Homework 5 - training problems

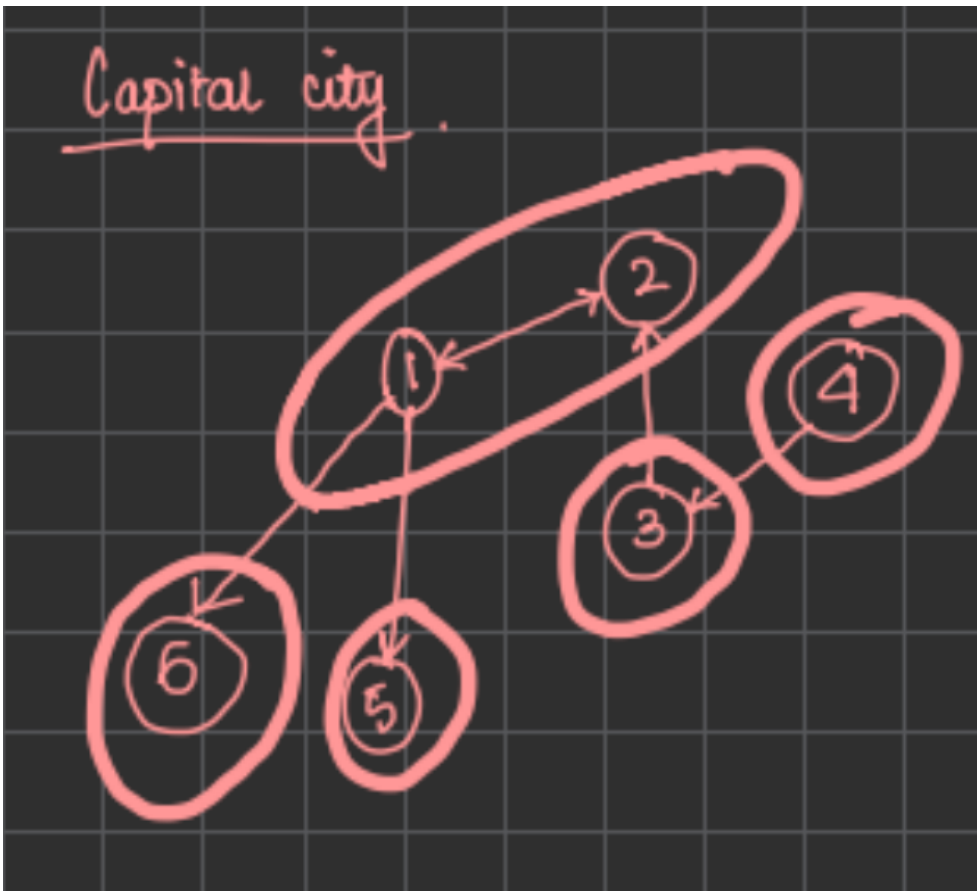
Pratyay Dutta (Codeforces ID : pdutt005)

March 6, 2024

Contents

1 Capital City - Submission ID : 249669120

Explanation: This problem is a graph problem where we have to find the nodes which are connected to all the other nodes in the graph. A naive approach would be to do a DFS on all nodes and see if all nodes are present in the DFS traversal of each node. The ones in which all nodes are present, can be considered as the capital. However, that will be a very time consuming algorithm.
Instead, we simplify the problem using Strongly Connected Components (SCCs). We assert that after making the strongly connected components graph from the original graph, if there exists a single SCC from which there are no outgoing edges, then all possible capitals will be in that SSC. To explain this intuition, we explain the following. The following example is used to demonstrate.



- There must be exactly one such SCC with no outgoing edges. If there were more than one SCC with zero outgoing edges, it would mean there are multiple disjoint sets of nodes, each of which cannot reach the other. This scenario contradicts the requirement that the capital can reach all nodes in the graph. Consider the given example where the edges are : [(1,2),(2,1),(3,2),(4,3),(1,6),(1,5)]. The SCCs are:[(1,2),3,4,5,6] where 5 and 6 both have zero outgoing edges. This means that 5 and 6 are disjoint and there exists no node in the graph which is reachable from all other nodes.
- The SCC with zero outgoing edges is the only SCC from which all other nodes are reachable, directly or indirectly. If an SCC has outgoing edges, it means there are nodes/SCCs it cannot reach independently without traversing through other SCCs. In the given example, if we remove the edge (1,6), we have the SSCs : [(1,2),3,4,5]. Here, (1,2) has an outgoing edge to 5. This means that 1 or 2 cannot be accessed from 5 therefore they cant be the capital.

Therefore, we have to create the SCCs and we have to find if there exists only one SSC with zero outgoing edges. If yes, then the nodes inside that SSC in increasing order, is our answer. We implement Kosaraju's algorithm to find the SCCs of the graph. We:

- We first do a DFS on each node and add nodes to a stack *path* having the nodes in the order they were completed.
- We reverse the graph and do DFS on the nodes which are at the top of the stack. Since the DFS returns the path, the SSCs are the paths from each DFS traversal.We do this till we have visited all nodes.
- In each SCC, all nodes are referred by the minimum of the nodes in that SCC.
- A dictionary *sccs* contains the SCC ids of each SCC. (the minimum value in each SCC)
- A dictionary dag_out keeps track of the outgoing edges from each SCC.
- We check if the count of SCCs where the number of outgoing edges is 0 is exactly equal to 1 i.e only one SCC with no outgoing edge.(Sink SCC). If yes, then we output the contents of the SCC having 0 outgoing edges. If no, then there can be no node in the given graph connected to all other nodes.

Time complexity : The time complexity of the algorithm can be analyzed as follows:

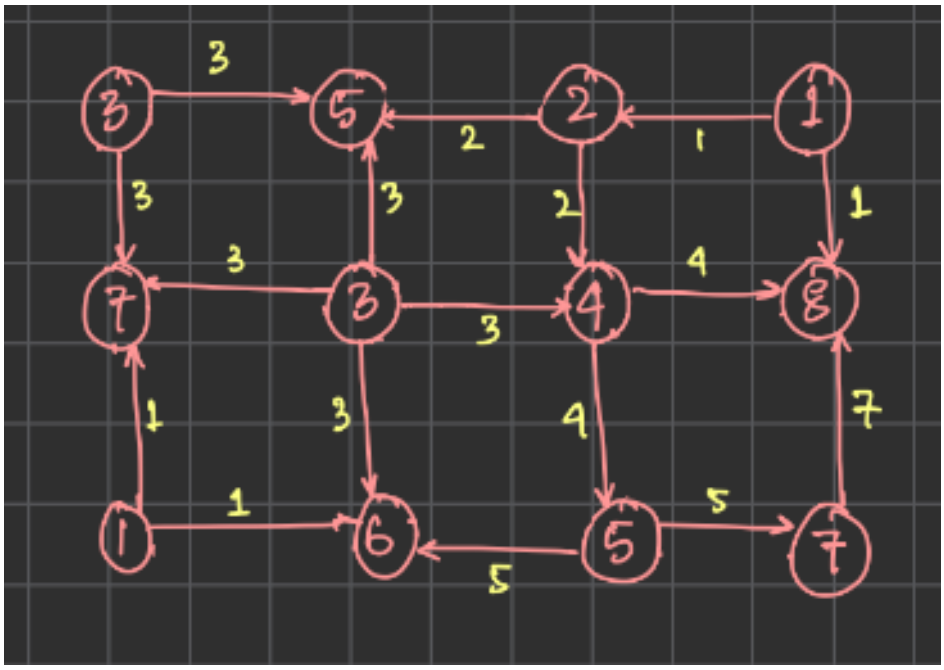
1. Graph Construction: $O(m)$. The graph and the reversed graph are constructed by iterating over all edges.
2. Depth-First Search (DFS): $O(n + m)$ for all DFS operations combined.
3. Finding Strongly Connected Components (SCCs): $O(n+m)$. This involves performing DFS from each vertex not yet visited in the reversed graph order determined by the first DFS. Although it seems like this could be $O(n^2+m)$in the worst case, each vertex and edge are visited at most once across all DFS calls. Thus, the total time for finding SCCs is O(n+m).
4. Constructing the Directed Acyclic Graph (DAG): $O(m)$. The DAG is constructed by examining the edges between different SCCs. Since each original edge is considered exactly once to determine if it connects two different SCCs, and since the mapping of nodes to their SCCs can be done in constant time, this step is O(m).
5. Identifying the Capital: $O(n + m)$. Identifying the capital involves checking the out-degrees of each SCC in the DAG, which is O(n) since it iterates through the SCCs and counts the number of outgoing edges. However, since the number of SCCs is less than or equal to n, and since each edge in the original graph contributes to at most one connection in the DAG, this step remains $O(n + m)$.

Given these components, the total time complexity of the algorithm is the sum of the complexities, leading to:

$$\text{Total Time Complexity} = O(n + m)$$

2 Making New Friends - Submission ID : 249854234

Explanation : The problem was about finding a path which connects all vertices in such a way that the total of the edge weights in that path has the minimum value. As apparent in the given problem, the vertices are the students. The students are seated in specific positions which have access to only its adjacent locations. The edges are the connections of each with its neighbour.
First we consider a graph with bidirectional edges with different edge weights in each direction. The weight from one vertex u to another v will be their shyness index $shyness(u)$ because that is the minimum number of candies required to convince that student to converse with its adjacent student.
But, we see that while we are looking for a path to minimize edge weights, we can consider only one edge between a pair of nodes which is the one with the lesser edge weight. As it can be seen from the graph illustration of the given example test case:



We can consider the edges to be undirected since having an edge ensures information flow. Node 3 is connected to node 5 and 7 each by edge weights 3. This means the number of candies given to node 3 will be 6 so that information given to 3 passes to 5 and 7. Similarly, this weight distribution paradigm applies to all other vertices as well.
So our problem has now reduced to finding a Minimum Spanning Tree on this graph where:

- Vertices are students as in the matrix
- Weight between each pair of vertices will be the edge with minimum weight. We can consider this edge undirected since the connectivity matters.

Our function $addedges(x, y)$ takes adjacent edges and adds them to a priority queue which we use to keep track of the edge with the minimum weight connected to each vertex. Our priority queue is a heap and the priority is the weight of the edge.
To find the MST of the graph, we:

- A set named *visited* is initialized with the tuple (0, 0), indicating that the top-left corner cell of the matrix is the starting point of the MST and is considered visited.

- Our priority queue *edges* is initialised to store potential edges to be added to the MST. The edges are stored as tuples (weight, (nx, ny)), where weight is the edge's weight, and (nx, ny) are the coordinates of the cell the edge leads to. The $addedges(x, y)$ functions adds the graphs edges on the fly into the priority queue.
- Constructing the MST:
  - The *addedges* function is called with the starting cell (0, 0) to add all potential edges from the starting cell to the priority queue.
  - The code enters a while loop that continues as long as there are edges in the priority queue and not all cells have been visited. The loop does the following:
    * Selecting the Minimum Edge: It pops the edge with the smallest weight from the priority queue using heapq.heappop(edges).
    * Visiting New Cells: If the cell (x, y) at the end of the popped edge has not already been visited, it is marked as visited. The edge's weight is added to the total weight of the MST (mst_weight), and the add_edges function is called with (x, y) to add all potential new edges from this newly visited cell.
- After the construction of the MST is complete, we take the sum of the weights of the MST which is the minimum number of candies Yihan has to distribute to certain people in the class to make sure information is propagated to every student in the class by telling only one person. This is our answer.

The code passed all test cases except one where I'm hitting Memory Limit Exceeded which might be because of the huge priority queues for very large and dense graphs.
Time Complexity : Given a matrix with dimensions $n \times m$, the implementation of Prim's algorithm for finding the Minimum Spanning Tree (MST) involves iterating over all cells of the matrix and considering up to four adjacent cells for each. The primary operations for maintaining the set of edges to be considered for inclusion in the MST are performed using a priority queue, with the key operations being insertion and extraction, both of which have a complexity of $O(\log(N))$, where N is the number of items in the priority queue.

The algorithm starts with a single cell, adding potential edges to the priority queue. As the MST grows, edges to adjacent, unvisited cells are considered. For each cell, up to four adjacent edges might be added to the queue, leading to a worst-case scenario where nearly all cells are in the queue at some point during the execution of the algorithm.

Given this, the time complexity of the algorithm can be analyzed as follows:

- Each cell might contribute up to four edges to the priority queue. However, this constant factor is disregarded in Big O notation.
- Each operation on the priority queue (insertion or extraction) has a complexity of $O(\log(n \cdot m))$ since, in the worst case, the queue may contain an edge for every cell.
- Since each of the $n \cdot m$ cells may at some point have an edge in the queue, and each such operation has a complexity of $O(\log(n \cdot m))$, the overall complexity of the algorithm is $O(n \cdot m \cdot \log(n \cdot m))$.

Therefore, the time complexity of the given implementation of Prim's algorithm for constructing a Minimum Spanning Tree from a matrix is $O(n \cdot m \cdot \log(n \cdot m))$.

3 Shelter from the rain - Submission ID : 249853716

Explanation : To find the minimum number of students getting wick, we have to maximise the number of students reaching shelters. If we consider the shelters to be a subgraph and the students to be another subgraph, then we have a biipartite graph. Getting the maximum students to shelters is finding the maximum matching on a bipartite graph. We will thus be implementing a bipartite matching algorithm on this bipartite graph of students and shelters.
To do this:

- **Graph Preparation** :
  - A 2D list *graph* is initialized to represent whether a student can reach a given shelter without getting sick. The list is initialized with False values.
  - For each student and shelter pair, it calculates the Euclidean distance between them. If the time it takes for the student to reach the shelter (calculated as distance divided by the student's running speed) does not exceed b, the corresponding entry in graph is set to True, indicating the student can reach that shelter in time.
- **Bipartite matching** :
  - The *bipartitematching* function is defined to find the maximum number of students that can be matched to different shelters without any student getting sick. It uses a depth-first search (DFS) approach to try and assign each student to a shelter. The dfs function, defined inside *bipartitematching*, attempts to find a shelter for student i. It iterates over all shelters; if a student can reach shelter j without getting sick (graph[i][j] == True) and shelter j is not yet visited in this iteration, it tries to match student i to shelter j.
  - If shelter j is already matched to another student (matching[j] != -1), it recursively tries to find another shelter for the previously matched student. If it succeeds, or if the shelter was not matched to anyone (matching[j] == -1), it updates the match for shelter j to student i and returns True. The matching list keeps track of the student assigned to each shelter, where matching[j] == i means student i is assigned to shelter j. If no student is assigned, matching[j] == -1.
  - For each student, it updates the visited list (used to mark shelters as visited in the current DFS iteration) and calls dfs. If dfs returns True, indicating a match was found for the current student, it increments *matchcount*.
  - The function returns the total number of successful matches (*matchcount*).
- **Result** : After computing the maximum number of matchings, the code calculates the number of students who cannot reach any shelter in time by subtracting the number of matched students from the total number of students ($n - matchedstudents$) and prints this number.

Time Complexity : The given code implements a bipartite matching algorithm to match students to shelters based on their ability to reach the shelter before getting sick, with n students and m shelters. The time complexity of this algorithm can be analyzed as follows:

- Graph Construction : The graph is constructed with a nested loop iterating over n students and m shelters, calculating the distance and checking if the student can reach the shelter in time. This step has a time complexity of $O(n \cdot m)$, as each of the n students is compared with each of the m shelters.
- DFS : The main part of the algorithm uses DFS to find matches. For each student i, DFS is called to explore potential matches in shelters:
  - DFS is called once for each student, leading to n calls in total.
  - Within each DFS call, the algorithm iterates over all m shelters to check for a potential match or to attempt to find an alternative match for an already matched shelter.

– In the worst case, the DFS may need to explore all paths from a student to all shelters, leading to a time complexity of $O(m)$ for each DFS call.

Hence, the DFS part has a worst-case time complexity of $O(n \cdot m)$, as it iterates over all shelters for each student.

- Overall : Combining the graph construction and the DFS-based matching, the overall time complexity of the algorithm is:

$$O(n \cdot m) + O(n \cdot m) = O(n \cdot m)$$