1 Nhóm 13

2 Phương pháp thiết kế (0.4)

Để giải quyết bài toán Pokémon, chúng tôi áp dụng phương pháp **Mô hình hóa Đồ thị (Graph Modeling)** kết hợp với thuật toán **Dijkstra** để tìm đường đi ngắn nhất. Tư duy thiết kế được chia nhỏ thành các bước lập luận sau:

2.1 1. Trừu tượng hóa bài toán (Abstraction)

Bài toán yêu cầu thay đổi trạng thái từ "Pokémon 1 đứng sân"sang "Pokémon n đứng sân"với chi phí tối thiểu. Chúng tôi ánh xạ các thực thể của bài toán sang lý thuyết đồ thị như sau:

- **Trạng thái (Nodes):** Mỗi Pokémon i ($1 \leq i \leq n$) được coi là một đỉnh trong đồ thị $G = (V, E)$. Đỉnh u đại diện cho trạng thái "Pokémon u đang đứng trong đấu trường".
- **Hành động (Edges):** Việc thuê Pokémon v để đánh bại Pokémon u tương ứng với một cạnh có hướng từ $u \to v$. Do bất kỳ Pokémon nào cũng có thể thách đấu Pokémon đang đứng sân, đồ thị này là *đồ thị đầy đủ (Complete Graph)* hoặc *đồ thị dày (Dense Graph)*.
- **Trọng số (Weights):** Chi phí để thực hiện hành động chuyển từ $u \to v$ chính là trọng số của cạnh $w(u, v)$.

2.2 2. Xây dựng hàm mục tiêu cục bộ (Edge Weight Formulation)

Vấn đề cốt lõi là xác định chi phí nhỏ nhất để chuyển từ u sang v. Theo đề bài, ta có thể chọn bất kỳ thuộc tính $j \in [1, m]$ để thi đấu. Để tối ưu hóa, ta phải chọn thuộc tính j sao cho tổng chi phí (thuê + nâng cấp) là thấp nhất. Công thức trọng số cạnh được thiết lập như sau:

$$w(u, v) = c_v + \min_{1 \leq j \leq m} \left(\underbrace{\max(0, a_{u,j} - a_{v,j})}_{\text{Chi phí nâng cấp chỉ số}} \right)$$

Trong đó: c_v là chi phí thuê cố định, phần còn lại là chi phí nâng cấp thuộc tính thứ j của v sao cho $a_{v,j} \geq a_{u,j}$.

2.3 3. Lựa chọn thuật toán tìm đường (Algorithm Selection)

Sau khi mô hình hóa, bài toán trở thành: *Tìm đường đi ngắn nhất từ đỉnh 1 đến đỉnh n trên đồ thị có hướng, có trọng số không âm.* Phương pháp được chọn là **Thuật toán Dijkstra**.

3 Tính phù hợp (0.3)

Việc lựa chọn phương pháp Mô hình hóa Đồ thị và thuật toán Dijkstra không phải là ngẫu nhiên mà dựa trên các đặc tính kỹ thuật sau:

3.1 1. Tại sao không dùng thuật toán Tham lam (Greedy)?

Một cách tiếp cận ngây thơ là: *"Tại mỗi bước, luôn chọn Pokémon tiếp theo có chi phí chuyển đổi rẻ nhất"*.

- **Phản biện:** Chiến lược này thất bại vì nó chỉ tối ưu cục bộ (Local Optimum). Việc chọn một Pokémon rẻ nhưng có chỉ số yếu (stats thấp) để đứng sân có thể khiến chi phí cho các bước *tiếp theo* tăng vọt, vì Pokémon yếu đó rất khó bị đánh bại bởi Pokémon đích (cần nâng cấp nhiều).
- **Giải pháp:** Dijkstra khắc phục điều này bằng cách duy trì nhãn $D[i]$ (chi phí nhỏ nhất từ nguồn), đảm bảo tính tối ưu toàn cục (Global Optimum).

3.2 2. Tại sao Dijkstra phù hợp hơn Bellman-Ford hay Floyd-Warshall?

- **Tính chất trọng số không âm:** Chi phí thuê $c_i \geq 0$ và chi phí nâng cấp $k \geq 0$, do đó $w(u,v) \geq 0$. Đây là điều kiện tiên quyết để Dijkstra hoạt động chính xác. Bellman-Ford là không cần thiết (chậm hơn) vì không có cạnh âm.
- **Đặc thù bài toán:** Ta chỉ cần tìm đường đi từ **một nguồn** (đỉnh 1) đến **một đích** (đỉnh n). Floyd-Warshall ($O(N^3)$) giải quyết cho mọi cặp đỉnh là thừa thãi và quá chậm với $N = 10^3 \sim 10^4$.

3.3 3. Sự phù hợp với ràng buộc dữ liệu

Bài toán có đặc điểm là số lượng cạnh rất lớn ($E \approx N^2$) nhưng số đỉnh N vừa phải. Dijkstra phiên bản duyệt mảng (không dùng Heap) cực kỳ hiệu quả với đồ thị dày, giảm thiểu chi phí quản lý cấu trúc dữ liệu phức tạp.

4 Phân tích độ phức tạp (0.3)

4.1 1. Độ phức tạp thời gian (Time Complexity)

Chúng tôi phân tích dựa trên phiên bản Dijkstra duyệt mảng (Dense Dijkstra), phù hợp nhất cho đồ thị đầy đủ.

Gọi N là số lượng Pokémon, M là số lượng thuộc tính.

1. **Khởi tạo:** Mất $O(N)$ để thiết lập mảng khoảng cách.
2. **Vòng lặp chính (Main Loop):** Thuật toán thực hiện N bước lặp để cố định nhãn cho N đỉnh.
    - *Tìm đỉnh u có $D[u]$ nhỏ nhất:* Mất $O(N)$ thao tác duyệt mảng.
    - *Thao tác làm trơn (Relaxation):* Với mỗi đỉnh u được chọn, ta xét tất cả N đỉnh v còn lại để cập nhật đường đi.
    - *Tính trọng số cạnh $w(u,v)$:* Để tính chi phí giữa u và v, ta cần duyệt qua M thuộc tính để tìm min. Bước này mất $O(M)$.
    - $\Rightarrow$ Tổng chi phí cho bước Relaxation trong một vòng lặp: $N \times O(M) = O(N \cdot M)$.

Tổng độ phức tạp thời gian:

$$T(N,M) = N \times (O(N) + O(N \cdot M)) = O(N^2 + N^2 M) \approx \mathbf{O(N^2 \cdot M)}$$

Đánh giá thực tế

Với $N \approx 2000$ và $M \approx 100$:

$$Operations \approx 2000^2 \times 100 = 4 \cdot 10^8 \text{ phép tính.}$$

Giới hạn thời gian là **5 giây** (tương đương khả năng xử lý $\approx 10^9 \sim 2.5 \cdot 10^9$ phép tính trên các máy chấm hiện đại như Codeforces/Judges). $\Rightarrow$ Thuật toán chạy an toàn trong khoảng **0.5s - 1s**.

4.2 2. Độ phức tạp không gian (Space Complexity)

- **Lưu trữ dữ liệu đầu vào:** Ma trận thuộc tính $A[N][M]$ tiêu tốn $O(N \cdot M)$.
- **Lưu trữ thuật toán:** Mảng khoảng cách $D[N]$, mảng đánh dấu $Visited[N]$ tiêu tốn $O(N)$.

- Chúng ta **không** cần lưu trữ danh sách kề (Adjacency List) hay ma trận trọng số (Adjacency Matrix) vì trọng số cạnh được tính toán trực tiếp (on-the-fly) khi cần. Điều này giúp tiết kiệm lượng lớn bộ nhớ ($O(N^2)$).

Tổng độ phức tạp không gian: $\mathbf{O(N \cdot M)}$. Với giới hạn bộ nhớ **512 MB**, không gian sử dụng chỉ vài chục MB là hoàn toàn tối ưu.