

BÁO CÁO NHÓM 6

Nguyễn Hồng Phúc, Lê Quang Trung

Ngày 4 tháng 12 năm 2025

1 Phát biểu bài toán

Bài toán yêu cầu tìm chi phí nhỏ nhất để đưa Pokemon thứ n ra sân đấu, xuất phát từ trạng thái ban đầu chỉ có Pokemon thứ 1 trên sân.

Chúng ta có n Pokemon, mỗi Pokemon có m thuộc tính. Để thay đổi Pokemon trên sân, ta phải thực hiện hành động "thuê" một Pokemon mới để đấu với Pokemon hiện tại. Pokemon mới sẽ thắng và ở lại sân nếu thuộc tính j được chọn của nó **lớn hơn hoặc bằng** thuộc tính j của Pokemon đang đứng sân ($a_{i,j} \geq a_{current,j}$). Ngoài ra, ta có thể tốn chi phí k để tăng chỉ số thuộc tính vĩnh viễn nhằm thỏa mãn điều kiện thắng.

2 Chiến lược phát triển thuật toán

Đây là bài toán tìm chuỗi thao tác tối ưu (chi phí thấp nhất), một dạng bài toán kinh điển có thể mô hình hóa bằng **GRAPH THEORY**. Chiến lược giải quyết bài toán sẽ đi từ cách tiếp cận ngay thơ đến tối ưu hóa cấu trúc đồ thị:

- **Mô hình hóa sơ khai:** Coi mỗi Pokemon là một đỉnh. Tuy nhiên, việc chuyển đổi giữa các Pokemon phụ thuộc vào m thuộc tính và khả năng nâng cấp chỉ số, tạo ra vô số trạng thái cạnh tranh phức tạp nếu xét duyệt toàn bộ (BRUTE-FORCE).
- **TRANSFORM AND CONQUER:** Thay vì xét quan hệ trực tiếp $N \times N$, ta biến đổi bài toán bằng cách tạo ra các **trạng thái trung gian** dựa trên việc **SORTING** giá trị thuộc tính. Kỹ thuật này giúp giảm số lượng cạnh từ $O(N^2)$ xuống $O(N \cdot M)$.

3 Phân tích hướng tiếp cận và Thiết kế chi tiết

3.1 Ý tưởng

Vấn đề khó nhất của bài toán là thao tác "Tăng thuộc tính với chi phí k ". Nếu xử lý trực tiếp, ta không biết nên tăng bao nhiêu là đủ. Giải pháp là chuyển đổi bài toán về dạng **SHORTEST PATH** bằng cách xây dựng một đồ thị đặc biệt:

1. Nút trạng thái (States):

- $Pos(u)$: Trạng thái Pokemon u đã chiến thắng và đang đứng trên sân.
- $State(u, t)$: Trạng thái trung gian, đại diện cho việc Pokemon u đang được xét theo thuộc tính t .

2. Sorting:

Với mỗi thuộc tính t , ta sắp xếp tất cả n Pokemon theo thứ tự tăng dần của giá trị $a_{i,t}$. Điều này tạo ra một "chuỗi" các giá trị liền kề, cho phép ta mô hình hóa việc tăng chỉ số bằng các cạnh nối giữa các Pokemon có thứ hạng gần nhau.

3.2 Thiết kế đồ thị (Graph Construction)

Dựa trên mã nguồn cài đặt, đồ thị được xây dựng với các loại cạnh sau:

- **Upgrade Edge:** Trên trực thuộc tính t đã sắp xếp, xét hai Pokemon u (hạng r) và v (hạng $r+1$):
 - Từ $v \rightarrow u$ (giá trị lớn về nhỏ): Chi phí 0.
 - Từ $u \rightarrow v$ (giá trị nhỏ lên lớn): Chi phí $a_{v,t} - a_{u,t}$.

- **Selection Edge:**
 - Từ $Pos(u) \rightarrow State(u, t)$: Chi phí 0. (Từ vị trí đứng sân, ta có thể xét bất kỳ thuộc tính t nào của u để làm mốc so sánh).
- **Battle Edge:**
 - Từ $State(u, t) \rightarrow Pos(u)$: Chi phí c_u . (Quyết định thuê u để đấu).

3.3 Thuật toán Conquer (Dijkstra)

Sau khi Transform bài toán về đồ thị, ta áp dụng thuật toán **DIJKSTRA**:

- **Khởi tạo:** Khoảng cách tới tất cả các đỉnh là vô cùng (INF), ngoại trừ điểm xuất phát ($S \rightarrow State(1, t)$ với chi phí 0).
- **Thực thi:** Sử dụng Priority Queue để liên tục mở rộng các trạng thái có chi phí thấp nhất.
- **Kết quả:** Chi phí nhỏ nhất để chạm tới trạng thái có Pokémon n đứng sân là $\min(dist[State(n, t)] + c_n)$.

4 Phân tích độ phức tạp

4.1 Độ phức tạp thời gian

- **Sorting:** Ta sắp xếp n Pokémon cho mỗi thuộc tính trong m thuộc tính: $O(m \cdot n \log n)$.
- **Dijkstra:**
 - Số lượng đỉnh $V \approx n \cdot m$.
 - Số lượng cạnh $E \approx 4 \cdot n \cdot m$.
 - Độ phức tạp Dijkstra tiêu chuẩn là $O(E \log V)$. $\Rightarrow O(nm \log(nm))$.

Với n, m trong giới hạn đề bài ($nm \leq 4 \cdot 10^5$), thuật toán chạy tốt trong giới hạn 5 giây.

4.2 Độ phức tạp không gian

- Lưu trữ đồ thị: $O(V + E) \approx O(nm)$.
- Các mảng phụ trợ (rank, dist): $O(nm)$.

Tổng cộng: $O(nm)$, hoàn toàn thỏa mãn giới hạn 512MB.

5 Lý do lựa chọn phương pháp

1. **Tính đúng đắn:** Phương pháp **BRUTE-FORCE** không thể áp dụng do không gian trạng thái quá lớn. Việc mô hình hóa thành đồ thị giúp bao quát tất cả các trường hợp nâng cấp chỉ số và lựa chọn đối thủ một cách toán học chặt chẽ. **DIJKSTRA** đảm bảo tìm ra nghiệm tối ưu toàn cục cho đồ thị trong số không âm.
2. **Tối ưu hóa nhờ cấu trúc dữ liệu:** Bài toán này dùng **SORTING** để "tuyến tính hóa" quan hệ giữa các Pokémon. Thay vì so sánh $O(n^2)$, ta chỉ cần so sánh với lân cận $O(n)$, giảm độ phức tạp xuống mức chấp nhận được.

6 Mã nguồn tham khảo (Python)

```
1 import sys
2 import heapq
3
4 def solve():
5     input_data = sys.stdin.read().split()
6     if not input_data:
7         return
8
9     iterator = iter(input_data)
10    try:
11        num_test_cases = int(next(iterator))
12    except StopIteration:
13        return
14
15    for _ in range(num_test_cases):
16        try:
17            n = int(next(iterator))
18            m = int(next(iterator))
19        except StopIteration:
20            break
21
22        c = [0] * (n + 1)
23        for i in range(1, n + 1):
24            c[i] = int(next(iterator))
25
26        a = [[0] * (m + 1) for _ in range(n + 1)]
27        b = [[] for _ in range(m + 1)]
28
29        for i in range(1, n + 1):
30            for j in range(1, m + 1):
31                val = int(next(iterator))
32                a[i][j] = val
33                b[j].append((val, i))
34
35        dec = [[0] * (n + 1) for _ in range(m + 1)]
36
37        for j in range(1, m + 1):
38            b[j].sort()
39            for r_idx, (val, pid) in enumerate(b[j]):
40                dec[j][r_idx + 1] = pid
41
42        def get_state(x, t):
43            return (x - 1) * m + t
44
45        def get_pos(x):
46            return n * m + x
47
48        STATE_CNT = n * m
49        S = STATE_CNT + n + 1
50        N_total = S
51        adj = [[] for _ in range(N_total + 1)]
52
53        def add_edge(u, v, w):
54            adj[u].append((v, w))
55
56        for t in range(1, m + 1):
57            for r in range(1, n + 1):
58                x = dec[t][r]
59                u = get_state(x, t)
60
61                if r < n:
62                    z = dec[t][r + 1]
63                    v = get_state(z, t)
64                    add_edge(u, v, 0)
65
66                if r > 1:
67                    z = dec[t][r - 1]
68                    v = get_state(z, t)
69                    add_edge(u, v, a[x][t] - a[z][t])
70
71        for x in range(1, n + 1):
72            px = get_pos(x)
73            for t in range(1, m + 1):
```

```

74         u = get_state(x, t)
75         add_edge(u, px, 0)
76
77         v = get_state(x, t)
78         add_edge(px, v, c[x])
79
80     for t in range(1, m + 1):
81         v = get_state(1, t)
82         add_edge(S, v, 0)
83
84     INF = 4 * 10**18
85     dist = [INF] * (N_total + 1)
86     dist[S] = 0
87     pq = [(0, S)]
88
89     while pq:
90         d, u = heapq.heappop(pq)
91         if d > dist[u]:
92             continue
93
94         for v, w in adj[u]:
95             nd = d + w
96             if nd < dist[v]:
97                 dist[v] = nd
98                 heapq.heappush(pq, (nd, v))
99
100    ans = INF
101    for t in range(1, m + 1):
102        v = get_state(n, t)
103        if dist[v] < INF:
104            val = dist[v] + c[n]
105            if val < ans:
106                ans = val
107
108    print(ans)
109
110 if __name__ == "__main__":
111     solve()

```