Final Report

Hoàng Anh

Ngày 25 tháng 7 năm 2025

Bài toán 1: Ferrers Diagram

Mục đích

Chương trình này sinh ra tất cả các phân hoạch của một số nguyên n thành k phần, và với mỗi phân hoạch, vẽ biểu đồ Ferrers và biểu đồ Ferrers chuyển vị (transpose). Có hai phiên bản: một bằng C++ và một bằng Python.

Giải thích thuật toán

- **Sinh phân hoạch:** Sử dụng đệ quy để sinh tất cả các phân hoạch của n thành k số nguyên dương, không tăng dần.
- **Vẽ Ferrers diagram:** Với mỗi phân hoạch, in ra mỗi dòng số lượng dấu * tương ứng với từng phần tử, căn phải theo phần tử lớn nhất.
- **Vẽ Ferrers transpose:** In ra ma trận chuyển vị của Ferrers diagram, tức là hoán đổi hàng và cột.

Phân tích chi tiết cách hoạt động của thuật toán

Thuật toán trong cả hai chương trình (C++ và Python) gồm hai phần chính: sinh phân hoạch số và vẽ biểu đồ Ferrers (cùng chuyển vị). Dưới đây là phân tích chi tiết:

1. **Sinh phân hoạch số:**
   - Ý tưởng là dùng đệ quy để liệt kê tất cả các cách phân tích số n thành k số nguyên dương, mỗi số không lớn hơn một giá trị cho trước (ban đầu là n).
   - Ở mỗi bước đệ quy, thuật toán chọn một số i (từ $\min(n, \text{max\_val})$ đến 1), thêm vào phân hoạch hiện tại, rồi tiếp tục phân tích $n - i$ thành $k - 1$ phần với số lớn nhất là i.
   - Khi $k = 0$ và $n = 0$, một phân hoạch hợp lệ được tạo ra và sẽ được xử lý tiếp.
2. **Vẽ Ferrers diagram:**

- Với mỗi phân hoạch, biểu đồ Ferrers được vẽ bằng cách in ra mỗi dòng số lượng dấu * tương ứng với từng phần tử của phân hoạch, căn phải theo phần tử lớn nhất.
- Ví dụ, phân hoạch $(3, 1)$ sẽ vẽ:

```
*** 3
*   1
```

3. **Vẽ Ferrers transpose:**
   - Biểu đồ Ferrers chuyển vị (transpose) là hoán đổi hàng và cột của biểu đồ gốc.
   - Cách làm là duyệt từng cột (theo chiều cao lớn nhất của phân hoạch), với mỗi cột kiểm tra xem từng hàng có dấu * không (tức là phần tử phân hoạch lớn hơn chỉ số cột).
   - Ví dụ, transpose của $(3, 1)$ là:

```
* *
*
*
3 1
```

Giải thích mã nguồn C++

- `maxp(const vector<int>& p)`: Tìm phần tử lớn nhất trong phân hoạch p.
- `Ferrers(const vector<int>& p)`: In biểu đồ Ferrers cho phân hoạch p.
- `FerrersTrans(const vector<int>& p)`: In biểu đồ Ferrers chuyển vị cho phân hoạch p.
- `genF(int n, int k, vector<int>& cur, int max_val)`: Đệ quy sinh các phân hoạch của n thành k phần, mỗi phần không lớn hơn max_val.
- `main()`: Đọc n, k từ đầu vào, gọi hàm sinh phân hoạch và in kết quả.

Giải thích mã nguồn Python

- `max_part(partition)`: Tìm phần tử lớn nhất trong phân hoạch.
- `print_ferrers(partition)`: In biểu đồ Ferrers.
- `print_ferrers_transpose(partition)`: In biểu đồ Ferrers chuyển vị.
- `generate_partitions(n, k, current, max_val)`: Đệ quy sinh phân hoạch.
- `main()`: Đọc n, k từ đầu vào, gọi hàm sinh phân hoạch và in kết quả.

Ví dụ đầu vào/đầu ra

```
  Input:
  4 2

  Output:
  F:
  ** 2
  ** 2
  FT:
  * *
  * *
  2 2
  ====================
  F:
  *** 3
  *   1
  FT:
  * *
  *
  *
  3 1
====================
```

So sánh hai phiên bản

- Cả hai phiên bản đều sử dụng cùng một thuật toán sinh phân hoạch và in biểu đồ Ferrers.
- Phiên bản C++ sử dụng `vector<int>` và thao tác nhập/xuất chuẩn của C++.
- Phiên bản Python sử dụng list và cú pháp Pythonic, dễ đọc hơn.
- Kết quả đầu ra của hai chương trình là tương đương.

Mã nguồn C++

```
#include <bits/stdc++.h>
using namespace std;

int maxp(const vector<int>&p) {
    int pmax = 0;
    int n = p.size();
```

```
    for (int i = 0; i < n; ++i) {
        if (p[i] > pmax)
            pmax = p[i];
    }
    return pmax;
}

void Ferrers(const vector<int>&p) {
    cout << "F:\n";
    int pmax = maxp(p);
    for (int r : p) {
        for (int i = 0; i < r; ++i) cout << '*';
        for (int i = 0; i < pmax - r; ++i) cout << "␣";
        cout << "␣" << r << '\n';
    }
}

void FerrersTrans(const vector<int>& p) {
    cout << "FT:\n";
    int pmax = maxp(p);
    int n = p.size();
    for (int r = 0; r < pmax; ++r) {
        for (int i = 0; i < n; ++i) {
            if (p[i] > r) cout << "*␣";
            else cout << "␣";
        }
        cout << '\n';
    }
    for (int i = 0; i < n; ++i) {
        cout << p[i];
        if (i != n - 1) cout << "␣";
    }
    cout << "\n";
}

void genF(int n, int k, vector<int>&cur, int max_val) {
    if (k == 0) {
        if (n == 0) {
            Ferrers(cur);
            FerrersTrans(cur);
            cout << "====================\n";
        }
        return;
    }
    for (int i = min(n, max_val); i >= 1; --i) {
        cur.push_back(i);
        genF(n - i, k - 1, cur, i);
```

```
        cur.pop_back();
    }
}

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0); cout.tie(0);
    int n, k;
    cin >> n >> k;
    vector<int> current;
    genF(n, k, current, n);
    return 0;
}
```

Mã nguồn Python

```
# ferreries_diagram.py

def max_part(partition):
    return max(partition) if partition else 0

def print_ferrers(partition):
    print("F:")
    pmax = max_part(partition)
    for r in partition:
        print("*" * r + "␣" * (pmax - r) + f"␣{r}")

def print_ferrers_transpose(partition):
    print("FT:")
    pmax = max_part(partition)
    n = len(partition)
    for r in range(pmax):
        row = ''
        for i in range(n):
            row += "*␣" if partition[i] > r else "␣␣"
        print(row.rstrip())
    print('␣'.join(str(x) for x in partition))

def generate_partitions(n, k, current, max_val):
    if k == 0:
        if n == 0:
            print_ferrers(current)
            print_ferrers_transpose(current)
            print("====================")
        return
    for i in range(min(n, max_val), 0, -1):
```

```
        current.append(i)
        generate_partitions(n - i, k - 1, current, i)
        current.pop()

def main():
    n, k = map(int, input().split())
    generate_partitions(n, k, [], n)

if __name__ == "__main__":
    main()
```

Bài toán 14-16: Thuật toán Dijkstra cho các loại đồ thị

Mục đích

Chương trình này triển khai thuật toán Dijkstra để tìm đường đi ngắn nhất từ một đỉnh nguồn đến tất cả các đỉnh khác trong đồ thị có trọng số. Ba phiên bản được phát triển để xử lý các loại đồ thị khác nhau: Simple Graph (Bài 14), Multigraph (Bài 15), và General Graph (Bài 16).

Giới thiệu thuật toán Dijkstra

Thuật toán Dijkstra, được phát triển bởi nhà khoa học máy tính người Hà Lan Edsger W. Dijkstra năm 1956, là một thuật toán tham lam (greedy algorithm) để tìm đường đi ngắn nhất từ một đỉnh nguồn đến tất cả các đỉnh khác trong đồ thị có trọng số không âm.

Ý tưởng cốt lõi:

1. Duy trì một tập hợp các đỉnh đã được xử lý (đã tìm được đường đi ngắn nhất)
2. Tại mỗi bước, chọn đỉnh chưa xử lý có khoảng cách ngắn nhất từ nguồn
3. Cập nhật khoảng cách đến các đỉnh kề thông qua phép "relaxation"
4. Lặp lại cho đến khi tất cả đỉnh được xử lý

Phân loại đồ thị và xử lý đặc biệt

Simple Graph (Đồ thị đơn giản)

- **Định nghĩa:** Không có khuyên (self-loop) và không có cạnh bội (multiple edges)
- **Validation:** Loại bỏ các cạnh (u, u) và cạnh trùng lặp
- **Ứng dụng:** Mạng đường bộ cơ bản, mạng xã hội đơn giản

Multigraph (Đồ thị đa cạnh)

- **Định nghĩa:** Cho phép cạnh bội nhưng không có khuyên
- **Validation:** Chỉ loại bỏ các khuyên (u, u)
- **Ứng dụng:** Mạng giao thông với nhiều tuyến đường giữa hai địa điểm

General Graph (Đồ thị tổng quát)

- **Định nghĩa:** Cho phép cả khuyên và cạnh bội
- **Validation:** Không loại bỏ cạnh nào, xử lý tất cả
- **Ứng dụng:** Mô hình mạng phức tạp, hệ thống với phản hồi

Phân tích chi tiết thuật toán

1. **Khởi tạo:**
   - Đặt khoảng cách từ nguồn đến chính nó bằng 0
   - Đặt khoảng cách từ nguồn đến tất cả đỉnh khác bằng ∞
   - Khởi tạo mảng cha để theo dõi đường đi
2. **Priority Queue:**
   - Sử dụng min-heap để luôn chọn đỉnh có khoảng cách nhỏ nhất
   - Độ phức tạp: $O(\log V)$ cho mỗi thao tác
3. **Relaxation (Cải thiện đường đi):**
   - Với mỗi cạnh (u, v) có trọng số w
   - Nếu $dist[u] + w < dist[v]$ thì cập nhật $dist[v] = dist[u] + w$
   - Cập nhật $parent[v] = u$ để theo dõi đường đi

Cấu trúc dữ liệu và triển khai

Cấu trúc chính (C++)

```
struct DijkstraResult {
    vector<int> distances;  // Shortest distances from source
    vector<int> parents;    // Parent array for path reconstruction
};

// Using priority queue for optimization
```

```
priority_queue<DistanceVertexPair, vector<DistanceVertexPair>,
               greater<DistanceVertexPair>> priorityQueue;
```

Hàm tính đường đi ngắn nhất

```
DijkstraResult calculateShortestPaths(int numberOfVertices,
    const vector<vector<DistanceVertexPair>>& adjacencyList,
    int sourceVertex) {

    vector<int> distances(numberOfVertices, INFINITY_DISTANCE);
    vector<int> parents(numberOfVertices, -1);
    priority_queue<DistanceVertexPair, vector<DistanceVertexPair>,
                   greater<DistanceVertexPair>> priorityQueue;

    distances[sourceVertex] = 0;
    priorityQueue.push({0, sourceVertex});

    while (!priorityQueue.empty()) {
        int currentDistance = priorityQueue.top().first;
        int currentVertex = priorityQueue.top().second;
        priorityQueue.pop();

        if (currentDistance > distances[currentVertex]) continue;

        // Relaxation for all adjacent edges
        for (const auto& edge : adjacencyList[currentVertex]) {
            int neighborVertex = edge.first;
            int edgeWeight = edge.second;

            if (distances[neighborVertex] >
                distances[currentVertex] + edgeWeight) {
                distances[neighborVertex] =
                    distances[currentVertex] + edgeWeight;
                parents[neighborVertex] = currentVertex;
                priorityQueue.push({distances[neighborVertex],
                                    neighborVertex});
            }
        }
    }
    return {distances, parents};
}
```

Validation cho từng loại đồ thị

Simple Graph - Validation nghiêm ngặt

```
bool isValidEdgeForSimpleGraph(int firstVertex, int secondVertex) {
    return firstVertex != secondVertex;  // No self-loops allowed
}

bool doesEdgeExist(const vector<vector<DistanceVertexPair>>& adj,
                   int u, int v) {
    for (const auto& edge : adj[u]) {
        if (edge.first == v) return true;  // Check for multiple edges
    }
    return false;
}
```

Multigraph - Validation trung bình

```
bool isValidEdgeForMultigraph(int firstVertex, int secondVertex) {
    return firstVertex != secondVertex;  // Only remove self-loops
    // Allow multiple edges
}
```

General Graph - Không validation

```
// No special validation required
// Process all edge types including self-loops and multiple edges
```

Tái tạo đường đi (Path Reconstruction)

```
string reconstructPath(const vector<int>& parents,
                       int targetVertex, int sourceVertex) {
    if (parents[targetVertex] == -1 && targetVertex != sourceVertex) {
        return "No path";
    }

    vector<int> path;
    int currentVertex = targetVertex;

    // Traverse backwards from target to source
    while (currentVertex != -1) {
        path.push_back(currentVertex);
        currentVertex = parents[currentVertex];
    }
```

```
    // Reverse to get path from source to target
    reverse(path.begin(), path.end());

    string pathString = "";
    for (int i = 0; i < path.size(); ++i) {
        if (i > 0) pathString += " -> ";
        pathString += to_string(path[i]);
    }
    return pathString;
}
```

Định dạng đầu ra

Tất cả ba phiên bản đều sử dụng định dạng bảng chuyên nghiệp:

```
void displayResults(const DijkstraResult& result, int sourceVertex) {
    cout << "Shortest paths from vertex " << sourceVertex << ":\n";
    cout << "+" << string(8, '-') << "+" << string(12, '-')
         << "+" << string(30, '-') << "+\n";
    cout << "|Vertex|Distance|Path|\n";
    cout << "+" << string(8, '-') << "+" << string(12, '-')
         << "+" << string(30, '-') << "+\n";

    for (int i = 0; i < result.distances.size(); ++i) {
        string distanceStr = (result.distances[i] == INFINITY_DISTANCE)
                             ? "INF" : to_string(result.distances[i]);
        string pathStr = reconstructPath(result.parents, i, sourceVertex);

        if (pathStr.length() > 28) {
            pathStr = pathStr.substr(0, 25) + "...";
        }

        cout << "| " << setw(6) << i << " | " << setw(10) << distanceStr
             << " | " << setw(28) << left << pathStr << " |\n";
    }
    cout << "+" << string(8, '-') << "+" << string(12, '-')
         << "+" << string(30, '-') << "+\n";
}
```

Ví dụ đầu vào/đầu ra

Input file (Simple Graph):

```
5 7
0 0 4    # Self-loop - will be removed
```

```
0 1 1
1 2 2
1 2 3    # Duplicate edge - will be removed
3 4 1
3 4 5    # Duplicate edge - will be removed
3 4 7    # Duplicate edge - will be removed
0
```

Output (Simple Graph):

```
Warning: Self-loops are not allowed in simple graph. Skipping edge (0, 0).
Warning: Multiple edges are not allowed in simple graph. Skipping duplicate edge (1, 2).
Warning: Multiple edges are not allowed in simple graph. Skipping duplicate edge (3, 4).
Warning: Multiple edges are not allowed in simple graph. Skipping duplicate edge (3, 4).

Shortest paths from vertex 0 (Simple Graph - no loops, no multiple edges):
+--------+------------+------------------------------+
| Vertex | Distance   | Path                         |
+--------+------------+------------------------------+
|      0 |          0 | 0                            |
|      1 |          1 | 0 -> 1                       |
|      2 |          3 | 0 -> 1 -> 2                  |
|      3 |        INF | No path                      |
|      4 |        INF | No path                      |
+--------+------------+------------------------------+
```

Độ phức tạp thuật toán

- **Thời gian:** $O((V + E) \log V)$ với V là số đỉnh, E là số cạnh
- **Không gian:** $O(V + E)$ để lưu trữ đồ thị và các mảng phụ trợ
- **Optimization:** Sử dụng priority queue (min-heap) để tối ưu việc chọn đỉnh

So sánh ba phiên bản

Tính năng	Simple Graph	Multigraph	General Graph
Self-loops	Không	Không	Có
Multiple edges	Không	Có	Có
Validation	Nghiêm ngặt	Trung bình	Không
Ứng dụng	Mạng cơ bản	Giao thông	Hệ thống phức tạp

Bảng 1: So sánh các loại đồ thị trong thuật toán Dijkstra

Ưu điểm và hạn chế

Ưu điểm:

- Đảm bảo tìm được đường đi ngắn nhất
- Hiệu quả với đồ thị có trọng số không âm
- Dễ hiểu và triển khai
- Có thể tái tạo đường đi cụ thể

Hạn chế:

- Không xử lý được trọng số âm
- Độ phức tạp cao với đồ thị dày đặc
- Cần lưu trữ toàn bộ đồ thị trong bộ nhớ