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 4: Các Bài Toán Duyệt Đồ Thị và Cây

2.1 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)

2.1.1 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

2.2 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++`

2.3 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`

2.4 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`

2.5 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`

2.6 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`

2.7 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`

2.8 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`

2.9 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`

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

3.1 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:

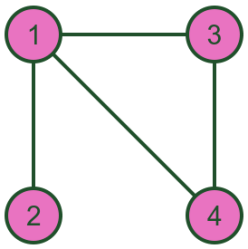
- Một simple graph G = (V, E)
- Đỉnh xuất phát (source)
- Đỉnh đích (target)

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
```

3.2 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:

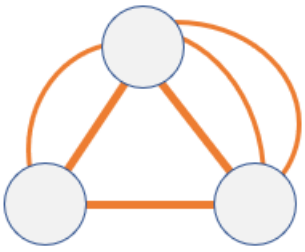
- Một multigraph G = (V, E) với trọng số không âm
- Đỉnh xuất phát (source)

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
```

3.3 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:

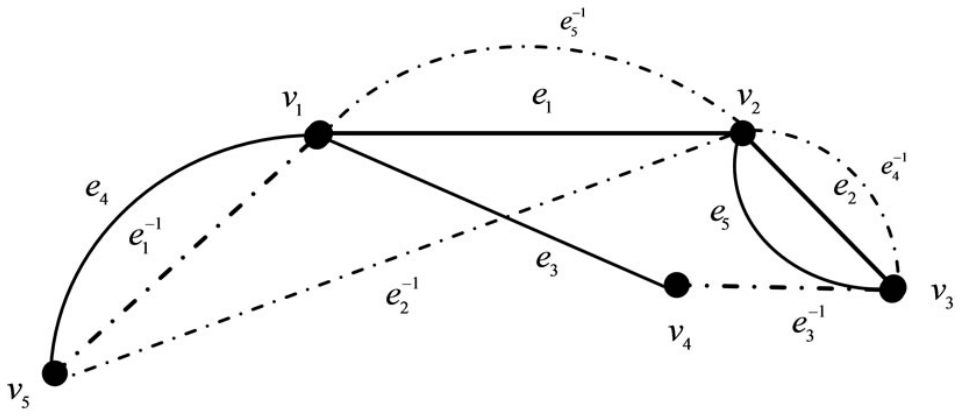
- Một general graph G = (V, E)
- Đỉnh xuất phát (source)

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] =` ∞
   - `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)
```