**ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH**
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

Khoa Khoa học Máy tính - KHTN2024

# PHÂN TÍCH VÀ THIẾT KẾ THUẬT TOÁN

# CS112

---

## Báo cáo giải bài tập: C. Pokémon

---

GVHD: Nguyễn Thanh Sơn

SV thực hiện: Bùi Huỳnh Tây – 24521589
SV thực hiện: Phạm Ngọc Thọ – 245200xx

Tp. Hồ Chí Minh, Tháng 12/2025

# Contents

# 1 Chương 1 - Tóm tắt đề bài

## 1.1 Mô tả bài toán

Bạn sở hữu $N$ Pokémon. Ban đầu, chỉ có Pokémon thứ 1 đang đứng trong đấu trường. Mục tiêu của bạn là đưa Pokémon thứ $N$ lên đứng đấu trường với chi phí thấp nhất.

**Dữ liệu vào:**

- Mỗi Pokémon $i$ có chi phí thuê là $c_i$.
- Mỗi Pokémon $i$ có $M$ thuộc tính, thuộc tính thứ $j$ là $a_{i,j}$.

**Thao tác cho phép:**

1. **Nâng cấp:** Chọn 3 số nguyên $i, j, k$. Tăng thuộc tính $a_{i,j}$ lên $k$ đơn vị vĩnh viễn. Chi phí là $k$.
2. **Thi đấu:** Thuê Pokémon $i$ đấu với Pokémon hiện tại $u$ dựa trên thuộc tính $j$. Nếu $a_{i,j} \geq a_{u,j}$, Pokémon $i$ thắng và thay thế vị trí. Chi phí là $c_i$.

**Chi phí chuyển đổi:** Để Pokémon $i$ thắng Pokémon $u$ bằng thuộc tính $j$, tổng chi phí tối thiểu cần trả là:

$$\text{Cost}(u \to i) = c_i + \max(0, a_{u,j} - a_{i,j})$$

Phần $\max(0, a_{u,j} - a_{i,j})$ chính là chi phí nâng cấp $a_{i,j}$ nếu nó nhỏ hơn $a_{u,j}$.

# 2 Chương 2 - Phương pháp thiết kế thuật toán

Để giải quyết bài toán này, nhóm sử dụng kết hợp hai phương pháp thiết kế thuật toán chính: **Transform and Conquer (Biến đổi và Trị)** và **Greedy (Tham lam)**.

## 2.1 Transform and Conquer (Biến đổi và Trị)

Theo lý thuyết, Transform and Conquer gồm hai giai đoạn: biến đổi bài toán gốc thành một dạng khác dễ giải quyết hơn, sau đó giải bài toán đã biến đổi. Cụ thể ở đây, chúng tôi sử dụng biến thể **Problem Reduction (Quy về bài toán khác)**.

- **Bài toán gốc:** Tìm chuỗi thao tác nâng cấp và thi đấu để đưa Pokémon $N$ lên sàn đấu với chi phí tối thiểu.
- **Biến đổi:** Chúng tôi quy bài toán này về **bài toán tìm đường đi ngắn nhất trên đồ thị có hướng**.
- **Chi tiết biến đổi:**
  - Mỗi Pokémon được xem là một đỉnh (node) trong đồ thị.
  - Chi phí chuyển đổi giữa các Pokémon (bao gồm phí thuê và phí nâng cấp chênh lệch thuộc tính) được xem là trọng số của các cạnh (edges).
  - Để tối ưu số lượng cạnh (tránh $O(N^2)$), chúng tôi sử dụng kỹ thuật **Representation Change** bằng cách tạo ra các đỉnh ảo (virtual nodes) và sắp xếp các Pokémon theo từng thuộc tính. Điều này giúp giảm không gian tìm kiếm và đơn giản hóa cấu trúc đồ thị.

### 2.2 Greedy Technique (Tham lam)

Sau khi đã biến đổi bài toán về dạng đồ thị với trọng số không âm, chúng tôi áp dụng chiến lược Tham lam để tìm đường đi ngắn nhất.

- **Nguyên lý:** Tại mỗi bước, thuật toán luôn đưa ra lựa chọn tốt nhất cục bộ (đỉnh có khoảng cách ngắn nhất hiện tại) với hy vọng đạt được tối ưu toàn cục.
- **Thuật toán cụ thể:** Sử dụng **Dijkstra**. Đây là thuật toán tham lam kinh điển để giải quyết bài toán đường đi ngắn nhất một nguồn (Single-Source Shortest Paths) trên đồ thị có trọng số không âm.

## 3 Chương 3 - Tính phù hợp của giải thuật

Việc lựa chọn kết hợp Transform and Conquer và Greedy là hoàn toàn phù hợp vì các lý do sau:

1. **Đặc điểm bài toán:** Bài toán yêu cầu tìm "chi phí nhỏ nhất". Đây là dấu hiệu điển hình của các bài toán tối ưu có cấu trúc con tối ưu (Optimal Substructure), nơi mà lời giải tối ưu của bài toán lớn chứa đựng lời giải tối ưu của các bài toán con.
2. **Tại sao dùng Problem Reduction (Đồ thị)?** Cách tiếp cận Brute Force (Vét cạn) sẽ phải thử tất cả các chuỗi thao tác, dẫn đến độ phức tạp hàm mũ, không khả thi với $N = 4 \cdot 10^5$. Việc chuyển sang mô hình đồ thị cho phép ta tận dụng các thuật toán hiệu quả đã biết (như Dijkstra).
3. **Tại sao dùng Greedy (Dijkstra) thay vì Dynamic Programming?**
   - Mặc dù Dynamic Programming (Quy hoạch động) cũng giải quyết tốt các bài toán tối ưu, nhưng nó thường yêu cầu duyệt qua không gian trạng thái lớn hoặc xây dựng bảng phương án tốn kém bộ nhớ.
   - Trong bài toán này, trọng số cạnh (chi phí) luôn không âm ($c_i \geq 1, k > 0$). Điều này thỏa mãn điều kiện tiên quyết của thuật toán Dijkstra. Chiến lược tham lam của Dijkstra (chọn đỉnh gần nhất để mở rộng) đảm bảo tìm ra lời giải tối ưu toàn cục một cách hiệu quả hơn so với việc quét toàn bộ không gian trạng thái.
   - Độ phức tạp của Dijkstra với Heap là $O(E \log V)$, rất tốt cho các bài toán có đồ thị thưa sau khi đã tối ưu cạnh bằng đỉnh ảo.

## 4 Chương 4 - Phân tích độ phức tạp

### 4.1 Độ phức tạp thời gian (Time Complexity)

Gọi $S = \sum(N \cdot M)$ là tổng kích thước dữ liệu đầu vào.

- **Giai đoạn Biến đổi (Transform):**
  - Sắp xếp $N$ Pokémon theo $M$ thuộc tính: Tốn $O(M \cdot N \log N)$.
  - Xây dựng đồ thị với đỉnh ảo: Số lượng đỉnh $V \approx 2 \cdot N \cdot M$. Số lượng cạnh $E \approx 4 \cdot N \cdot M$. Quá trình duyệt và tạo cạnh tốn $O(N \cdot M)$.

- **Giai đoạn Trị (Conquer - Dijkstra):**
  - Sử dụng Priority Queue (Heap), độ phức tạp của Dijkstra là $O(E \log V)$.
  - Thay thế $V, E$ vào: $O(N \cdot M \cdot \log(N \cdot M))$.

Tổng độ phức tạp thời gian: $O(N \cdot M \log(N \cdot M))$. Với giới hạn $\sum N \cdot M \leq 4 \cdot 10^5$, thuật toán hoàn toàn đáp ứng giới hạn thời gian (Time Limit).

### 4.2 Độ phức tạp không gian (Space Complexity)

- **Lưu trữ đồ thị:** Ta cần lưu danh sách kề cho $V$ đỉnh và $E$ cạnh.

$$Space \approx O(V + E) \approx O(N \cdot M)$$

- **Mảng khoảng cách (dist) và Heap:** Tốn $O(V) \approx O(N \cdot M)$.

Tổng độ phức tạp không gian: $O(N \cdot M)$. Với giới hạn bộ nhớ 512 MB, việc lưu trữ vài triệu phần tử integer là khả thi.

## 5 Chương 5 - Cài đặt (Source Code)

Dưới đây là mã nguồn Python hiện thực thuật toán:

```python
import sys
import heapq

# Tang gioi han de quy
sys.setrecursionlimit(200000)

def solve():
    def input_generator():
        for line in sys.stdin:
            for token in line.split():
                yield token

    iterator = input_generator()

    try:
        t_str = next(iterator)
        t = int(t_str)
    except StopIteration:
        return

    for _ in range(t):
        try:
            n = int(next(iterator))
            m = int(next(iterator))

            c = []
```

```
            for _ in range(n):
                c.append(int(next(iterator)))

            a = []
            for _ in range(n):
                row = []
                for _ in range(m):
                    row.append(int(next(iterator)))
                a.append(row)

        except StopIteration:
            break

        # Xay dung do thi voi cac dinh ao (Virtual Nodes)
        # Ky thuat: Problem Reduction & Representation Change
        total_nodes = n + 2 * n * m
        adj = [[] for _ in range(total_nodes)]
        current_virtual_id = n

        for j in range(m):
            # Sap xep Pokemon theo thuoc tinh j de toi uu canh
            sorted_pokes = []
            for i in range(n):
                sorted_pokes.append((a[i][j], i))
            sorted_pokes.sort()

            up_start = current_virtual_id
            down_start = current_virtual_id + n
            current_virtual_id += 2 * n

            for k in range(n):
                val, u = sorted_pokes[k]
                up_node = up_start + k
                down_node = down_start + k

                # Tao canh ket noi giua cac node ao va node thuc
                if k + 1 < n:
                    adj[up_node].append((up_node + 1, 0))
                adj[u].append((up_node, 0))
                adj[up_node].append((u, c[u]))

                if k > 0:
                    prev_val = sorted_pokes[k-1][0]
                    diff = val - prev_val
                    adj[down_node].append((down_node - 1, diff))
                adj[u].append((down_node, 0))
                adj[down_node].append((u, c[u]))

        # Thuat toan Greedy (Dijkstra)
        INF = 10**18
```

```
        dist = [INF] * total_nodes
        dist[0] = 0
        pq = [(0, 0)]
        min_cost = -1

        while pq:
            d, u = heapq.heappop(pq)
            if d > dist[u]: continue
            if u == n - 1:
                min_cost = d
                break
            for v, w in adj[u]:
                if dist[u] + w < dist[v]:
                    dist[v] = dist[u] + w
                    heapq.heappush(pq, (dist[v], v))

        print(min_cost)

if __name__ == '__main__':
    solve()
```

Listing 1: Giải thuật Dijkstra với tối ưu đồ thị