BÁO CÁO PHÂN TÍCH THUẬT TOÁN

Đề tài: Giải quyết bài toán "Đấu trường Pokémon" bằng Đồ thị

Người thực hiện: [Tên của bạn]
Lớp/Mã SV: [Thông tin của bạn]

Ngày 4 tháng 12 năm 2025

Mục lục

1 Tóm tắt bài toán

1.1 Đề bài

Cho n Pokémon và m thuộc tính.

- Ma trận thuộc tính A kích thước $n \times m$, trong đó $a_{i,j}$ là giá trị thuộc tính j của Pokémon i.
- Mảng chi phí C kích thước n, trong đó c_i là chi phí thuê Pokémon i.
- **Trạng thái đầu:** Pokémon 1 đang đứng trong đấu trường.
- **Trạng thái đích:** Pokémon n đứng trong đấu trường.

1.2 Các thao tác cho phép

Người chơi có thể thực hiện hai loại thao tác với số lần tùy ý:

1. **Nâng cấp chỉ số:** Chọn một thuộc tính bất kỳ và tăng giá trị lên k đơn vị. Chi phí thực hiện là k.
2. **Thách đấu (Thuê Pokémon):** Chọn Pokémon i và thuộc tính j để đấu với Pokémon hiện tại đang giữ đấu trường (gọi là u).
   - Điều kiện thắng: $a_{i,j} \geq a_{u,j}$.
   - Chi phí thực hiện: c_i.
   - Kết quả: Nếu thắng, Pokémon i sẽ thay thế vị trí của u.

Yêu cầu: Tìm chi phí nhỏ nhất để đưa Pokémon n vào đấu trường.

2 Phương pháp thiết kế và Tính phù hợp

2.1 1. Phương pháp thiết kế (Design Paradigm)

Để giải quyết bài toán này, chúng tôi đã sử dụng kết hợp hai phương pháp thiết kế thuật toán kinh điển:

- **Transform and Conquer (Biến đổi để trị):** Cụ thể là kỹ thuật *Representation Change* (Thay đổi biểu diễn). Chúng tôi biến đổi bài toán thao tác trạng thái ban đầu thành bài toán **Tìm đường đi ngắn nhất trên đồ thị (Shortest Path on Graph)**.
- **Greedy (Tham lam):** Sau khi mô hình hóa thành đồ thị, chúng tôi sử dụng thuật toán **Dijkstra** để tìm chi phí tối ưu.

2.2 2. Phân tích tính phù hợp (Suitability)

Việc lựa chọn hai phương pháp trên là hoàn toàn phù hợp với đặc thù bài toán vì các lý do sau:

1. **Khắc phục nhược điểm của Vét cạn (Brute Force):** Không gian trạng thái của bài toán là vô hạn (do có thể tăng chỉ số mãi mãi). Phương pháp *Transform and Conquer* giúp giới hạn bài toán lại trong một đồ thị hữu hạn với $N(M+1)$ đỉnh, giúp bài toán trở nên khả thi (Solvable).

2. **Tối ưu hóa tài nguyên (Space/Time Efficiency):** Nếu mô hình hóa đồ thị một cách ngây thơ (nối mọi cặp Pokémon), số cạnh sẽ là $O(N^2)$, gây tràn bộ nhớ với $N = 4 \cdot 10^5$. Kỹ thuật **Đỉnh ảo** (một phần của việc thay đổi biểu diễn) giúp giảm số cạnh xuống $O(NM)$, phù hợp với giới hạn bộ nhớ và thời gian cho phép.

3. **Đảm bảo tính tối ưu toàn cục:** Bài toán yêu cầu tìm "chi phí nhỏ nhất". Trên đồ thị đã xây dựng, trọng số các cạnh luôn không âm ($cost \geq 0$). Đây là điều kiện lý tưởng để thuật toán *Dijkstra* hoạt động. Tính chất *Greedy* của Dijkstra đảm bảo tìm ra nghiệm tối ưu toàn cục nhanh hơn nhiều so với quy hoạch động hay Bellman-Ford trong trường hợp này.

3 Phân tích thuật toán chi tiết

3.1 Cấu trúc Đồ thị

Gọi $N_{total} = n + n \cdot m$. Đồ thị bao gồm:

- **Đỉnh thực** ($0 \to n-1$)**:** Đại diện cho n Pokémon.
- **Đỉnh ảo (từ n trở đi):** Với mỗi cột thuộc tính j, ta sắp xếp n giá trị thuộc tính tăng dần. Đỉnh ảo $V_{k,j}$ đại diện cho mức thách đấu tại giá trị thứ k của thuộc tính j.

3.2 Xây dựng Cạnh và Trọng số

Có 3 loại cạnh để mô phỏng logic bài toán:

Loại 1: Thiết lập thế trận (Thực → Ảo)

Từ Pokémon u đang giữ sàn, nó có thể chấp nhận thách đấu ở bất kỳ thuộc tính j nào với mức độ khó bằng chỉ số hiện tại của nó.

$$u \xrightarrow{\text{cost }=0} V_{pos(u),j}$$

Loại 2: Thuê Pokémon (Ảo → Thực)

Từ một mức thách đấu $V_{pos(v),j}$ (tương ứng với giá trị của Pokémon v), ta thuê Pokémon v để chiếm sàn.

$$V_{pos(v),j} \xrightarrow{\text{cost }=c_v} v$$

Loại 3: Chuyển đổi mức độ (Giữa các đỉnh ảo)

Xét cột thuộc tính j đã sắp xếp, với hai đỉnh liền kề: V_{k-1} (giá trị nhỏ) và V_k (giá trị lớn).

- **Đi lên (Mạnh thắng Yếu):** Nếu Pokémon thách đấu có chỉ số cao (V_k) đấu với người giữ sàn chỉ số thấp (V_{k-1}), không tốn phí nâng cấp.

$$V_{k-1} \xrightarrow{\text{cost } = 0} V_k$$

- **Đi xuống (Yếu thắng Mạnh):** Nếu Pokémon thách đấu có chỉ số thấp (V_{k-1}) đấu với người giữ sàn chỉ số cao (V_k), ta phải nâng cấp chỉ số. Chi phí là phần chênh lệch.

$$V_k \xrightarrow{\text{cost } = val(V_k) - val(V_{k-1})} V_{k-1}$$

4 Chứng minh tính đúng đắn

Giả sử cần chuyển trạng thái từ Pokémon u sang Pokémon v tại thuộc tính j. Gọi $val_u = a_{u,j}$ và $val_v = a_{v,j}$.

1. **Trường hợp 1:** $val_v \geq val_u$ **(Không cần nâng cấp)**
   - Theo đồ thị, ta đi từ $u \to V_u \to \dots (\text{cạnh đi lên}) \dots \to V_v \to v$.
   - Tổng trọng số: $0 + (0 + \dots + 0) + c_v = c_v$.
   - Đúng với đề bài (chỉ mất phí thuê).
2. **Trường hợp 2:** $val_v < val_u$ **(Cần nâng cấp)**
   - Theo đồ thị, ta đi từ $u \to V_u \to \dots (\text{cạnh đi xuống}) \dots \to V_v \to v$.
   - Tổng trọng số các cạnh đi xuống là tổng các hiệu số liền kề (chuỗi telescopic):
   $$\sum (val_k - val_{k-1}) = val_{start} - val_{end} = a_{u,j} - a_{v,j}$$
   - Tổng chi phí toàn trình: $0 + (a_{u,j} - a_{v,j}) + c_v$.
   - Đúng với đề bài (Phí thuê + Phí nâng cấp phần thiếu hụt).

Do Dijkstra luôn tìm đường đi ngắn nhất trên đồ thị trọng số dương, kết quả tìm được chắc chắn là tối ưu.

5 Phân tích độ phức tạp

Gọi $S = N \times M$.

1. **Độ phức tạp thời gian:**
   - Sắp xếp M cột thuộc tính: $O(M \cdot N \log N)$.
   - Thuật toán Dijkstra trên đồ thị với $V \approx S$ đỉnh và $E \approx 4S$ cạnh: $O(E \log V) \approx O(S \log S)$.
   - Với $N \cdot M \leq 4 \cdot 10^5$, thuật toán thực thi rất nhanh (dưới 1 giây).
2. **Độ phức tạp không gian:**
   - Lưu trữ ma trận A, mảng C, danh sách kề adj và mảng khoảng cách $dist$: $O(N \cdot M)$.
   - Phù hợp với giới hạn bộ nhớ thông thường (256MB - 512MB).

6 Mã nguồn cài đặt (C++)

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <queue>

using namespace std;

const long long INF = 1e18;

struct Edge {
    int to;
    long long weight;
};

struct NodeState {
    long long dist;
    int u;
    bool operator>(const NodeState& other) const {
        return dist > other.dist;
    }
};

void solve() {
    int n, m;
    if (!(cin >> n >> m)) return;

    vector<long long> c(n);
    for (int i = 0; i < n; ++i) cin >> c[i];

    vector<vector<long long>> a(n, vector<long long>(m));
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            cin >> a[i][j];
        }
    }

    int num_total = n + n * m;
    vector<vector<Edge>> adj(num_total);

    for (int j = 0; j < m; ++j) {
        vector<pair<long long, int>> col_vals(n);
        for (int i = 0; i < n; ++i) {
            col_vals[i] = {a[i][j], i};
        }
        sort(col_vals.begin(), col_vals.end());

        int start_node_id = n + j * n;

        for (int k = 0; k < n; ++k) {
            int current_virtual = start_node_id + k;
            int pokemon_id = col_vals[k].second;
            long long val = col_vals[k].first;

            // 1. Thuc -> Ao (Cost 0)
            adj[pokemon_id].push_back({current_virtual, 0});
```

```cpp

            // 2. Ao -> Thuc (Cost c[i])
            adj[current_virtual].push_back({pokemon_id, c[pokemon_id]});

            // 3. Canh giua cac dinh ao
            if (k > 0) {
                int prev_virtual = start_node_id + k - 1;
                long long prev_val = col_vals[k-1].first;

                // Di len (Nho -> Lon): Cost 0
                adj[prev_virtual].push_back({current_virtual, 0});

                // Di xuong (Lon -> Nho): Cost = Hieu so
                adj[current_virtual].push_back({prev_virtual, val - prev_val});
            }
        }
    }

    // Dijkstra
    priority_queue<NodeState, vector<NodeState>, greater<NodeState>> pq;
    vector<long long> dist(num_total, INF);

    dist[0] = 0;
    pq.push({0, 0});

    while (!pq.empty()) {
        long long d = pq.top().dist;
        int u = pq.top().u;
        pq.pop();

        if (d > dist[u]) continue;
        if (u == n - 1) {
            cout << d << "\n";
            return;
        }

        for (const auto& edge : adj[u]) {
            if (dist[u] + edge.weight < dist[edge.to]) {
                dist[edge.to] = dist[u] + edge.weight;
                pq.push({dist[edge.to], edge.to});
            }
        }
    }
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    int t;
    if (cin >> t) {
        while (t--) {
            solve();
        }
    }
    return 0;
}
```