CS-218-HW5-Challenge

Lakhan Kumar Sunilkumar

06 June 2025

Student ID: 862481700

1 A. Shuttles: Codeforces Submission ID: 322790935

Explanation:

We are given a list of subway stations and Olympic event venues, each defined by their 2D coordinates. A shuttle operates between a venue and any subway station that lies within a given Euclidean distance d. These shuttles require charging stations at either endpoint of the route to function.

The task is to determine the minimum number of charging stations needed such that every active shuttle route has at least one of its endpoints equipped with a charging station.

Plan of Attack:

To model this problem efficiently, we interpret the input as a bipartite graph:

- Left set of nodes: Subway stations.
- Right set of nodes: Venues.
- An edge connects a station i and a venue j if their squared Euclidean distance is $\leq d^2$.

The goal is to ensure that each edge (representing a shuttle route) is covered by placing a charging station on at least one of its endpoints. This reduces to finding a minimum vertex cover in the bipartite graph.

By König's Theorem, in bipartite graphs:

$$\text{Minimum Vertex Cover Size} = \text{Maximum Matching Size}$$

Hence, the problem reduces to computing the maximum matching in this bipartite graph, which can be solved using a depth-first search (DFS) based augmenting path algorithm.

Algorithm Steps:

1. Parse the input values: number of stations n, number of venues m, and the maximum allowed shuttle distance d.
2. Store the coordinates of all stations and venues.
3. Build an adjacency list where each station is connected to venues within distance d.
4. Initialize the matching list: for each venue, track the station it is matched to (or -1 if unmatched).

5. For each station:
    - Use DFS to find an augmenting path that can increase the size of the matching.
    - Track visited venues during DFS to avoid cycles.
    - If an augmenting path is found, update the matching and increment the result.
6. At the end, the result equals the size of the maximum matching, which is also the minimum number of charging stations needed.

Code Structure

```
main():
    Read n, m, d (number of stations, venues, max distance)
    Read coordinates of n subway stations
    Read coordinates of m venues

    Build adjacency list:
        For each station, connect to venues within distance <= d

    Initialize matchR = [-1] * m (venue to station matchings)

    For each station i:
        Initialize visited array of size m (to avoid cycles)
        Attempt DFS to find an augmenting path:
            - If a venue is unmatched, match it
            - If matched, try to re-match its partner

        If match is successful, increment the result

    Print result (minimum number of charging stations)
```

Time Complexity:

- Distance check per station-venue pair: $O(n \cdot m)$
- DFS-based matching: each DFS takes $O(m)$ in worst case, called $O(n)$ times

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

References

- `https://en.wikipedia.org/wiki/K%C5%91nig%27s_theorem_(graph_theory)`
- `https://cp-algorithms.com/graph/edmonds_karp.html`
- `https://www.geeksforgeeks.org/maximum-bipartite-matching/`

2 B. Moving Stairs: Codeforces Submission ID: 322792403

Explanation:

We are given a network of platforms connected via magical stairways inside Hogwarts. Each stairway connects one fixed platform x to two other platforms y and z, but not at the same time:

- At even timestamps t, the stairway connects $x \leftrightarrow y$.
- At odd timestamps t, the stairway connects $x \leftrightarrow z$.

Harry Potter starts at platform 0 at timestamp 0. In one minute, he can either:

- Wait on the current platform, which increments time by 1 and toggles parity (even $\leftrightarrow$ odd).
- Step onto an active stairway. After exactly 1 minute, he arrives at another platform depending on how the stair has changed in that time.

The goal is to find the **minimum time required** for Harry to reach a given destination platform k.

Plan of Attack:

We model this as a **state-space graph search**, where each state is a tuple:

$$(\text{platform},\ \text{parity}) = (u, p)$$

- u: Current platform (from 0 to $n-1$).
- p: Current time parity (0 = even, 1 = odd).

We apply a modified BFS (Breadth-First Search) to find the shortest time to reach platform k. The BFS considers two kinds of transitions:

1. **Wait on the platform:** This flips parity and increases time by 1.
2. **Board a stairway:**
   - At even time, boarding from x or y allows moving to z (post-flip endpoint).
   - At odd time, boarding from x or z allows moving to y (post-flip endpoint).
   - Additionally, when at y or z, it's possible to move to x if currently connected.

Algorithm Steps:

1. Parse input: total number of platforms n, number of stairways m, and destination k.
2. Store each stairway as a triple (x, y, z), meaning:
   - Connects $x \leftrightarrow y$ at even times.
   - Connects $x \leftrightarrow z$ at odd times.
3. Build an adjacency list: for each platform, store all stairway indices that touch it.
4. Initialize a 2D array dist$[p][u]$ where:
   - dist$[p][u]$: shortest time to reach platform u at time parity p.
   - All distances are initialized to ∞ except dist$[0][0] = 0$.
5. Run BFS from state $(0, 0)$:

    - If waiting helps and the new state is unvisited, enqueue $(u, 1-p)$.
    - For each stair that touches u, check all legal transitions based on parity.

6. At the end, the answer is:
$$\min(\text{dist}[0][k],\ \text{dist}[1][k])$$
If both values are ∞, then print -1.

Code Structure

```
main():
    Read n, m, k
    Store each stairway as (x, y, z)
    Build an adjacency list: for each platform, track which stairs touch it
    Initialize dist[2][n] = INF, with dist[0][0] = 0

    Initialize queue with (0, 0) → platform 0 at time 0 (even)

    While queue is not empty:
        (u, p) = current state
        Try to WAIT → move to (u, 1-p) if unvisited
        For each stair touching u:
            if p == 0 (even):
                if u == x: move to z
                if u == y: move to x and z
            if p == 1 (odd):
                if u == x: move to y
                if u == z: move to x and y
            For each move, update dist and enqueue next state

    Output min(dist[0][k], dist[1][k]) if reachable, else -1
```

Time Complexity:

- Each platform-parity pair is visited at most once: $O(2n)$
- Each stair is processed per adjacent platform: $O(m)$

Overall Time Complexity: $\mathbf{O(n+m)}$
Space Complexity: $\mathbf{O(n+m)}$ for adjacency list and distance table

References

- `https://www.youtube.com/watch?v=G_j14xj-zGc`
- `https://cs.stackexchange.com/questions/168416/shortest-path-between-two-nodes-with-time-dependent-`

3 C. Catch the Theft!: Codeforces Submission ID: 322791779

Explanation:

We are given a city represented as an undirected graph, where intersections are nodes and roads are edges. Each road has an associated weight, denoting the number of police vehicles required to block it. The goal is to determine the minimum total number of police vehicles required to prevent any possible movement between two specific intersections: Place A and Place B.

This is equivalent to computing the minimum cut between nodes A and B in a flow network, where the capacity of each edge corresponds to the number of police vehicles needed to block that road.

According to the Max-Flow Min-Cut Theorem, the minimum cut capacity between two nodes in a graph is equal to the maximum possible flow between them. Hence, we solve this problem by converting the undirected graph into a directed flow network and applying Dinic's Algorithm to compute the maximum flow from A to B.

Plan of Attack:

To efficiently compute the minimum cut using Dinic's algorithm, we perform the following transformations and steps:

- Each undirected road between u and v with cost c is represented using two directed edges:
  - $u \to v$ with capacity c
  - $v \to u$ with capacity c
- For each directed edge, we also create a reverse edge with capacity 0, allowing flow reversals in the residual graph.
- This gives us a total of four directed edges per undirected road.

We then apply Dinic's algorithm to compute the maximum flow from source A to sink B. The result directly corresponds to the minimum total number of police vehicles required to block all routes between A and B.

Algorithm Steps:

1. Parse the input values: number of nodes n, number of roads m, source A, and sink B.
2. Use array-based structures to efficiently store the graph:
   - `head[u]` stores the index of the first outgoing edge from node u.
   - `to[e]` stores the destination of edge e.
   - `cap[e]` stores the remaining capacity on edge e.
   - `nxt[e]` links to the next edge in node u's adjacency list.
3. For each road (u, v, c), create four directed edges to model capacity and reversibility.
4. Perform Dinic's Algorithm:
   - Use BFS to construct the level graph from source A.
   - Use DFS to find blocking flows and send as much flow as possible from A to B.
   - Repeat until no more augmenting paths are found.
5. Output the total accumulated flow, which is the minimum cut between A and B.

Code Structure

```
main():
    Read n, m, A, B (number of nodes, edges, source, target)

    Initialize arrays:
        - head[] as array of size (n + 1), initialized to -1
        - to[], cap[], nxt[] as dynamic arrays using array module

    add_edge(u, v, c):
        - Adds u → v with capacity c, and v → u with capacity 0
        - Also adds v → u with capacity c, and u → v with capacity 0
        - Maintains residual capacities and reversibility

    For each edge (u, v, c), add four directed arcs as above

    Dinic’s Algorithm:
        - BFS builds level graph from source A
        - DFS pushes flow through blocking paths
        - Repeat until no path exists from A to B

    Output the final flow value (minimum number of police vehicles)
```

Time Complexity:

- Each level graph construction (BFS): $O(E)$
- Each DFS to find blocking flows: $O(E)$
- Overall iterations: $O(\sqrt{V})$ in practice

Overall Time Complexity: $\mathbf{O(E \cdot \sqrt{V})}$
Space Complexity: $\mathbf{O(E + V)}$, safely under 256MB due to use of compact `array` structures

References

- `https://cp-algorithms.com/graph/dinic.html`
- `https://cp-algorithms.com/graph/edmonds_karp.html`

4 D. Go Cycling: Codeforces Submission ID: 322792236

Explanation:

We are given an $n \times m$ grid where each cell contains a height value. Yan starts his bike ride from any cell in the top row (an entrance) and can move only to strictly lower neighboring cells (up, down, left, right). Each top cell requires a one-time ticket to access, and bottom-row cells are considered exits. From any exit, Yan can return to any entrance for free.

The objective is to determine whether all exits (bottom-row cells) can be reached through downhill paths starting from some entrance. If yes, we must compute the minimum number of entrance tickets required so that Yan can cover all exits.

Plan of Attack:

This problem involves three main phases, combining grid traversal and interval covering:

- **Phase 1: Downhill Reachability (BFS)**
  Perform a multi-source BFS from every top-row cell, only traversing downhill neighbors. Mark all reachable cells. If not all bottom-row cells are reachable, print the number of reachable ones and terminate.
- **Phase 2: Interval Propagation (Reverse Dijkstra)**
  For each bottom-row cell (exit), define its reachable interval $[j, j]$ (column index). Propagate these intervals upward through higher neighbors using a max-heap. Each cell eventually holds the interval of bottom columns it can reach.
- **Phase 3: Greedy Covering**
  Each top-row cell (entrance) has an interval $[L_j, R_j]$ of reachable exits. We select the minimum number of such intervals that together cover all columns in $[0, m-1]$. This is solved via the greedy interval cover algorithm.

Algorithm Steps:

1. Parse the input values: n, m and the height grid as a flat array for efficient indexing.
2. **BFS Reachability:**
   - Start from each top-row cell.
   - Perform BFS downhill and mark all visited cells.
   - Count how many bottom-row columns are reached.
   - If any exit is unreachable, return immediately.
3. **Reverse Dijkstra Interval Propagation:**
   - Initialize each bottom-row cell with interval $[j, j]$.
   - Use a max-heap to process cells in descending order of height.
   - For each cell, update the interval of its higher neighbors.
   - Intervals represent the union of reachable exits via downhill movement.
4. **Greedy Interval Cover:**
   - For each entrance, collect its final interval of reachable exits.
   - Sort all intervals by their starting column.
   - Use a greedy method to cover $[0, m-1]$ with the fewest intervals.

- Each chosen interval corresponds to one required ticket.

5. Print:
    - If not all exits are reachable: 0" and the count of reachable exits.
    - Otherwise: 1" and the minimal number of tickets (intervals).

Code Structure

```
main():
    Read n, m and the grid into a flat list.

    Phase 1 - Reachability:
        - Start BFS from all top-row cells.
        - Traverse only downhill neighbors.
        - Mark reachable cells.
        - Count reachable bottom-row columns.
        - If not all reachable, print 0 and count.

    Phase 2 - Interval Propagation:
        - Initialize L and R arrays to track reachable exit ranges.
        - Push all bottom cells into max-heap with initial intervals.
        - While heap is not empty:
            - Pop the cell with highest height.
            - Try to propagate its interval to higher neighbors.

    Phase 3 - Greedy Cover:
        - For each top-row cell, get its [L, R] interval.
        - Sort intervals by L.
        - Greedily cover [0, m-1] using minimum intervals.

    Print:
        - 1
        - Minimum number of tickets (intervals) needed
```

Time Complexity:

- BFS for reachability: $O(n \cdot m)$
- Reverse Dijkstra with heap: $O(n \cdot m \cdot \log(n \cdot m))$
- Sorting intervals: $O(m \log m)$
- Greedy cover: $O(m)$

Overall Time Complexity: $\mathbf{O(n \cdot m \cdot \log(n \cdot m))}$
Space Complexity: $\mathbf{O(n \cdot m)}$ for storing the grid, intervals, visited cells, and heap.

References

- `https://cp-algorithms.com/graph/dijkstra.html`