# TỔ HỢP VÀ LÝ THUYẾT ĐỒ THỊ

Học kỳ: Mùa hè 2025

Sinh viên thực hiện:

2201700147 - Phan Nguyễn Duy Kha

Giảng viên hướng dẫn: **Nguyễn Quản Bá Hồng**

# FINAL PROJECT

Đồ án cuối kỳ môn học

Khoa Công nghệ
Trường Đại học Quản lý và Công nghệ TP.HCM

# LỊCH SỬ SỬA ĐỔI

| Ngày | Mô tả | Tác giả |
|---|---|---|
| 10/07/2025 | Khởi tạo file báo cáo | Duy Kha |
| 14/07/2025 | Cập nhật Project 5 | Duy Kha |

# Mục lục

# 1 Tổng quan các Project

## 1.1 Project 1: Quy Nạp Toán Học và Quan Hệ Truy Hồi

## 1.2 Project 2: Đếm, Xác Suất, Banh và Hộp

## 1.3 Project: Phân Hoạch Số Nguyên

## 1.4 Project 4: Các Bài Toán Duyệt Đồ Thị và Cây

## 1.5 Project 5: Các Bài Toán Tìm Đường Đi Ngắn Nhất Trên Đồ Thị

# 2 Project 3: Integer Partition – Đồ Án: Phân Hoạch Số Nguyên

## 2.1 Bài toán 1: Biểu đồ Ferrers and biểu đồ Ferrers chuyển vị

**Đề bài**

Nhập $n, k \in \mathbb{N}$. Viết chương trình C/C++, Python để in ra $p_k(n)$ biểu đồ Ferrers $F$ & biểu đồ Ferrers chuyển vị $F^\top$ cho mỗi phân hoạch $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_k) \in (\mathbb{N}^*)^k$ có định dạng các dấu chấm được biểu diễn bởi dấu *.

**Phân tích đề bài:**

- **Input:** Hai số nguyên dương n và k
- **Yêu cầu:** Tìm tất cả các phân hoạch của n thành đúng k phần
- **Output:** Với mỗi phân hoạch, in biểu đồ Ferrers và biểu đồ Ferrers chuyển vị
- **Định dạng:** Sử dụng dấu * để biểu diễn các phần tử trong biểu đồ

**Kiến thức nền tảng:**

**Phân hoạch số nguyên (Integer Partition):**

- Là cách viết số nguyên dương n thành tổng các số nguyên dương
- Ví dụ: 5 = 5 = 4+1 = 3+2 = 3+1+1 = 2+2+1 = 2+1+1+1 = 1+1+1+1+1

**Phân hoạch thành k phần (pk(n)):**

- Là các phân hoạch có đúng k số hạng
- Ký hiệu: $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_k)$ với $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_k \geq 1$ và $\sum_{i=1}^{k} \lambda_i = n$

**Biểu đồ Ferrers:**

- Biểu diễn hình học của phân hoạch bằng các hàng dấu *
- Hàng thứ i có $\lambda_i$ dấu *
- Các hàng sắp xếp từ trên xuống theo thứ tự không tăng

**Biểu đồ Ferrers chuyển vị ($F^T$):**

- Là biểu đồ Ferrers của phân hoạch liên hợp
- Được tạo bằng cách hoán đổi hàng và cột của biểu đồ gốc
- Phần tử thứ j trong phân hoạch chuyển vị = số dấu * ở cột thứ j của biểu đồ gốc

**Công thức và tính chất:**

- **Điều kiện phân hoạch:**

$$\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_k) \text{ với } \lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_k \geq 1$$

$$\sum_{i=1}^{k} \lambda_i = n$$

- **Giới hạn giá trị:** Nếu $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_k)$ thì $\lambda^T = (\mu_1, \mu_2, \ldots, \mu_m)$ với:

$$\mu_j = |\{i : \lambda_i \geq j\}|$$

**Ý tưởng thuật toán:**

**Bước 1: Sinh phân hoạch** $p_k(n)$

1. Sử dụng đệ quy để sinh tất cả các phân hoạch
2. Đảm bảo thứ tự không tăng: $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_k$
3. Giới hạn giá trị của phần tử đầu tiên để đảm bảo tồn tại phân hoạch

**Bước 2: Vẽ biểu đồ Ferrers**

1. Với mỗi phần $\lambda_i$, in ra $\lambda_i$ dấu *
2. Xuống dòng sau mỗi hàng

**Bước 3: Tìm và vẽ biểu đồ chuyển vị**

1. Đếm số dấu * ở mỗi cột của biểu đồ gốc
2. Tạo phân hoạch mới từ các giá trị đếm được
3. Vẽ biểu đồ Ferrers cho phân hoạch mới

**Cài đặt Python:**

- File tổng hợp: `.\code\Project_3\BT1\ferrers_diagrams.py`

**Cài đặt C++:**

- File tổng hợp: `.\code\Project_3\BT1\ferrers_diagrams.cpp`

### 2.2 Bài toán 2:

**Đề bài**

Nhập $n, k \in \mathbb{N}$. Đếm số phân hoạch của $n$ sao cho phần tử lớn nhất là $k$. So sánh $p_k(n)$ & $p_{\max}(n, k)$.

### 2.3 Bài toán 3:

**Đề bài**

Nhập $n, k \in \mathbb{N}$.

(a) Đếm số phân hoạch tự liên hợp của $n$ có $k$ phần, ký hiệu $p_k^{\text{self-conj}}(n)$, rồi in ra các phân hoạch đó.

(b) Đếm số phân hoạch của $n$ có lẻ phần, rồi so sánh với $p_k^{\text{self-conj}}(n)$.

(c) Thiết lập công thức truy hồi cho $p_k^{\text{self-conj}}(n)$, rồi implement bằng:

(i) đệ quy,

(ii) quy hoạch động.

## 3 Project 4: Các Bài Toán Duyệt Đồ Thị và Cây

### 3.1 Bài toán 4: Chuyển đổi giữa các dạng biểu diễn đồ thị và cây

**Đề bài**

**Viết chương trình C/C++, Python chuyển đổi giữa 4 dạng biểu diễn: adjacency matrix, adjacency list, extended adjacency list, adjacency map cho 3 đồ thị: đơn đồ thị, đa đồ thị, đồ thị tổng quát; & 3 dạng biểu diễn: array of parents, first-child next-sibling, graph-based representation of trees của cây.**
Sẽ có $3 \times A_4^3 + A_2^3 = 36 + 6 = 42$ converter programs.

**Phân tích bài toán:** Bài toán yêu cầu implement tổng cộng 42 converter functions, được chia thành:

- **Đồ thị:** $3 \times 4 \times 3 = 36$ converters (3 loại đồ thị × 4 dạng biểu diễn × 3 chuyển đổi mỗi dạng)
- **Cây:** $3 \times 2 = 6$ converters (3 dạng biểu diễn × 2 chuyển đổi mỗi dạng)

**Cài đặt Python:**

- File: `.\code\Project_4\BT4\simple_graph\simple_graph.py`
- File: `.\code\Project_4\BT4\multi_graph\multi_graph.py`
- File: `.\code\Project_4\BT4\general_graph\general_graph.py`
- File: `.\code\Project_4\BT4\\tree\tree.py`

**Cài đặt C++:**

- File: `.\code\Project_4\BT4\simple_graph\simple_graphy.cpp`

- File: `.\code\Project_4\BT4\multi_graph\multi_graph.cpp`
- File: `.\code\Project_4\BT4\general_graph\general_graph.cpp`
- File: `.\code\Project_4\BT4\tree\tree.cpp`

### 3.1.1 Simple Graph (Đơn đồ thị) - 12 Converters

**Định nghĩa:** Simple Graph $G = (V, E)$ là đồ thị không có cạnh song song và không có khuyên (self-loops).

**4 dạng biểu diễn:**

- 1. Adjacency Matrix (Ma trận kề)
- 2. Adjacency List (Danh sách kề)
- 3. Extended Adjacency List
- 4. Adjacency Map

**Converter 1: Matrix → List**
**Ý tưởng chính:** Duyệt upper triangle của ma trận đối xứng để tránh xử lý duplicate edges.
**Thuật toán:**

1. Khởi tạo $n$ danh sách rỗng cho adjacency list
2. Chỉ duyệt các vị trí $(i, j)$ với $i \leq j$ (upper triangle)
3. Nếu $matrix[i][j] = 1$:
    - Nếu $i = j$: thêm $j$ vào $adj\_list[i]$ (self-loop)
    - Nếu $i \neq j$: thêm $j$ vào $adj\_list[i]$ và $i$ vào $adj\_list[j]$ (undirected)

**Converter 2: Matrix → Extended List**
**Ý tưởng chính:** Tương tự Matrix → List nhưng tạo Edge objects thay vì chỉ lưu đỉnh.
**Thuật toán:**

1. Duyệt upper triangle của ma trận
2. Với mỗi cạnh $(i, j)$ tìm được:
    - Tạo $Edge(i, j)$ và thêm vào $outgoing[i]$, $incoming[j]$
    - Nếu $i \neq j$: tạo $Edge(j, i)$ và thêm vào $outgoing[j]$, $incoming[i]$
    - Thêm cả 2 edges vào $all\_edges$

**Converter 3: Matrix → Map**
**Ý tưởng chính:** Duyệt toàn bộ ma trận và thêm neighbors vào hash set.
**Thuật toán:**

1. Khởi tạo $n$ hash sets rỗng
2. Duyệt toàn bộ ma trận $(i, j)$ với $0 \leq i, j < n$
3. Nếu $matrix[i][j] = 1$: thêm $j$ vào $adj\_map[i]$

**Converter 4: List → Matrix**
**Ý tưởng chính:** Duyệt adjacency list và đánh dấu các vị trí tương ứng trong ma trận.
**Thuật toán:**

1. Khởi tạo ma trận $n \times n$ toàn $0$
2. Với mỗi đỉnh $u = 0, 1, \ldots, n-1$:
3. Với mỗi neighbor $v \in adj\_list[u]$: đặt $matrix[u][v] = 1$

**Converter 5: List → Extended List**
**Ý tưởng chính:** Tránh tạo duplicate Edge objects bằng canonical form technique.
**Thuật toán:**

1. Khởi tạo $processed\_edges = \emptyset$ (hash set)
2. Với mỗi đỉnh $u$ và neighbor $v \in adj\_list[u]$:
   - Tính $edge\_key = (\min(u, v), \max(u, v))$
   - Nếu $edge\_key \notin processed\_edges$:
     - Thêm $edge\_key$ vào $processed\_edges$
     - Tạo $Edge(u, v)$ và $Edge(v, u)$
     - Cập nhật các cấu trúc dữ liệu

**Converter 6: List → Map**
**Ý tưởng chính:** Chuyển đổi trực tiếp từ list sang set cho mỗi đỉnh.
**Thuật toán:**

1. Với mỗi đỉnh $u = 0, 1, \ldots, n-1$:
2. Chuyển $adj\_list[u]$ (list) thành $adj\_map[u]$ (set)

**Converter 7: Extended List → Matrix**
**Ý tưởng chính:** Đơn giản nhất - duyệt tất cả edges và đánh dấu ma trận.
**Thuật toán:**

1. Khởi tạo ma trận $n \times n$ toàn $0$
2. Với mỗi $edge \in all\_edges$:
3. Đặt $matrix[edge.source][edge.target] = 1$

**Converter 8: Extended List → List**
**Ý tưởng chính:** Trích xuất target vertices từ outgoing edges.
**Thuật toán:**

1. Với mỗi đỉnh $u = 0, 1, \ldots, n-1$:
2. Với mỗi $edge \in outgoing[u]$:

3. Thêm $edge.target$ vào $adj\_list[u]$ (nếu chưa có)

**Converter 9: Extended List → Map**
**Ý tưởng chính:** Tương tự converter 8 nhưng sử dụng set thay vì list.
**Thuật toán:**

1. Với mỗi đỉnh $u = 0, 1, \ldots, n-1$:
2. Với mỗi $edge \in outgoing[u]$:
3. Thêm $edge.target$ vào $adj\_map[u]$ (set tự động loại duplicate)

**Converter 10: Map → Matrix**
**Ý tưởng chính:** Duyệt hash sets và đánh dấu ma trận.
**Thuật toán:**

1. Khởi tạo ma trận $n \times n$ toàn $0$
2. Với mỗi đỉnh $u = 0, 1, \ldots, n-1$:
3. Với mỗi $v \in adj\_map[u]$: đặt $matrix[u][v] = 1$

**Converter 11: Map → List**
**Ý tưởng chính:** Chuyển đổi trực tiếp từ set sang list cho mỗi đỉnh.
**Thuật toán:**

1. Với mỗi đỉnh $u = 0, 1, \ldots, n-1$:
2. Chuyển $adj\_map[u]$ (set) thành $adj\_list[u]$ (list)

**Converter 12: Map → Extended List**
**Ý tưởng chính:** Chuyển đổi gián tiếp qua List để tái sử dụng algorithm đã có.
**Thuật toán:**

1. Gọi $list\_graph =$ Map → List$(map_graph)$
2. Gọi $ext\_graph =$ List → Extended List$(list_graph)$
3. Trả về $ext\_graph$

#### 3.1.2 Multigraph (Đa đồ thị) - 12 Converters

**Định nghĩa:** Multigraph $G = (V, E)$ là đồ thị cho phép có nhiều cạnh song song giữa hai đỉnh (parallel edges) nhưng không có khuyên (self-loops).

**4 dạng biểu diễn:**

- 1. Adjacency Matrix (Ma trận kề): $matrix[i][j] \in \{0, 1, 2, 3, \ldots\}$ - số cạnh giữa đỉnh $i$ và $j$
- 2. Adjacency List (Danh sách kề): cho phép entries trùng lặp

- 3. Extended Adjacency List: nhiều Edge objects cho cùng một cặp đỉnh
- 4. Adjacency Map: $adj\_map[u][v] = count$ - số cạnh từ $u$ đến $v$

**Converter 1: Matrix → List**
**Ý tưởng chính:** Duyệt upper triangle và thêm multiple entries cho parallel edges.
**Thuật toán:**

1. Khởi tạo $n$ danh sách rỗng cho adjacency list
2. Chỉ duyệt các vị trí $(i, j)$ với $i \leq j$ (upper triangle)
3. Nếu $matrix[i][j] = count > 0$:
    - Nếu $i = j$: thêm $j$ vào $adj\_list[i]$ đúng $count$ lần (self-loop)
    - Nếu $i \neq j$: thêm $j$ vào $adj\_list[i]$ và $i$ vào $adj\_list[j]$ mỗi cái $count$ lần

**Converter 2: List → Matrix**
**Ý tưởng chính:** Đếm frequency của mỗi neighbor thay vì chỉ đánh dấu 0/1.
**Thuật toán:**

1. Khởi tạo ma trận $n \times n$ toàn $0$
2. Với mỗi đỉnh $u = 0, 1, \ldots, n-1$:
    - Tạo $neighbor\_count = \emptyset$ (hash map)
    - Với mỗi $v \in adj\_list[u]$: tăng $neighbor\_count[v]$
    - Với mỗi $(v, count) \in neighbor\_count$: đặt $matrix[u][v] = count$

**Converter 3: Matrix → Extended List**
**Ý tưởng chính:** Tạo multiple Edge objects cho parallel edges, mỗi edge có unique ID.
**Thuật toán:**

1. Duyệt upper triangle của ma trận
2. Với mỗi cạnh $(i, j)$ có $matrix[i][j] = count > 0$:
    - Lặp $count$ lần:
        - Tạo $Edge(i, j, weight, unique\_id)$
        - Thêm vào $outgoing[i]$, $incoming[j]$, $all\_edges$
        - Nếu $i \neq j$: tạo $Edge(j, i)$ tương tự

**Converter 4: Extended List → Matrix**
**Ý tưởng chính:** Đếm frequency của mỗi $(source, target)$ pair từ $all\_edges$.
**Thuật toán:**

1. Khởi tạo ma trận $n \times n$ toàn $0$
2. Tạo $edge\_count = \emptyset$ (hash map)

3. Với mỗi $edge \in all\_edges$: tăng $edge\_count[(edge.source, edge.target)]$
4. Với mỗi $((u, v), count) \in edge\_count$: đặt $matrix[u][v] = count$

**Converter 5: List → Extended List**
**Ý tưởng chính:** Sử dụng canonical form để tránh duplicate và xử lý đúng undirected edges.
**Thuật toán:**

1. Tạo $edge\_count = \emptyset$ (hash map)
2. Với mỗi đỉnh $u$ và neighbor $v \in adj\_list[u]$:
   - Tính $canonical\_key = (\min(u, v), \max(u, v))$
   - Tăng $edge\_count[canonical\_key]$
3. Với mỗi $((u, v), total\_count) \in edge\_count$:
   - $actual\_count =$ nếu $u = v$ thì $total\_count$ ngược lại $total\_count/2$
   - Tạo $actual\_count$ Edge objects cho cặp $(u, v)$

**Converter 6: Extended List → List**
**Ý tưởng chính:** Trích xuất target vertices từ outgoing edges (bao gồm duplicates).
**Thuật toán:**

1. Với mỗi đỉnh $u = 0, 1, \ldots, n - 1$:
2. Với mỗi $edge \in outgoing[u]$:
3. Thêm $edge.target$ vào $adj\_list[u]$ (bao gồm cả duplicates)

**Converter 7: Matrix → Map**
**Ý tưởng chính:** Duyệt toàn bộ ma trận và lưu non-zero entries vào hash map.
**Thuật toán:**

1. Khởi tạo $n$ hash maps rỗng
2. Duyệt toàn bộ ma trận $(i, j)$ với $0 \leq i, j < n$
3. Nếu $matrix[i][j] = count > 0$: đặt $adj\_map[i][j] = count$

**Converter 8: Map → Matrix**
**Ý tưởng chính:** Duyệt hash maps và sao chép counts vào ma trận.
**Thuật toán:**

1. Khởi tạo ma trận $n \times n$ toàn $0$
2. Với mỗi đỉnh $u = 0, 1, \ldots, n - 1$:
3. Với mỗi $(v, count) \in adj\_map[u]$: đặt $matrix[u][v] = count$

**Converter 9: List → Map**
**Ý tưởng chính:** Đếm frequency của neighbors và lưu vào hash map.
**Thuật toán:**

1. Với mỗi đỉnh $u = 0, 1, \ldots, n-1$:
   - Tạo $neighbor\_count = \emptyset$ (hash map)
   - Với mỗi $v \in adj\_list[u]$: tăng $neighbor\_count[v]$
   - Đặt $adj\_map[u] = neighbor\_count$

**Converter 10: Map → List**
**Ý tưởng chính:** Mở rộng mỗi entry $(v, count)$ thành $count$ duplicates trong list.
**Thuật toán:**

1. Với mỗi đỉnh $u = 0, 1, \ldots, n-1$:
2. Với mỗi $(v, count) \in adj\_map[u]$:
3. Thêm $v$ vào $adj\_list[u]$ đúng $count$ lần

**Converter 11: Extended List → Map**
**Ý tưởng chính:** Đếm frequency của targets từ outgoing edges.
**Thuật toán:**

1. Với mỗi đỉnh $u = 0, 1, \ldots, n-1$:
   - Tạo $target\_count = \emptyset$ (hash map)
   - Với mỗi $edge \in outgoing[u]$: tăng $target\_count[edge.target]$
   - Đặt $adj\_map[u] = target\_count$

**Converter 12: Map → Extended List**
**Ý tưởng chính:** Sử dụng canonical form để tránh duplicate edges và tạo multiple Edge objects.
**Thuật toán:**

1. Tạo $processed\_edges = \emptyset$ (hash set)
2. Với mỗi đỉnh $u$ và $(v, count) \in adj\_map[u]$:
   - Tính $canonical\_key = (\min(u, v), \max(u, v))$
   - Nếu $canonical\_key \notin processed\_edges$:
     - Thêm $canonical\_key$ vào $processed\_edges$
     - Lặp $count$ lần: tạo $Edge(u, v)$ và $Edge(v, u)$ (nếu $u \neq v$)
     - Cập nhật các cấu trúc dữ liệu

#### 3.1.3 General Graph (Đồ thị tổng quát) - 12 Converters

**Định nghĩa:** General Graph $G = (V, E)$ là đồ thị cho phép có cả cạnh song song (parallel edges) và khuyên (self-loops).

**Converter 1: Matrix → List**
**Ý tưởng chính:** Duyệt toàn bộ ma trận (không chỉ upper triangle) để xử lý self-loops riêng biệt.
**Thuật toán:**

1. Khởi tạo $n$ danh sách rỗng cho adjacency list
2. Duyệt toàn bộ ma trận $(i, j)$ với $0 \leq i, j < n$
3. Nếu $matrix[i][j] = count > 0$:
   - Nếu $i = j$: thêm $j$ vào $adj\_list[i]$ đúng $count$ lần (self-loop)
   - Nếu $i \neq j$ và $i < j$: thêm $j$ vào $adj\_list[i]$ và $i$ vào $adj\_list[j]$ mỗi cái $count$ lần (undirected)

**Converter 2: List → Matrix**
**Ý tưởng chính:** Đếm frequency của mỗi neighbor, xử lý self-loops đặc biệt.
**Thuật toán:**

1. Khởi tạo ma trận $n \times n$ toàn 0
2. Với mỗi đỉnh $u = 0, 1, \ldots, n-1$:
   - Tạo $neighbor\_count = \emptyset$ (hash map)
   - Với mỗi $v \in adj\_list[u]$: tăng $neighbor\_count[v]$
   - Với mỗi $(v, count) \in neighbor\_count$:
     - Nếu $u = v$: đặt $matrix[u][v] = count$ (self-loop)
     - Nếu $u \neq v$: đặt $matrix[u][v] = count$ (undirected edge sẽ được xử lý từ cả 2 phía)

**Converter 3: Matrix → Extended List**
**Ý tưởng chính:** Tạo Edge objects với unique IDs, xử lý self-loops và normal edges riêng biệt.
**Thuật toán:**

1. Duyệt toàn bộ ma trận $(i, j)$ với $0 \leq i, j < n$
2. Với mỗi entry có $matrix[i][j] = count > 0$:
   - Nếu $i = j$ (self-loop):
     - Lặp $count$ lần: tạo $Edge(i, j, weight, unique\_id)$
     - Thêm vào $outgoing[i]$, $incoming[j]$, $all\_edges$
     - **Chú ý:** Chỉ tạo 1 edge object cho mỗi self-loop

- Nếu $i \neq j$ và $i < j$ (normal edge):
  - Lặp $count$ lần: tạo $Edge(i, j)$ và $Edge(j, i)$
  - Cập nhật các cấu trúc dữ liệu tương ứng

**Converter 4: Extended List → Matrix**
**Ý tưởng chính:** Đếm frequency của mỗi $(source, target)$ pair từ $all\_edges$.
**Thuật toán:**

1. Khởi tạo ma trận $n \times n$ toàn $0$
2. Tạo $edge\_count = \emptyset$ (hash map)
3. Với mỗi $edge \in all\_edges$: tăng $edge\_count[(edge.source, edge.target)]$
4. Với mỗi $((u, v), count) \in edge\_count$: đặt $matrix[u][v] = count$

**Converter 5: List → Extended List**
**Ý tưởng chính:** Xử lý self-loops trước, sau đó xử lý normal edges với canonical form.
**Thuật toán:**

1. **Xử lý self-loops trước:**
   - Với mỗi đỉnh $u$: đếm $self\_loop\_count = |\{v \in adj\_list[u] : v = u\}|$
   - Nếu $self\_loop\_count > 0$: tạo $self\_loop\_count$ Edge objects cho $(u, u)$
2. **Xử lý normal edges với canonical form:**
   - Tạo $edge\_count = \emptyset$ (hash map)
   - Với mỗi đỉnh $u$ và neighbor $v \in adj\_list[u]$ với $u \neq v$:
     - Tính $canonical\_key = (\min(u, v), \max(u, v))$
     - Tăng $edge\_count[canonical\_key]$
   - Với mỗi $((u, v), total\_count) \in edge\_count$:
     - $actual\_count = total\_count/2$ (undirected edges được đếm 2 lần)
     - Tạo $actual\_count$ cặp Edge objects: $Edge(u, v)$ và $Edge(v, u)$

**Converter 6: Extended List → List**
**Ý tưởng chính:** Trích xuất target vertices từ outgoing edges (bao gồm cả self-loops).
**Thuật toán:**

1. Với mỗi đỉnh $u = 0, 1, \ldots, n - 1$:
2. Với mỗi $edge \in outgoing[u]$:
3. Thêm $edge.target$ vào $adj\_list[u]$ (bao gồm cả self-loops và duplicates)

**Converter 7: Matrix → Map**
**Ý tưởng chính:** Duyệt toàn bộ ma trận và lưu non-zero entries vào hash map.
**Thuật toán:**

1. Khởi tạo $n$ hash maps rỗng
2. Duyệt toàn bộ ma trận $(i, j)$ với $0 \leq i, j < n$
3. Nếu $matrix[i][j] = count > 0$: đặt $adj\_map[i][j] = count$

**Converter 8: Map → Matrix**
**Ý tưởng chính:** Duyệt hash maps và sao chép counts vào ma trận.
**Thuật toán:**

1. Khởi tạo ma trận $n \times n$ toàn $0$
2. Với mỗi đỉnh $u = 0, 1, \ldots, n-1$:
3. Với mỗi $(v, count) \in adj\_map[u]$: đặt $matrix[u][v] = count$

**Converter 9: List → Map**
**Ý tưởng chính:** Đếm frequency của neighbors (bao gồm self-loops) và lưu vào hash map.
**Thuật toán:**

1. Với mỗi đỉnh $u = 0, 1, \ldots, n-1$:
   - Tạo $neighbor\_count = \emptyset$ (hash map)
   - Với mỗi $v \in adj\_list[u]$: tăng $neighbor\_count[v]$ (bao gồm $v = u$ cho self-loops)
   - Đặt $adj\_map[u] = neighbor\_count$

**Converter 10: Map → List**
**Ý tưởng chính:** Mở rộng mỗi entry $(v, count)$ thành $count$ duplicates trong list.
**Thuật toán:**

1. Với mỗi đỉnh $u = 0, 1, \ldots, n-1$:
2. Với mỗi $(v, count) \in adj\_map[u]$:
3. Thêm $v$ vào $adj\_list[u]$ đúng $count$ lần (bao gồm self-loops khi $v = u$)

**Converter 11: Extended List → Map**
**Ý tưởng chính:** Đếm frequency của targets từ outgoing edges (bao gồm self-loops).
**Thuật toán:**

1. Với mỗi đỉnh $u = 0, 1, \ldots, n-1$:
   - Tạo $target\_count = \emptyset$ (hash map)
   - Với mỗi $edge \in outgoing[u]$: tăng $target\_count[edge.target]$
   - Đặt $adj\_map[u] = target\_count$

**Converter 12: Map → Extended List**

**Ý tưởng chính:** Sử dụng canonical form để tránh duplicate edges, xử lý self-loops riêng biệt.

**Thuật toán:**

1. Tạo $processed\_edges = \emptyset$ (hash set)
2. Với mỗi đỉnh $u$ và $(v, count) \in adj\_map[u]$:
   - Nếu $u = v$ (self-loop):
     - Lặp $count$ lần: tạo $Edge(u, u, weight, unique\_id)$
     - Cập nhật $outgoing[u]$, $incoming[u]$, $all\_edges$
   - Nếu $u \neq v$ (normal edge):
     - Tính $canonical\_key = (\min(u, v), \max(u, v))$
     - Nếu $canonical\_key \notin processed\_edges$:
       * Thêm $canonical\_key$ vào $processed\_edges$
       * Lặp $count$ lần: tạo $Edge(u, v)$ và $Edge(v, u)$
       * Cập nhật các cấu trúc dữ liệu

#### 3.1.4 Tree Representations (Biểu diễn cây) - 6 Converters

**Định nghĩa:** Tree $T = (V, E)$ là đồ thị liên thông không có chu trình với $|E| = |V| - 1$, có một đỉnh đặc biệt gọi là root.

**3 dạng biểu diễn cây:**

- 1. Array of Parents: Mảng $P[v] = parent[v]$ hoặc $nil$ nếu $v$ là root
- 2. First-Child Next-Sibling: Cặp mảng $(F, N)$ với first-child và next-sibling links
- 3. Graph-Based Representation: Sử dụng Extended Adjacency List hoặc Adjacency Map

**Array of Parents Representation**

**Định nghĩa:** Let $T = (V, E)$ be a tree with $n$ nodes. Array-of-parents representation là mảng $P$ với $n$ phần tử, indexed theo nodes của tree.

$$P[v] = \begin{cases} parent[v] & \text{if } v \neq root[T] \\ nil & \text{if } v = root[T] \end{cases} \quad \forall v \in V$$

**First-Child Next-Sibling Representation**

**Định nghĩa:** Let $T = (V, E)$ be a tree with $n$ nodes. First-child next-sibling representation là cặp $(F, N)$ với hai mảng $n$ phần tử.

$$F[v] = \begin{cases} first[v] & \text{if } v \text{ is not a leaf} \\ nil & \text{if } v \text{ is a leaf} \end{cases}$$

$$N[v] = \begin{cases} next[v] & \text{if } v \text{ is not a last child} \\ nil & \text{if } v \text{ is a last child} \end{cases}$$

**Graph-Based Representation of Trees**

**Định nghĩa:** Trees được biểu diễn sử dụng abstract operations trên graphs, implement bằng Extended Adjacency List hoặc Adjacency Map.

**Converter 1: Array of Parents → First-Child Next-Sibling**

**Ý tưởng chính:** Xây dựng children lists từ parent array, sau đó tạo first-child và next-sibling links.
**Thuật toán:**

1. Khởi tạo mảng $F$ và $N$ toàn $nil$
2. Tạo $children\_lists[u] = \emptyset$ cho mọi $u \in \{0, 1, \ldots, n-1\}$
3. Với mỗi node $v$ có $P[v] \neq nil$:
   - $parent \leftarrow P[v]$
   - Thêm $v$ vào $children\_lists[parent]$
4. Với mỗi node $u$ có $children\_lists[u] \neq \emptyset$:
   - Sắp xếp $children\_lists[u]$ theo thứ tự tăng dần
   - $F[u] \leftarrow children\_lists[u][0]$ (first child)
   - Với $i = 0, 1, \ldots, |children\_lists[u]| - 2$:
     - $N[children\_lists[u][i]] \leftarrow children\_lists[u][i+1]$
   - $N[children\_lists[u][-1]] \leftarrow nil$ (last child)

**Converter 2: First-Child Next-Sibling → Array of Parents**

**Ý tưởng chính:** DFS traversal từ root để xác định parent cho mỗi node.
**Thuật toán:**

1. Khởi tạo mảng $P$ toàn $nil$
2. Tìm root node $r$ (có thể tìm bằng cách check node nào không là child của node khác)
3. Gọi DFS từ root:
   - **procedure** DFS$(node, parent)$:
     - $P[node] \leftarrow parent$
     - $child \leftarrow F[node]$
     - **while** $child \neq nil$ **do**:
       - DFS$(child, node)$
       - $child \leftarrow N[child]$
4. Gọi DFS$(r, nil)$

**Converter 3: Array of Parents → Graph-Based**

**Ý tưởng chính:** Tạo directed edges từ parent đến children trong graph representation.
**Thuật toán:**

1. Khởi tạo graph $G$ với $n$ vertices
2. Với mỗi node $v$ có $P[v] \neq nil$:
   - $parent \leftarrow P[v]$
   - Thêm directed edge $(parent, v)$ vào $G$
3. Lưu root reference: node $r$ có $P[r] = nil$

**Converter 4: Graph-Based → Array of Parents**

**Ý tưởng chính:** Trích xuất parent relationships từ incoming edges của graph.
**Thuật toán:**

1. Khởi tạo mảng $P$ toàn $nil$
2. Với mỗi node $v$ trong graph:
   - Nếu $|incoming[v]| = 0$: $P[v] \leftarrow nil$ (root node)
   - Ngược lại: $P[v] \leftarrow source(incoming[v][0])$ (parent)

**Converter 5: First-Child Next-Sibling → Graph-Based**

**Ý tưởng chính:** DFS traversal từ root để xây dựng directed edges trong graph.
**Thuật toán:**

1. Khởi tạo graph $G$ với $n$ vertices
2. Tìm root node $r$
3. Gọi DFS từ root:
   - **procedure** DFS$(node, parent)$:
     - **if** $parent \neq nil$ **then** thêm edge $(parent, node)$ vào $G$
     - $child \leftarrow F[node]$
     - **while** $child \neq nil$ **do**:
       * DFS$(child, node)$
       * $child \leftarrow N[child]$
4. Gọi DFS$(r, nil)$

**Converter 6: Graph-Based → First-Child Next-Sibling**

**Ý tưởng chính:** Sử dụng outgoing edges để xây dựng first-child và next-sibling links.
**Thuật toán:**

1. Khởi tạo mảng $F$ và $N$ toàn $nil$
2. Với mỗi node $u$ trong graph:
   - Lấy $children \leftarrow [target(e) : e \in outgoing[u]]$
   - Sắp xếp $children$ theo thứ tự tăng dần
   - Nếu $children \neq \emptyset$:
     - $F[u] \leftarrow children[0]$ (first child)
     - Với $i = 0, 1, \ldots, |children| - 2$:
       - $N[children[i]] \leftarrow children[i + 1]$
     - $N[children[-1]] \leftarrow nil$ (last child)

### 3.2 Bài toán 5: Làm Problems 1.1 - 1.6 và Exercises 1.1 - 1.10 [Val21, pp. 39-40]

**Đề bài**

**Làm Problems 1.1–1.6 và Exercises 1.1–1.10 [Val21, pp. 39–40]**

**Cài đặt Python:**

- File: `.\code\Project_4\BT5\python`

**Cài đặt C++:**

- File: `.\code\Project_4\BT5\c++`

### 3.3 Bài toán 6: Tree Edit Distance

**Đề bài**

**Viết chương trình C/C++, Python để giải bài toán tree edit distance problem bằng cách sử dụng: (a) Backtracking. (b) Branch bound. (c) Divide conquer. (d) Dynamic programming.**

**Phân tích bài toán:** Tìm khoảng cách nhỏ nhất để biến đổi cây T1 thành cây T2 bằng 3 phép toán cơ bản: chèn nút (insert), xóa nút (delete), và đổi nhãn nút (relabel).

**Các loại thuật toán:**

- **Backtracking:** Duyệt tất cả các khả năng biến đổi, sử dụng đệ quy để thử tất cả các phép toán. Có thể cắt tỉa nhánh khi chi phí hiện tại vượt quá chi phí tốt nhất đã tìm thấy.
- **Branch and Bound:** Cải tiến của backtracking bằng cách sử dụng cận trên và cận dưới để loại bỏ các nhánh không tiềm năng, giúp giảm không gian tìm kiếm.
- **Divide and Conquer:** Chia bài toán lớn thành các bài toán con nhỏ hơn, giải quyết độc lập từng phần rồi kết hợp kết quả. Phù hợp với tư duy đệ quy nhưng có thể lặp lại tính toán.
- **Dynamic Programming:** Sử dụng bảng lưu trữ kết quả các bài toán con để tránh tính toán lặp lại. Là phương pháp tối ưu nhất với độ phức tạp đa thức khi được cài đặt đúng cách.

**Cài đặt Python:**

- File: `.\code\Project_4\BT6\python`

**Cài đặt C++:**

- File: `.\code\Project_4\BT6\c++`

### 3.4 Bài toán 7: Tree traversal – Duyệt cây

**Đề bài**

**Viết chương trình C/C++, Python để duyệt cây: (a) preorder traversal. (b) postorder traversal. (c) top-down traveral. (d) bottom-up traversal.**

**Phân tích bài toán:**

- Duyệt cây nghĩa là bạn phải ghé thăm tất cả các nút trong cây. Ví dụ, bạn có thể muốn cộng tất cả các giá trị trong cây hoặc tìm giá trị lớn nhất. Với tất cả các thao tác này, bạn sẽ cần ghé thăm từng nút của cây. (Theo programiz)
- Các cấu trúc dữ liệu tuyến tính như mảng, ngăn xếp , hàng đợi và danh sách liên kết chỉ có một cách để đọc dữ liệu. Nhưng một cấu trúc dữ liệu phân cấp như cây có thể được duyệt theo nhiều cách khác nhau

**Các loại duyệt cây:**

- **Preorder traversal:** Duyệt theo thứ tự gốc - trái - phải. Thích hợp để sao chép cây hoặc biểu diễn tiền tố.
- **Postorder traversal:** Duyệt theo thứ tự trái - phải - gốc. Thường dùng để tính toán biểu thức hoặc xóa cây.
- **Top-down traveral:** Duyệt theo cấp độ từ trên xuống dưới, trái sang phải (còn gọi là level-order traversal). Sử dụng hàng đợi để thực hiện.
- **Bottom-up traversal:** Duyệt theo cấp độ từ dưới lên trên, trái sang phải. Thực hiện giống level-order nhưng kết quả được đảo ngược.

**Cài đặt Python:**

- File: `.\code\Project_4\BT7\python`
- File tổng hợp: `.\code\Project_4\BT7\tree_traversal.py`

**Cài đặt C++:**

- File: `.\code\Project_4\BT7\c++`
- File tổng hợp: `.\code\Project_4\BT7\tree_traversal.cpp`

### 3.5 Bài toán 8: BFS trên đồ thị đơn (Simple Graph)

**Đề bài**

**Let G = (V, E) be a finite simple graph. Implement the breadth-first search on G.**

**Phân tích bài toán:**

**Thuật toán BFS:**

- Là thuật toán duyệt đồ thị theo chiều rộng, bắt đầu từ một đỉnh nguồn $s \in V$.
- Khám phá lần lượt tất cả các đỉnh kề của đỉnh hiện tại trước khi đi sâu hơn.
- Sử dụng cấu trúc dữ liệu hàng đợi (queue) để lưu trữ các đỉnh đang chờ duyệt.
- Đảm bảo rằng mỗi đỉnh chỉ được thăm một lần nhờ mảng đánh dấu `visited`.

**Đặc điểm đồ thị đơn (Simple graph):**

- Đồ thị vô hướng hoặc có hướng không chứa đa cạnh (tức mỗi cặp đỉnh chỉ có tối đa 1 cạnh) và không có khuyên (self-loop).
- Danh sách kề được lưu trữ dưới dạng mảng vector hoặc list.

**Input:**

- Đồ thị $G = (V, E)$, biểu diễn bằng danh sách kề.
- Đỉnh nguồn $s \in V$.

**Output:**

- Thứ tự các đỉnh được thăm theo BFS.
- Mảng khoảng cách từ đỉnh $s$ đến các đỉnh còn lại.

**Cài đặt Python:**

- File: `.\code\Project_4\BT8\bfs_simple_graph.py`

**Cài đặt C++:**

- File: `.\code\Project_4\BT8\bfs_simple_graph.cpp`

### 3.6 Bài toán 9: BFS trên đồ thị đa (Multigraph)

**Đề bài**

**Let G = (V, E) be a finite multigraph. Implement the breadth-first search on G.**

**Phân tích:**

- Multigraph cho phép đa cạnh giữa hai đỉnh, tức một cặp đỉnh có thể nối với nhau bởi nhiều hơn một cạnh.
- Có thể có khuyên (self-loop).
- BFS vẫn hoạt động như trên đồ thị đơn, nhờ mảng `visited` đảm bảo không thăm lại đỉnh.
- Đa cạnh và khuyên chỉ làm danh sách kề có thể lặp lại phần tử, không làm thay đổi logic duyệt.

**Cài đặt Python:**

- File: `.\code\Project_4\BT9\bfs_multigraph.py`

**Cài đặt C++:**

- File: `.\code\Project_4\BT9\bfs_multigraph.cpp`

### 3.7 Bài toán 10: BFS trên đồ thị tổng quát (General Graph)

**Đề bài**

**Let G = (V, E) be a general graph. Implement the breadth-first search on G.**

**Phân tích:**

- Đồ thị tổng quát có thể không liên thông, gồm nhiều thành phần rời.
- Gồm đa cạnh, khuyên và có thể có hướng hoặc vô hướng.
- Thuật toán BFS cần chạy lặp lại trên tất cả các đỉnh chưa thăm để đảm bảo duyệt hết mọi thành phần.

**Cài đặt Python:**

- File: `.\code\Project_4\BT10\bfs_general_graph.py`

**Cài đặt C++:**

- File: `.\code\Project_4\BT10\bfs_general_graph.cpp`

### 3.8 Bài toán 11: DFS trên đồ thị đơn (Simple Graph)

**Đề bài**

**Let G = (V, E) be a finite simple graph. Implement the depth-first search on G.**

**Phân tích bài toán:**

- Thuật toán duyệt theo chiều sâu, đi sâu vào từng nhánh của đồ thị trước khi quay lui.
- Thường sử dụng đệ quy hoặc stack để duyệt.
- Đảm bảo thăm mỗi đỉnh một lần nhờ mảng `visited`.
- Thích hợp để tìm đường đi, kiểm tra liên thông, hoặc phân tích cấu trúc đồ thị.

**Cài đặt Python:**

- File: `.\code\Project_4\BT11\dfs_simple_graph.py`

**Cài đặt C++:**

- File: `.\code\Project_4\BT11\dfs_simple_graph.cpp`

### 3.9 Bài toán 12: DFS trên đồ thị đa (Multigraph)

**Đề bài**

**Let G = (V, E) be a finite multigraph. Implement the depth-first search on G.**

**Phân tích:**

- Đồ thị đa cạnh, có thể có khuyên.
- DFS vẫn giữ nguyên nguyên lý: duyệt sâu, không thăm lại đỉnh.
- Mảng `visited` ngăn vòng lặp do đa cạnh hoặc khuyên.

**Cài đặt Python:**

- File: `.\code\Project_4\BT12\dfs_multigraph.py`

**Cài đặt C++:**

- File: `.\code\Project_4\BT12\dfs_multigraph.cpp`

### 3.10 Bài toán 13: DFS trên đồ thị tổng quát (General Graph)

**Đề bài**

**Let G = (V, E) be a finite general graph. Implement the depth-first search on G.**

**Phân tích:**

- Đồ thị tổng quát gồm nhiều thành phần rời, đa cạnh, khuyên.
- Cần chạy DFS nhiều lần trên các đỉnh chưa thăm để duyệt hết mọi thành phần.
- Tạo ra rừng DFS (DFS forest).

**Cài đặt Python:**

- File: `.\code\Project_4\BT13\dfs_general_graph.py`

**Cài đặt C++:**

- File: `.\code\Project_4\BT13\dfs_general_graph.cpp`

# 4 Project 5: Các Bài Toán Tìm Đường Đi Ngắn Nhất Trên Đồ Thị

## 4.1 Bài toán 14: Thuật toán Dijkstra trên Simple Graph

**Đề bài**

**Let G = (V, E) be a finite simple graph. Implement the Dijkstra's algorithm to find the shortest path problem on G.**

**Phân tích bài toán:**
**INPUT:**

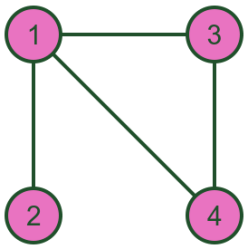
- Một simple graph G = (V, E)
- Đỉnh xuất phát (source)
- Đỉnh đích (target)

**OUTPUT:**

- Một simple graph G = (V, E)
- Đỉnh xuất phát (source)
- Đỉnh đích (target)

**Simple Graph:** là một đồ thị:

- Khoảng cách ngắn nhất từ đỉnh nguồn đến tất cả các đỉnh khác
- Đường đi tương ứng



Simple graph

**Dijkstra's algorithm:** là một thuật toán giải quyết bài toán đường đi ngắn nhất từ một đỉnh đến các đỉnh còn lại của đồ thị có hướng không có cạnh mang trọng số không âm. (Nguồn: Wikipedia)

**Các bước thuật toán:**

1. **Khởi tạo:**
   - `dist[start] = 0`, `dist[other] = ∞`
   - `parent[all] = -1`
   - `visited[all] = false`
2. **Priority Queue:** Min-heap chứa các cặp `(khoảng_cách, đỉnh)`
3. **Vòng lặp chính:**
   - Lấy đỉnh `u` có khoảng cách nhỏ nhất chưa được xử lý
   - Đánh dấu `u` đã xử lý
   - **Relax** tất cả láng giềng `v` của `u`:

```
if dist[u] + weight(u,v) < dist[v]:
   dist[v] = dist[u] + weight(u,v)
   parent[v] = u
   push (dist[v], v) vào priority queue
```

4. **Kết thúc:** Khi priority queue rỗng

**Tính chất quan trọng:**

- **Optimal Substructure:** Đường đi ngắn nhất chứa các đường đi con ngắn nhất
- **Greedy Choice:** Lựa chọn tham lam luôn cho kết quả tối ưi
- **Không hoạt động với trọng số âm:** Thuật toán sẽ cho kết quả sai

**Độ phức tạp:**

- **Time Complexity:** O((V + E) log V)
  - V lần extract-min từ priority queue: O(V log V)
  - E lần decrease-key: O(E log V)
- **Space Complexity:** O(V)
  - Mảng `dist`, `parent`, `visited`: O(V)
  - Priority queue: O(V)

**Cài đặt Python:**

- File: `.\code\Project_5\BT14\dijkstra_simple_graph.py`

**Cài đặt C++:**

- File: `.\code\Project_5\BT14\dijkstra_simple_graph.cpp`

**Format Input (simple_graph.inp):**

```
6 7
0 1 4
1 2 8
2 5 2
0 3 3
3 4 2
4 5 3
1 4 5
```

**Giải thích:**

- Dòng 1: `n m` (n = số đỉnh, m = số cạnh)
- m dòng tiếp: `u v w` (cạnh từ đỉnh u đến v với trọng số w)

**Output mẫu:**

```
Khoảng cách ngắn nhất từ đỉnh 0:
Đỉnh 0: 0 | Đường đi: 0
Đỉnh 1: 4 | Đường đi: 0 → 1
Đỉnh 2: 12 | Đường đi: 0 → 1 → 2
Đỉnh 3: 3 | Đường đi: 0 → 3
Đỉnh 4: 5 | Đường đi: 0 → 3 → 4
Đỉnh 5: 8 | Đường đi: 0 → 3 → 4 → 5
```

### 4.2 Bài toán 15: Thuật toán Dijkstra trên Multigraph

**Đề bài**

**Let G = (V, E) be a finite multigraph. Implement the Dijkstra's algorithm to find the shortest path problem on G.**

**INPUT:**

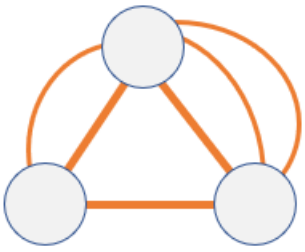
- Một multigraph G = (V, E) với trọng số không âm
- Đỉnh xuất phát (source)

**OUTPUT:**

- Khoảng cách ngắn nhất từ đỉnh nguồn đến tất cả các đỉnh khác
- Đường đi tương ứng

**Multigraph:** là một đồ thị:

- Có nhiều cạnh giữa cùng một cặp đỉnh
- Có thể có self-loops (cạnh từ đỉnh về chính nó)



Multi Graph

**Các bước thuật toán:**

1. **Khởi tạo:**
   - `dist[start] = 0, dist[other] = ∞`
   - `parent[all] = -1`
   - `visited[all] = false`
2. **Priority Queue:** Min-heap chứa các cặp `(khoảng_cách, đỉnh)`
3. **Vòng lặp chính:**
   - Lấy đỉnh `u` có khoảng cách nhỏ nhất chưa được xử lý
   - Đánh dấu `u` đã xử lý

- Duyệt **tất cả cạnh** từ `u` (bao gồm multiple edges)
- **Bỏ qua self-loops:** `if u == v: continue`
- **Relax** các láng giềng `v` của `u`:

```
if dist[u] + weight < dist[v]:
    dist[v] = dist[u] + weight
    parent[v] = u
    push (dist[v], v) vào priority queue
```

4. **Kết thúc:** Khi priority queue rỗng

**Cài đặt Python:**

- File: `.\code\Project_5\BT14\dijkstra_multigraph.py`

**Cài đặt C++:**

- File: `.\code\Project_5\BT14\dijkstra_multigraph.cpp`

**Format Input (multigraph.inp):**

```
6 9
0 1 4
0 1 6
1 2 8
2 5 2
0 3 3
3 4 2
4 5 3
1 4 5
4 4 1
```

**Output mẫu:**

```
Khoảng cách ngắn nhất từ đỉnh 0:
Đỉnh 0: 0 | Đường đi: 0
Đỉnh 1: 4 | Đường đi: 0 → 1
Đỉnh 2: 12 | Đường đi: 0 → 1 → 2
Đỉnh 3: 3 | Đường đi: 0 → 3
Đỉnh 4: 5 | Đường đi: 0 → 3 → 4
Đỉnh 5: 8 | Đường đi: 0 → 3 → 4 → 5
```

### 4.3 Bài toán 16: Thuật toán Dijkstra trên General Graph

**Đề bài**

**Let G = (V, E) be a general graph. Implement the Dijkstra's algorithm to find the shortest path problem on G.**

**Phân tích bài toán:**
**INPUT:**

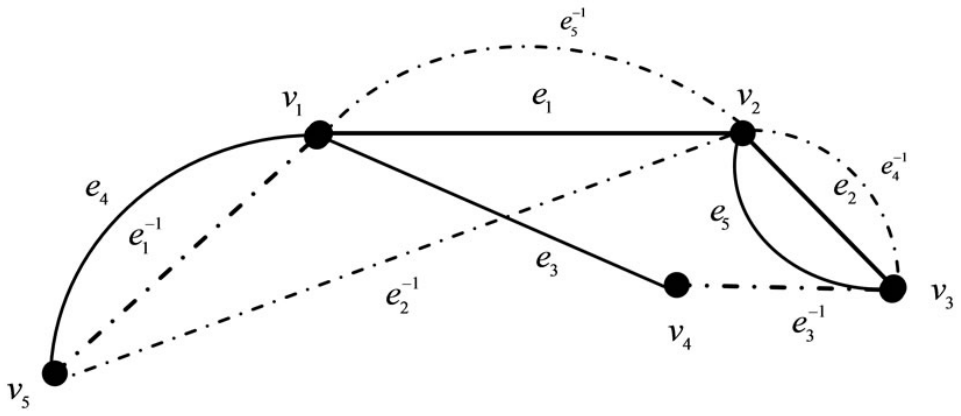
- Một general graph G = (V, E)
- Đỉnh xuất phát (source)

**OUTPUT:**

- Khoảng cách ngắn nhất từ đỉnh nguồn đến tất cả các đỉnh khác
- Đường đi tương ứng (nếu thuật toán hoạt động)

**General Graph:** là đồ thị tổng quát nhất, bao gồm:

- **Directed Graph:** Đồ thị có hướng
- **Mixed Graph:** Kết hợp cạnh có hướng và vô hướng
- **Negative Weights:** Có thể có trọng số âm
- **Directed Cycles:** Chu trình có hướng




**General Graph**

**Vấn đề với General Graph:**
**Dijkstra's Algorithm KHÔNG hoạt động với trọng số âm** vì:

- Greedy choice có thể cho kết quả sai
- Một khi đỉnh được đánh dấu "visited", khoảng cách không được cập nhật lại
- Trọng số âm có thể tạo ra đường đi ngắn hơn sau khi đỉnh đã được xử lý

**Các bước thuật toán:**

1. **Kiểm tra điều kiện:**
   - Duyệt tất cả cạnh trong đồ thị
   - Nếu phát hiện trọng số âm → dừng thuật toán
   - Thông báo lỗi và đề xuất Bellman-Ford
2. **Khởi tạo (nếu không có trọng số âm):**
   - `dist[start] = 0, dist[other] =` $\infty$
   - `parent[all] = -1`
   - `visited[all] = false`
3. **Priority Queue:** Min-heap chứa các cặp `(khoảng_cách, đỉnh)`
4. **Vòng lặp chính:**
   - Lấy đỉnh `u` có khoảng cách nhỏ nhất chưa được xử lý
   - Đánh dấu `u` đã xử lý
   - Duyệt **chỉ cạnh đi ra** từ `u` (directed)
   - **Relax** các láng giềng `v` của `u`:

```
if dist[u] + weight < dist[v]:
    dist[v] = dist[u] + weight
    parent[v] = u
    push (dist[v], v) vào priority queue
```

5. **Kết thúc:** Khi priority queue rỗng

**Cài đặt Python:**

- File: `.\code\Project_5\BT14\dijkstra_general_graph.py`

**Cài đặt C++:**

- File: `.\code\Project_5\BT14\dijkstra_general_graph.cpp`

**Format Input:**
**1. Directed Graph (general_graph.inp):**

```
# directed
6 8
0 1 4
1 2 8
2 5 2
0 3 3
3 4 2
4 5 3
1 4 5
4 1 1
```

**2. Graph với trọng số âm (negative_graph.inp):**

```
# directed
4 5
0 1 4
1 2 -3
2 3 2
0 3 7
1 3 5
```

**Output mẫu:**

**1. Directed Graph hợp lệ:**

```
Khoảng cách ngắn nhất từ đỉnh 0:
Đỉnh 0: 0 | Đường đi: 0
Đỉnh 1: 4 | Đường đi: 0 → 1
Đỉnh 2: 12 | Đường đi: 0 → 1 → 2
Đỉnh 3: 3 | Đường đi: 0 → 3
Đỉnh 4: 5 | Đường đi: 0 → 3 → 4
Đỉnh 5: 8 | Đường đi: 0 → 3 → 4 → 5
```

**2. Graph với trọng số âm:**

```
Phát hiện trọng số âm tại cạnh 1→2 (trọng số: -3)
```