CS-218-HW5-Basic

Lakhan Kumar Sunilkumar

30 May 2025

Student ID: 862481700

1 A. Making New Friends: Codeforces Submission ID: 321984677

Explanation:

We are given a classroom of students arranged in a grid of size $r \times c$. Each student has a shyness level (an integer between 1 and 200), and they are only willing to talk to an adjacent student (up, down, left, or right) if they are given at least as many candies as their shyness level.

Once a student talks to another, they become friends and share all information with each other. The goal is to ensure that announcements can be propagated to all students (i.e., the students form a single connected group), while minimizing the total number of candies distributed.

Plan of Attack:

This can be modeled as a Minimum Spanning Tree (MST) problem on a grid graph. The idea is to represent each student as a node and each possible friendship (between adjacent students) as an undirected edge with weight equal to the minimum shyness between the two.

- Flatten the grid of students into a 1D array for easier indexing.
- For every student, create edges to their right and bottom neighbors to avoid duplication.
- The weight of each edge is $\min(\text{shyness}[u], \text{shyness}[v])$.
- Store each edge in a bucket corresponding to its weight (from 0 to 200).
- Apply Kruskal's MST algorithm:
  - Iterate through edge buckets from smallest to largest weight.
  - Use Disjoint Set Union (DSU) to connect components.
  - If the union of two students is successful (i.e., they were not already connected), add the edge's weight to the total cost.

Algorithm Steps:

1. Read the grid dimensions r and c, then flatten the shyness grid into a 1D list.
2. Create 201 buckets for all possible edge weights (0 to 200).
3. For each student, generate edges to their right and bottom neighbors and insert them into the corresponding bucket based on edge weight.
4. Initialize the DSU structure with all students in separate components.

5. Iterate over edge weights from 0 to 200:
    - For each edge of that weight, use the DSU to check if the two students can be connected.
    - If so, perform the union and add the edge's weight to the total candy count.
6. After all connections are made, print the total candies used.

Code Structure

```
main():
    Read grid dimensions r and c
    Flatten the 2D grid into a 1D array 'grid'

    Initialize a list of 201 buckets to group edges by weight
    For each student (cell):
        - Add edge to bottom neighbor (if exists)
        - Add edge to right neighbor (if exists)
        - Edge weight is min(shyness of u, shyness of v)

    Initialize a parent array for DSU with -1 values

    total = 0
    For w in 0 to 200:
        For each edge (u, v) in bucket w:
            If union(u, v) is successful:
                total += w

    Print total
```

Time Complexity:

- Reading and processing the grid: $O(r \cdot c)$
- Bucketing edges: $O(r \cdot c)$
- DSU operations for Kruskal: $O(N \cdot \alpha(N))$, where $N = r \cdot c$

Overall Time Complexity: $\mathbf{O(r \cdot c)}$ amortized
Space Complexity: $\mathbf{O(r \cdot c)}$ for grid, buckets, and DSU

References

- Kruskal's Algorithm:
    - Prof Yan Gu Slides: 18-Graph
    - `https://cp-algorithms.com/graph/mst_kruskal.html`
- Disjoint Set Union (DSU): `https://cp-algorithms.com/data_structures/disjoint_set_union.html`

2 B. Capital City: Codeforces Submission ID: 321984427

Explanation:

We are given a directed graph representing cities and unidirectional roads between them. The task is to determine which cities can be selected as the capital city of the kingdom. A city qualifies as a capital if and only if it is reachable from every other city in the graph.

This requirement is equivalent to identifying all nodes in a strongly connected component (SCC) that acts as a **sink** in the condensation graph. The condensation graph is a Directed Acyclic Graph (DAG) where each SCC is treated as a single node. A sink component in this DAG has no outgoing edges and must be uniquely reachable from all others for any of its nodes to qualify as a valid capital.

Plan of Attack:

We apply Kosaraju's algorithm to efficiently identify all strongly connected components in the graph.

- Construct the original graph and its reverse.
- Perform the first DFS pass on the original graph to compute the finish order of nodes.
- Perform the second DFS pass on the reversed graph in reverse finish order to assign SCC IDs.
- Build a condensation DAG using SCC IDs and track the out-degree of each component.
- Identify sink components, components with zero out-degree.
- If there is exactly one sink SCC, return all nodes belonging to it as candidate capital cities.
- If there are multiple sink SCCs or none, return 0.

Algorithm Steps:

1. Read the number of nodes n and edges m.
2. Build two adjacency lists:
   - g for the original directed graph.
   - rg for the reversed graph.
3. Run the first depth-first search (DFS) iteratively to compute the finish order.
4. Run the second DFS on the reversed graph in reverse finish order to label all strongly connected components (SCCs).
5. Construct a condensation DAG by iterating over each edge and checking if the source and destination belong to different components.
6. Count the out-degree of each SCC. Track SCCs with out-degree zero.
7. If exactly one such SCC exists, all its constituent nodes are candidate capitals.
8. Otherwise, there are no valid capital cities.

Code Structure

```
main():
    - Read input values n (number of nodes) and m (number of edges).
    - Create adjacency lists for the original graph (g) and the reverse graph (rg).

    First Pass (DFS for Finish Order):
        - Initialize visited array and empty stack for each node.
        - Perform an iterative DFS to track the post-order of nodes.
        - This determines the order in which nodes should be processed in the second pass.

    Second Pass (SCC Assignment):
        - Process nodes in reverse finish order.
        - Run DFS on the reversed graph to group nodes into SCCs.
        - Assign component IDs (comp array) to each node based on which SCC it belongs to.

    Build Condensation Graph:
        - For each edge, check if it connects two different components.
        - If so, increase the out-degree of the source component.

    Sink SCC Detection:
        - Collect all components with zero out-degree.
        - If there's exactly one sink component:
            - Print the number of nodes in it.
            - Print their 1-based indices.
        - Else:
            - Output 0 and an empty line.
```

Time Complexity:

- Building adjacency lists: $O(m)$
- Kosaraju's two-pass DFS: $O(n + m)$
- Condensation graph and sink detection: $O(n + m)$

Overall Time Complexity: $\mathbf{O(n + m)}$
Space Complexity: $\mathbf{O(n + m)}$ for storing the graph and component data

References

- `https://cp-algorithms.com/graph/strongly-connected-components.html`
- `https://www.geeksforgeeks.org/strongly-connected-components/`
- Prof Yan Gu Slides: 19-Graph

3 C. The Speed Bump: Codeforces Submission ID: 321985051

Explanation:

We are given a map of a city represented as a weighted undirected graph with n locations and m roads. Each road connects two locations and is labeled with two attributes: the number of speed bumps and the length (distance) of the road.

Sheldon wants to travel from his home at location 0 to the university at location $n-1$. His objective is twofold:

1. Minimize the total number of speed bumps encountered along the route.
2. Among all such routes with the minimum number of speed bumps, choose the one with the shortest distance.

This is a classical shortest-path problem but with a lexicographically ordered cost pair (total_bumps, total_distance), instead of a single scalar value.

Plan of Attack:

To solve this efficiently, we use a modified version of Dijkstra's algorithm that prioritizes the number of speed bumps first, and the distance second. This is achieved by maintaining a tuple $(bumps, distance)$ as the cost metric and comparing these tuples lexicographically using a priority queue (min-heap).

- Build an adjacency list representation of the graph.
- Maintain a distance array `dist[]` where `dist[u]` stores the best $(bumps, distance)$ found so far to reach node u.
- Initialize the distance of the start node 0 as $(0, 0)$ and push it into the heap.
- While the heap is not empty:
  - Pop the node with the smallest $(bumps, distance)$ value.
  - For each of its neighbors, compute the updated bumps and distance.
  - If the new path offers a better lexicographic value, update the neighbor's entry and push it into the heap.
- When the destination node $n-1$ is reached, the value in `dist[n-1]` gives the answer.

Algorithm Steps:

1. Read the values of n and m, and initialize an adjacency list to store the graph.
2. For each of the m roads, read four integers a_i, b_i, c_i, d_i, where:
   - a_i and b_i are the connected nodes,
   - c_i is the number of speed bumps,
   - d_i is the length of the road.
3. Use a min-heap to implement a modified Dijkstra's algorithm:
   - Each node in the heap stores a tuple $(bumps, distance, node)$.
   - Always expand the node with the lexicographically smallest pair.
   - Only update the distance if the new pair is better.
4. After processing all nodes, output the result $(bumps, distance)$ for node $n-1$.

Code Structure

```
main():
    - Read number of nodes n and edges m.
    - Construct a graph as an adjacency list, where each edge contains:
        (neighbor, speed_bumps, length)
    - Initialize distance array with (INF, INF) for all nodes.
    - Set distance of source node (node 0) to (0, 0).
    - Use a min-heap (priority queue) and push (0, 0, 0).

    While heap is not empty:
        - Pop the node with the smallest (bumps, length) value.
        - If the popped entry is outdated (not equal to dist[u]), skip.
        - For each neighbor:
            - Compute new_bumps = bumps_so_far + current_edge_bumps
            - Compute new_len = len_so_far + current_edge_length
            - If (new_bumps, new_len) is better than dist[v]:
                - Update dist[v] and push into heap.

    Print dist[n - 1], which contains (min_bumps, min_len).
```

Time Complexity:

- Graph Construction: $O(m)$
- Modified Dijkstra's Algorithm: $O((n+m)\log n)$

Overall Time Complexity: $\mathbf{O((n+m)\log n)}$
Space Complexity: $\mathbf{O(n+m)}$ for graph and heap structures

References

- `https://cp-algorithms.com/graph/dijkstra.html`
- `https://stackoverflow.com/questions/22897209/how-to-use-tuples-in-dijkstras-algorithm`