum number of lives. This process would ensure that he has maximal number of lives at the end.
As all the edges have different negative weights, we first iterate through all the edges and convert their weight from negative to positive value. After this, we apply Dijkstra algorithm on graph G starting from V_s and this algorithm would give us the minimum path length to V_t. For calculating the lives left with Mario, we subtract this path length from the initial lives and thus obtain the lives still remaining with Mario.

Analysing TC:
First, we iterate through all the edges, so it takes O(E) time and then Dijkstra algorithm takes O((V+E)logV) time. So, the time complexity of the problem is O((V+E)logV).
Analysing SC:
Space complexity of the given problem is same as that of Dijkstra algorithm because we don't require any extra space storage. So, space complexity is O(V+E).

(c) For this problem, I reduced it to somewhat similar to part (b).
For handling the lives which Mario can get from each node, I modified the given graph and for all the vertices, I added that vertex value to all the outgoing edges from that vertex. In this way, I ensure that Mario uses the lives after reaching that node only as in the original case, he would have got the lives upon reaching that vertex

and then could have gone through any of the edges leaving from that vertex.
Now the difference in this part and part (b) is that some edges might become positive as there might a node which has high positive value and whose outgoing edge has low weight, so after modification, the edge would have a positive weight. So, we won't be able to use Dijkstra algorithm as upon negating the edge weights, we would have a mix of positive and negative weights. This suggests that we should use Bellman Ford algorithm

Algorithm:
First, we iterate through all the edges and multiply their weights by -1 so that afterwards, we get the cost of travelling along the shortest path which can be subtracted easily from the lives given at initial node V_s. This graph contains both positive and negative weighted edges (explained earlier). So, now we apply Bellman Ford algorithm to get the path lengths from vertex s to all the other vertices. The output matrix would tell the number of lives spent in moving from s to that vertex. There is minor modification which needs to be done in the updation part of the algorithm. While updating the distance of vertex v considering vertex u and edge e_{uv}, we also need to ensure that this sum doesn't become positive at vertex v because otherwise we won't be able to traverse that path as all our lives would be lost in that path before reaching to the vertex v.

Input: Graph and a source vertex *src*
Output: Shortest distance to all vertices from *src*. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.
1) This step initializes distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.
2) This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.
.....**a)** Do following for each edge u-v
.....If dist[v] > dist[u] + weight of edge uv, then update dist[v]
.....dist[v] = dist[u] + weight of edge uv
3) This step reports if there is a negative weight cycle in graph. Do following for each edge u-v
.....If dist[v] > dist[u] + weight of edge uv, then "Graph contains negative weight cycle"
The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn't contain a negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

Figure 2: Bellman Ford Algo

So the update condition for the algorithm becomes:
if dist[v] > dist[u] + e(u,v) and dist[u] + e(u,v) ≤ 0 :
' dist[v] = dist[u] + e(u,v)

Analysing TC:
Time complexity for Bellman Ford is O(VE) and time spent on updating the values of all the edges and then negating them is O(E). So the total time complexity of the algorithm is O(VE)
Analysing SC:
The space complexity of the algorithm is same as the space complexity of the Bellman Ford algorithm i.e. O(V)

(d) For this scenario, where Mario has single skip superpower, we will proceed in this way:
First do the transformation of the given graph as described in part (c) of this problem, i.e add vertex value to all it's outgoing edges and multiply all the edges by -1 so that now the edge weight would give us the number of lives spent in moving along that edge.
Now, some of the edges can be positive and some can be negative as explained in the previous part (part c). We break the given problem into two subtasks:
1. Find the vertex from s which will require least number of lives to be spent in reaching to that vertex
2. Find the vertex which will require least number of lives to be spent to reach to vertex t
For solving these two subtasks, we run the Floyd-Warshall algorithm and get the output in the matrix form which will give the minimum number of lives spent in moving from vertex a to vertex b for all pairs of a and

b.

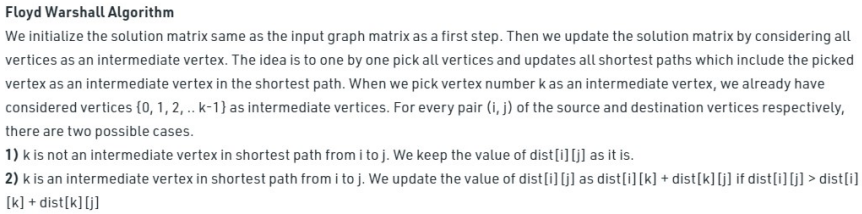

Figure 3: Floyd Warshall Algo

The update condition for this algorithm is also modified:
if dist[i][k] + dist[k][j] < dist[i][j] and dist[i][k] + dist[k][j] <= 0:
’ dist[i][j] = dist[i][k] + dist[k][j]

Now, for finding the answer to subtask 1, we go to row of vertex s and find the least value in that row so as to find the vertex which will require the minimum number of lives to be spent to reach to it from vertex s. Let say this vertex is x.
Similarly, for subtask 2, we go to the column of vertex t and then find the minimum value in that column so as to get the vertex which will require minimum number of lives to be spent to reach to vertex t. Let say this vertex is y.
So, in order to maximize the number of lives at the end, Mario should take the following path :
1. Move from vertex s to vertex x
2. Jump from vertex x to vertex y
3. Move from vertex y to vertex t
This would ensure that the lives are maximum at the end because we have reduced the number of lives which can be spent in between the two vertices s and t, while incorporating his superpower.

Proof:
Let say there is another optimal solution other than the path taken by the given algorithm (s-x-y-t). Now, there are two cases:
1. Optimal algorithm chose vertex x’:
For this to happen, this means that there was some other vertex which required least amount of lives to be spent to reach from s. However, this is not possible as the given algorithm would have chosen that node in the first place rather than choosing vertex x.
2. Optimal algorithm chose vertex y’:
This means that there was some other vertex from which Mario could have reached to vertex t minimising the number of lives spent in that path. However, this is not possible as our algorithm would have chosen that instead of choosing vertex y initially.
3. Optimal algorithm choses vertex x’ and y’:
This means that there were vertices x’ and y’ which have led to more number of lives with Mario but this is not possible as the given algorithm choses maximal lives it can get from vertex s to vertex x. Similarly, the case goes for maximising the lives of vertex y to vertex t. In this case, it means that lives spent from s to x’ were minimum which is not possible as then our algorithm would have chosen x’ then. Similarly, the argument goes for y’ to t. Hence, this is not possible.
So, this shows that this algorithm provides the optimal number of lives which Mario can have while going from vertex s to vertex t with a single skip superpower.

Analysing TC:
Time complexity for Floyd Warshall is O(V^3) and time spent on updating the values of all the edges and then negating them is O(E). So the total time complexity of the algorithm is O(V^3)
Analysing SC:
The space complexity of the algorithm is same as the space complexity of the Floyd Warshall algorithm i.e. O(V^2)

Problem 3

Collaborators: List students that you have discussed problem 3 with: Vishnu Teja Narapareddy, Tulika Sureka
For this problem, we can use the max flow algorithm in the directed graph.
Construction:
Let X be the vertex set consisting of 2n balls i.e it has 2n nodes
Let Y be the vertex set consisting of buckets i.e it has m nodes
Now, to reduce this bipartite graph problem to max flow problem:
First, we create a source node s and connect it to all the vertices in set X with an edge weight of 1 as the ball has to go fully in one of the buckets.
Then, we create a sink node t and connect all vertices in the set Y to t with weight of the edges equal to the capacities of the bucket respectively.
Now, for connecting the edges between the vertices of set X and set Y:
We know that each ball has a list of buckets associated with it in which it can go. For every pair of balls, we first calculate the intersection of the two lists. Then, connect node in the set X to nodes in the set Y which are in the intersection list through the use of intermediate nodes. Basically, connect the balls which belong to the same pair to an intermediate node for the common bucket and then connect this intermediate node to the bucket. Give these edges the weight of 1 so as to ensure that no two balls belonging to the same pair go in the same bucket.
Afterwards, connect each ball to remaining list of buckets directly in which it can go with an edge weight of 1. Here is a pictorial representation of the construction discussed above:
The maximum number of balls that can be assigned to the buckets is equal to finding the max flow in the constructed graph.
While constructing the graph, I have taken care of all the constraints given in the question. Bucket B_i can take atmost b_i balls (edge capacities while connecting bucket to sink node). The balls forming the pair cannot be assigned to the same bucket (through the use of intermediate nodes). Each ball goes into atmost one bucket (outgoing edge from each ball has weight 1).
So, this ensures that every integral flow in this graph would give a valid assignment of balls to buckets. Presence of integral flow ensures that either every edge would be at its full capacity or it would be untouched Every valid assignment of balls to buckets can also be represented by some flow in the graph.
Algorithm:
We have reduced the given problem to finding the max flow in the constructed graph (discussed above). Now, we can simply run the Ford-Fulkerson algorithm and the maximum flow value would be equal to the maximum number of balls assigned to the bucket.

Analysing TC:
Time complexity for the Ford-Fulkerson algorithm is given by O(E*f) where E is number of edges in the graph and f is max flow for the graph. So, in our graph construction, the total number of edges could be 2n + nm + m and the max flow can be 2n. So, the overall TC would be O(n^2m) where n is the number of pairs of ball and m is the number

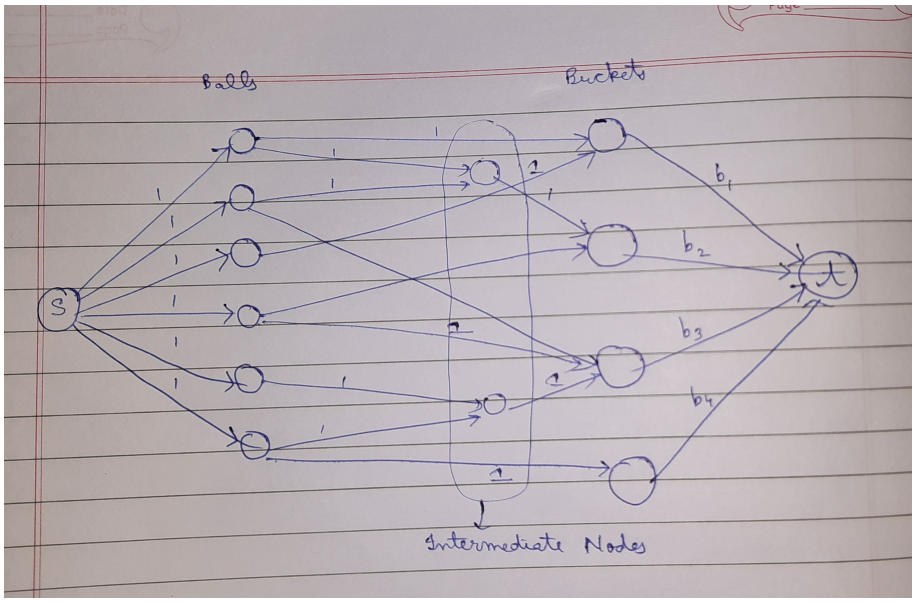

Figure 4: Graph Construction

of buckets.
Analysing SC:
We require to store an additional nm +2 nodes in our constructed graph. So the space comlexity of our algorithm: O(nm)

Problem 4

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