ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
**KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH**


## CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT - CO2003

---

## BÀI TẬP LỚN 3

# HIỆN THỰC ĐỒ THỊ TRI THỨC SỬ DỤNG CẤU TRÚC DỮ LIỆU ĐỒ THỊ

---

TP. HỒ CHÍ MINH, THÁNG 11/2025

# ĐẶC TẢ BÀI TẬP LỚN

**Phiên bản 1.0**

## 1 Chuẩn đầu ra

Sau khi hoàn thành bài tập lớn này, sinh viên sẽ có khả năng:

- Thành thạo lập trình hướng đối tượng (OOP).
- Phát triển cấu trúc dữ liệu dạng đồ thị.
- Sử dụng các cấu trúc dữ liệu dạng đồ thị để xây dựng một Đồ thị Tri thức

## 2 Dẫn nhập

Trong các bài tập lớn trước, sinh viên đã xây dựng thành công *VectorStore* với khả năng lưu trữ và tìm kiếm dựa trên độ tương đồng vector, sử dụng các cấu trúc dữ liệu từ cơ bản (Danh sách) đến nâng cao (Cây AVL, Red-Black Tree). Tuy nhiên, trong các bài toán thực tế về quản lý tri thức, dữ liệu không chỉ tồn tại độc lập hay chỉ được liên kết qua khoảng cách toán học, mà còn gắn kết với nhau bởi các mối quan hệ ngữ nghĩa trực tiếp (ví dụ: bài viết trích dẫn nhau, sản phẩm cùng danh mục, hay các khái niệm liên quan).

Bài tập lớn 3 yêu cầu sinh viên nâng cấp hệ thống hiện tại bằng cách tích hợp cấu trúc dữ liệu **Đồ thị (Graph)** để xây dựng một **Đồ thị Tri thức (Knowledge Graph)**. Trong mô hình này, mỗi chuỗi sẽ đóng vai trò là một đỉnh (vertex) của đồ thị, và các mối liên hệ giữa chúng sẽ được biểu diễn bằng các cạnh (edge).

Việc chuyển đổi này cho phép hệ thống không chỉ trả lời câu hỏi "cái gì giống cái này nhất" mà còn trả lời được các câu hỏi mang tính cấu trúc như "các thực thể nào có liên quan đến nhau" hoặc "lộ trình kết nối giữa hai thực thể bất kỳ". Thông qua bài tập này, sinh viên sẽ hoàn thiện bức tranh tổng thể về việc ứng dụng các Cấu trúc Dữ liệu và Giải thuật để giải quyết bài toán lưu trữ và khai thác tri thức đa chiều.

# 3 Mô tả

## 3.1 Cấu trúc dữ liệu đồ thị

Cấu trúc dữ liệu đồ thị (Graph) trong BTL này được thiết kế với cách tiếp cận đơn giản hóa, gồm các class chính như sau:

- class `DGraphModel<T>`: Là class chính để hiện thực đồ thị có hướng (Directed Graph). Class này chứa toàn bộ các phương thức để thêm, xóa và quản lý các đỉnh và cạnh trong đồ thị có hướng. Không sử dụng mô hình kế thừa phức tạp hay interface, tất cả logic được tích hợp trực tiếp trong class này.
- class `Edge`: Đại diện cho một cạnh trong đồ thị, bao gồm thông tin về hai đỉnh mà nó kết nối (nguồn và đích) cùng với trọng số.
- class `VertexNode<T>`: Đại diện cho một đỉnh trong đồ thị cùng tất cả các cạnh liên quan (danh sách kề).

### 3.1.1 Edge

class `Edge` đại diện cho một cạnh trong đồ thị, bao gồm thông tin về hai đỉnh mà nó kết nối (`from` và `to`) cùng với trọng số (`weight`).

1. **Các thuộc tính:**
   - `VertexNode* from`: Con trỏ tới đỉnh nguồn của cạnh.
   - `VertexNode* to`: Con trỏ tới đỉnh đích của cạnh.
   - `float weight`: Trọng số của cạnh. Mặc định là 0 nếu không được chỉ định.
2. **Hàm khởi tạo và hàm hủy:**
   - `Edge()`: Hàm khởi tạo mặc định, khởi tạo một cạnh mà không có thông tin cụ thể về các đỉnh và trọng số.
   - `Edge(VertexNode* from, VertexNode* to, float weight = 0)`: Hàm khởi tạo với các tham số chỉ định đỉnh nguồn `from`, đỉnh đích `to`, và trọng số của cạnh `weight` (mặc định là 0).
3. **Các phương thức:**
   - **bool equals(Edge* edge)**
     - **Chức năng**: So sánh cạnh hiện tại với một cạnh khác `edge`. Trả về `true` nếu cả hai cạnh có cùng đỉnh nguồn và đỉnh đích, và `false` nếu không.

  - **Ngoại lệ**: Không có.
- **static bool edgeEQ(Edge*& edge1, Edge*& edge2)**
  - **Chức năng**: So sánh hai đối tượng `Edge` `edge1` và `edge2` bằng cách gọi phương thức `equals` của class `Edge`. Trả về `true` nếu hai cạnh giống nhau, và `false` nếu không.
  - **Ngoại lệ**: Không có.
- **string toString()**
  - **Chức năng**: Trả về chuỗi biểu diễn của cạnh, bao gồm thông tin về đỉnh nguồn `from`, đỉnh đích `to`, và trọng số của cạnh `weight`
  - **Ngoại lệ**: Không có.

### 3.1.2 VertexNode

class `VertexNode<T>` được sử dụng để đại diện cho một đỉnh và các cạnh liên quan trong danh sách kề. Mỗi đỉnh lưu trữ giá trị của nó, danh sách các cạnh đi ra ngoài (outward edges) và danh sách các cạnh đi vào (inward edges).

1. **Các thuộc tính:**
   - `T vertex`: Dữ liệu được lưu trữ trong đỉnh, với `T` là kiểu dữ liệu tổng quát.
   - `int inDegree_`: Bậc vào của đỉnh (số cạnh đi vào đỉnh).
   - `int outDegree_`: Bậc ra của đỉnh (số cạnh đi ra từ đỉnh).
   - `vector<Edge*> adList`: Danh sách liên kết đôi lưu trữ danh sách kề của các cạnh kết nối với đỉnh.
   - `bool (*vertexEQ)(T&, T&)`: Con trỏ hàm dùng để so sánh dữ liệu được lưu tại hai đỉnh có bằng nhau hay không.
   - `string (*vertex2str)(T&)`: Con trỏ hàm dùng để chuyển dữ liệu của đỉnh thành chuỗi ký tự.
2. **Hàm khởi tạo:**
   - `VertexNode(T vertex, bool (*vertexEQ)(T&, T&), string (*vertex2str)(T&))`: Hàm khởi tạo, tạo một đỉnh với dữ liệu `vertex`, hàm kiểm tra tính bằng nhau `vertexEQ`, và hàm chuyển đổi sang chuỗi `vertex2str`.
3. **Các phương thức:**
   - `T& getVertex()`
     - **Chức năng**: Trả về tham chiếu đến dữ liệu của đỉnh.

  - **Ngoại lệ**: Không có.
- `void connect(VertexNode* to, float weight = 0)`
  - **Chức năng**: Kết nối đỉnh hiện tại với đỉnh `to` bằng cách tạo một cạnh với trọng số (mặc định là 0).
  - **Ngoại lệ**: Không có.
- `Edge* getEdge(VertexNode* to)`
  - **Chức năng**: Trả về con trỏ tới cạnh nối từ đỉnh hiện tại tới đỉnh `to`. Trả về `nullptr` nếu không tìm thấy cạnh.
  - **Ngoại lệ**: Không có.
- `bool equals(VertexNode* node)`
  - **Chức năng**: So sánh đỉnh hiện tại với một đỉnh khác `node` dựa trên hàm `vertexEQ`.
  - **Ngoại lệ**: Không có.
- `void removeTo(VertexNode* to)`
  - **Chức năng**: Xóa cạnh nối từ đỉnh hiện tại tới đỉnh `to`.
  - **Ngoại lệ**: Không có.
- `int inDegree()`
  - **Chức năng**: Trả về bậc vào của đỉnh.
  - **Ngoại lệ**: Không có.
- `int outDegree()`
  - **Chức năng**: Trả về bậc ra của đỉnh.
  - **Ngoại lệ**: Không có.
- `string toString()`
  - **Chức năng**: Trả về chuỗi biểu diễn của đỉnh, bao gồm dữ liệu, bậc vào, và bậc ra.
  - **Ngoại lệ**: Không có.

### 3.1.3 Directed Graph

class `DGraphModel<T>` là một class hiện thực trực tiếp một mô hình đồ thị có hướng (Directed Graph). Class này cung cấp các phương thức để thêm, xóa và quản lý các đỉnh và cạnh trong một đồ thị có hướng. Không sử dụng interface hay abstract class, tất cả logic được tích hợp trực tiếp trong class này.

1. **Các thuộc tính:**

- `vector<VertexNode<T>*> nodeList`: Danh sách chứa toàn bộ các đỉnh của đồ thị.
- `bool (*vertexEQ)(T&, T&)`: Con trỏ tới hàm so sánh hai đỉnh, dùng để kiểm tra tính bằng nhau của các đỉnh trong đồ thị.
- `string (*vertex2str)(T&)`: Con trỏ tới hàm chuyển đổi dữ liệu của một đỉnh thành chuỗi, được sử dụng để biểu diễn đỉnh dưới dạng chuỗi.

2. **Hàm khởi tạo và hàm hủy:**

- `DGraphModel(bool (*vertexEQ)(T&, T&)=0, string (*vertex2str)(T&)=0)`: Hàm khởi tạo class `DGraphModel` với hai tham số con trỏ hàm `vertexEQ` và `vertex2str` dùng để so sánh và biểu diễn đỉnh. Nếu không có giá trị nào được cung cấp, các tham số này sẽ là `nullptr`.
- `~DGraphModel()`: Hàm hủy, giải phóng tài nguyên của đồ thị, xóa các đỉnh và các cạnh liên quan.

3. **Các phương thức:**

- `VertexNode<T>* getVertexNode(T& vertex)`
  - **Chức năng**: Tìm kiếm và trả về con trỏ đến nút `VertexNode` chứa đỉnh `vertex` trong đồ thị. Nếu không tìm thấy, trả về `nullptr`.
  - **Ngoại lệ**: Không có.
- `string vertex2Str(VertexNode<T>& node)`
  - **Chức năng**: Chuyển đổi một nút `VertexNode` thành chuỗi biểu diễn của đỉnh thông qua hàm `vertex2str`.
  - **Ngoại lệ**: Không có.
- `string edge2Str(Edge& edge)`
  - **Chức năng**: Chuyển đổi một đối tượng `Edge` thành chuỗi biểu diễn của cạnh, bao gồm thông tin về đỉnh nguồn và đỉnh đích.
  - **Ngoại lệ**: Không có.
- `void add(T vertex)`
  - **Chức năng**: Thêm một đỉnh mới vào đồ thị.
  - **Ngoại lệ**: Không có.
- `bool contains(T vertex)`
  - **Chức năng**: Kiểm tra xem đồ thị có chứa đỉnh `vertex` hay không.
  - **Ngoại lệ**: Không có.
- `float weight(T from, T to)`
  - **Chức năng**: Trả về trọng số của cạnh nối từ đỉnh `from` đến đỉnh `to`.

  - **Ngoại lệ**: Ném ngoại lệ `VertexNotFoundException` nếu không tìm thấy đỉnh, và `EdgeNotFoundException` nếu không tìm thấy cạnh giữa hai đỉnh.

- `vector<T> getOutwardEdges(T from)`
  - **Chức năng**: Lấy danh sách các cạnh đi ra từ đỉnh `from`.
  - **Hướng hiện thực**:
    (a) Tìm `VertexNode` tương ứng với đỉnh `from` thông qua hàm `getVertexNode(from)`.
    (b) Nếu không tìm thấy đỉnh `from`, ném ngoại lệ `VertexNotFoundException` với thông tin về đỉnh.
    (c) Nếu đỉnh `from` tồn tại, trả về danh sách các cạnh đi ra từ đỉnh này thông qua phương thức `getOutwardEdges()` của đối tượng `VertexNode`.
  - **Ngoại lệ**:
    * `VertexNotFoundException`: Nếu không tìm thấy đỉnh `from`.
- `void connect(T from, T to, float weight = 0)`
  - **Chức năng**: Kết nối hai đỉnh `from` và `to` bằng một cạnh, với trọng số mặc định là `0`.
  - **Hướng hiện thực**:
    (a) Kiểm tra xem đỉnh `from` đã tồn tại trong đồ thị chưa. Nếu chưa, ném ngoại lệ `VertexNotFoundException` với thông tin về đỉnh `from`.
    (b) Kiểm tra xem đỉnh `to` đã tồn tại trong đồ thị chưa. Nếu chưa, ném ngoại lệ `VertexNotFoundException` với thông tin về đỉnh `to`.
    (c) Lấy con trỏ tới `VertexNode` của đỉnh `from` và `to`.
    (d) Gọi phương thức `connect` của đỉnh `from` để tạo cạnh nối tới đỉnh `to` với trọng số `weight`.
  - **Tham số**:
    * `T from`: Đỉnh bắt đầu.
    * `T to`: Đỉnh kết thúc.
    * `float weight`: Trọng số của cạnh (mặc định là 0).
  - **Ngoại lệ**:
    * `VertexNotFoundException`: Nếu không tìm thấy đỉnh `from` hoặc đỉnh `to`.
- `void disconnect(T from, T to)`
  - **Chức năng**: Xóa cạnh giữa hai đỉnh `from` và `to`.
  - **Hướng hiện thực**:
    (a) Kiểm tra xem đỉnh `from` đã tồn tại trong đồ thị chưa. Nếu chưa, ném ngoại lệ `VertexNotFoundException` với thông tin về đỉnh `from`.

(b) Kiểm tra xem đỉnh `to` đã tồn tại trong đồ thị chưa. Nếu chưa, ném ngoại lệ `VertexNotFoundException` với thông tin về đỉnh `to`.

(c) Lấy con trỏ tới `VertexNode` của đỉnh `from` và `to`.

(d) Gọi phương thức `removeTo` của đỉnh `from` để xóa cạnh nối tới đỉnh `to`.

- **Tham số**:
  - `T from`: Đỉnh bắt đầu.
  - `T to`: Đỉnh kết thúc.
- **Ngoại lệ**:
  - `VertexNotFoundException`: Nếu không tìm thấy đỉnh `from` hoặc đỉnh `to`.

- `bool connected(T from, T to)`
  - **Chức năng**: Kiểm tra xem hai đỉnh `from` và `to` có được kết nối bằng một cạnh hay không.
  - **Hướng hiện thực**:
    (a) Tìm `VertexNode` tương ứng với đỉnh `from` thông qua hàm `getVertexNode(from)`.
    (b) Nếu không tìm thấy đỉnh `from`, ném ngoại lệ `VertexNotFoundException` với thông tin về đỉnh.
    (c) Tìm `VertexNode` tương ứng với đỉnh `to` thông qua hàm `getVertexNode(to)`.
    (d) Nếu không tìm thấy đỉnh `to`, ném ngoại lệ `VertexNotFoundException` với thông tin về đỉnh.
    (e) Gọi phương thức `getEdge` của đỉnh `from` để kiểm tra xem có cạnh tới đỉnh `to` không.
    (f) Trả về `true` nếu tìm thấy cạnh, ngược lại trả về `false`.
  - **Tham số**:
    - `T from`: Đỉnh bắt đầu.
    - `T to`: Đỉnh kết thúc.
  - **Giá trị trả về**: `true` nếu hai đỉnh được kết nối, ngược lại `false`.
  - **Ngoại lệ**:
    - `VertexNotFoundException`: Nếu không tìm thấy đỉnh `from` hoặc đỉnh `to`.
- `int size()`
  - **Chức năng**: Trả về số lượng đỉnh trong đồ thị.
  - **Ngoại lệ**: Không có.
- `bool empty()`
  - **Chức năng**: Kiểm tra xem đồ thị có rỗng hay không.
  - **Ngoại lệ**: Không có.

- `void clear()`
  - **Chức năng**: Xóa tất cả các đỉnh và cạnh trong đồ thị.
  - **Ngoại lệ**: Không có.
- `int inDegree(T vertex)`
  - **Chức năng**: Trả về bậc vào (số cạnh đi vào) của đỉnh `vertex`.
  - **Ngoại lệ**: Ném ngoại lệ `VertexNotFoundException` nếu không tìm thấy đỉnh `vertex`.
- `int outDegree(T vertex)`
  - **Chức năng**: Trả về bậc ra (số cạnh đi ra) của đỉnh `vertex`.
  - **Ngoại lệ**: Ném ngoại lệ `VertexNotFoundException` nếu không tìm thấy đỉnh `vertex`.
- `vector<T> vertices()`
  - **Chức năng**: Trả về danh sách tất cả các đỉnh trong đồ thị.
  - **Ngoại lệ**: Không có.
- `string toString()`
  - **Chức năng**: Trả về chuỗi biểu diễn toàn bộ đồ thị, bao gồm danh sách các đỉnh và các cạnh.
  - **Ngoại lệ**: Không có.
- `string BFS(T start)`
  - **Chức năng**: Duyệt đồ thị theo chiều rộng (Breadth-First Search) bắt đầu từ đỉnh `start`, trả về một chuỗi biểu diễn danh sách các đỉnh được thăm theo thứ tự BFS, trong đó mỗi đỉnh được chuyển sang chuỗi bằng hàm `vertex2Str`
  - **Tham số**:
    - `T start`: Đỉnh bắt đầu duyệt BFS.
  - **Giá trị trả về**: `string` chứa biểu diễn chuỗi của các đỉnh được thăm theo thứ tự BFS bắt đầu từ `start`.
  - **Ngoại lệ**:
    - `VertexNotFoundException`: Nếu không tìm thấy đỉnh `start` trong đồ thị.
- `string DFS(T start)`
  - **Chức năng**: Duyệt đồ thị theo chiều sâu (Depth-First Search) bắt đầu từ đỉnh `start`, trả về một chuỗi biểu diễn danh sách các đỉnh được thăm theo thứ tự DFS, trong đó mỗi đỉnh được chuyển sang chuỗi bằng hàm `vertex2Str`.
  - **Tham số**:
    - `T start`: Đỉnh bắt đầu duyệt DFS.

- **Giá trị trả về**: `string` chứa biểu diễn chuỗi của các đỉnh được thăm theo thứ tự DFS bắt đầu từ `start`.
- **Ngoại lệ**:
  * `VertexNotFoundException`: Nếu không tìm thấy đỉnh `start` trong đồ thị.

## 3.2 Đồ thị Tri thức

### 3.2.1 Giới thiệu

Đồ thị Tri thức (Knowledge Graph) là một mô hình biểu diễn thông tin dưới dạng đồ thị, trong đó các thực thể (entities) như người, địa điểm, sự kiện, hay khái niệm được biểu diễn bằng các nút (nodes), và các mối quan hệ giữa chúng được biểu diễn bằng các cạnh (edges). Khác với các cơ sở dữ liệu truyền thống lưu trữ dữ liệu dạng bảng rời rạc, Đồ thị Tri thức tổ chức dữ liệu thành một mạng lưới liên kết có cấu trúc và ngữ nghĩa rõ ràng.

Cơ chế hoạt động của Đồ thị Tri thức bắt đầu từ việc tập hợp dữ liệu từ nhiều nguồn khác nhau, sau đó xử lý để nhận diện chính xác các thực thể và xác định các mối quan hệ giữa chúng. Các thuật toán xử lý ngôn ngữ tự nhiên và học máy được áp dụng để trích xuất, làm sạch và hợp nhất thông tin. Đồ thị này cho phép truy vấn thông minh dựa trên các mối quan hệ và ngữ cảnh của dữ liệu, thay vì chỉ tìm kiếm dữ liệu theo từ khóa đơn giản.

Thông qua Đồ thị Tri thức, việc truy xuất, phân tích và khai thác thông tin trở nên hiệu quả hơn, hỗ trợ các ứng dụng trí tuệ nhân tạo, hệ thống đề xuất, hỗ trợ ra quyết định và nhiều lĩnh vực khác như thương mại điện tử, y tế, giáo dục và quản trị dữ liệu doanh nghiệp.

### 3.2.2 Lớp KnowledgeGraph

Lớp `KnowledgeGraph` đại diện cho một đồ thị tri thức, bao gồm các thành phần như sau.

**Các thuộc tính**

- `DGraphModel<string> graph`: Đối tượng đồ thị cơ sở lưu trữ các thực thể và mối quan hệ.
- `vector<string> entities`: Danh sách tất cả các thực thể trong đồ thị (đồng bộ với `graph`).

**Các phương thức**

- `KnowledgeGraph()`

  - **Chức năng**: Hàm khởi tạo mặc định của lớp `KnowledgeGraph`.

- `void addEntity(string entity)`
  - **Chức năng**: Thêm thực thể mới vào Đồ thị Tri thức.
  - **Hướng hiện thực**:
    1. Kiểm tra xem thực thể đã tồn tại hay chưa, nếu có ném `EntityExistsException`.
    2. Thêm thực thể mới vào đồ thị tri thức,
    3. Thêm `entity` vào cuối danh sách các thực thể.
  - **Tham số**:
    * `string entity`: Tên thực thể.
  - **Giá trị trả về**: Không có.
  - **Ngoại lệ**:
    * `EntityExistsException`: Nếu thực thể đã tồn tại.

- `void addRelation(string from, string to, float weight = 1.0f)`
  - **Chức năng**: Thêm mối quan hệ có hướng với loại quan hệ cụ thể.
  - **Hướng hiện thực**:
    1. Kiểm tra cả `from` và `to` có tồn tại hay không.
    2. Nếu không, ném `EntityNotFoundException`.
    3. Gọi hàm để nối một cạnh mới.
  - **Tham số**:
    * `string from`: Thực thể nguồn.
    * `string to`: Thực thể đích.
    * `float weight`: Trọng số (mặc định 1.0).
  - **Giá trị trả về**: Không có.
  - **Ngoại lệ**:
    * `EntityNotFoundException`: Nếu `from` hoặc `to` không tồn tại.

- `vector<string> getAllEntities()`
  - **Chức năng**: Trả về danh sách tất cả thực thể trong đồ thị.
  - **Giá trị trả về**: `vector<string>` chứa tất cả thực thể.

- `vector<string> getNeighbors(string entity)`
  - **Chức năng**: Trả về danh sách các thực thể kề trực tiếp (out-neighbors).
  - **Hướng hiện thực**:
    1. Kiểm tra xem đồ thị có chứa thực thể `entity` hay không, nếu không ném `EntityNotFoundException`.

2. Trả về danh sách các thực thể kề.

- **Tham số**:
  - `string entity`: Thực thể cần tìm hàng xóm.
- **Giá trị trả về**: `vector<string>` các thực thể kề.
- **Ngoại lệ**:
  - `EntityNotFoundException`: Nếu thực thể không tồn tại.

- `string bfs(string start)`
  - **Chức năng**: Duyệt BFS từ thực thể bắt đầu, trả về chuỗi thứ tự duyệt. Mỗi nút
  - **Hướng hiện thực**:
  - **Tham số**:
    - `string start`: Thực thể bắt đầu.
  - **Giá trị trả về**: `string` thứ tự duyệt BFS.
  - **Ngoại lệ**:
    - `EntityNotFoundException`: Nếu thực thể không tồn tại.
- `string dfs(string start)`
  - **Chức năng**: Duyệt DFS từ thực thể bắt đầu, trả về chuỗi thứ tự duyệt.
  - **Tham số**:
    - `string start`: Thực thể bắt đầu.
  - **Giá trị trả về**: `string` thứ tự duyệt DFS.
  - **Ngoại lệ**:
    - `EntityNotFoundException`: Nếu thực thể không tồn tại.
- `bool isReachable(string from, string to)`
  - **Chức năng**: Kiểm tra xem có đường đi từ thực thể nguồn đến đích.
  - **Tham số**:
    - `string from`: Thực thể nguồn.
    - `string to`: Thực thể đích.
  - **Giá trị trả về**: `bool` - true nếu có đường đi.
  - **Ngoại lệ**:
    - `EntityNotFoundException`: Nếu một trong hai không tồn tại.
- `string toString()`
  - **Chức năng**: Biểu diễn chuỗi tổng quan Đồ thị Tri thức.
  - **Giá trị trả về**: `string` biểu diễn đồ thị.

- `vector<string> getRelatedEntities(string entity, int depth = 2)`
  - **Chức năng**: Tìm tất cả thực thể liên quan đến một thực thể trong phạm vi `depth` bước.
  - **Hướng hiện thực**:
    1. Kiểm tra xem đỉnh `entity` có tồn tại hay không, nếu không thì ném ra lỗi `EntityNotFoundException`.
    2. Dùng BFS với giới hạn độ sâu `depth`, thu thập tất cả đỉnh thăm được.
    3. Loại bỏ trùng lặp và trả về danh sách các thực thể liên quan.
  - **Tham số**:
    * `string entity`: Thực thể trung tâm.
    * `int depth`: Độ sâu tìm kiếm (mặc định 2).
  - **Giá trị trả về**: `vector<string>` các thực thể liên quan.
  - **Ngoại lệ**:
    * `EntityNotFoundException`: Nếu thực thể không tồn tại.
- `string findCommonAncestors(string entity1, string entity2)`
  - **Chức năng**: Tìm tổ tiên chung gần nhất (LCA) của 2 thực thể (đặc trưng KG: reasoning về mối quan hệ chung).
  - **Hướng hiện thực**:
    1. Tìm tất cả tổ tiên của `entity1` và `entity2` bằng DFS ngược.
    2. Tìm giao của 2 tập tổ tiên, chọn tổ tiên gần nhất (có đường đi ngắn nhất đến cả 2).
  - **Tham số**:
    * `string entity1`: Thực thể thứ nhất.
    * `string entity2`: Thực thể thứ hai.
  - **Giá trị trả về**: `string` tên tổ tiên chung gần nhất hoặc "No common ancestor".
  - **Ngoại lệ**:
    * `EntityNotFoundException`: Nếu một trong hai không tồn tại.

# 4 Yêu cầu và chấm điểm

## 4.1 Yêu cầu

Để hoàn thành bài tập này, sinh viên cần:

1. Đọc toàn bộ file mô tả này.
2. Tải file **initial.zip** và giải nén. Sau khi giải nén, sinh viên sẽ nhận được các file bao gồm: `utils.h`, `main.cpp`, `main.h`, `KnowledgeGraph.h`, `KnowledgeGraph.cpp`. Sinh viên chỉ nộp 2 file, đó là `KnowledgeGraph.h` và `KnowledgeGraph.cpp`. Do đó, không được phép chỉnh sửa file `main.h` khi kiểm tra chương trình.
3. Sinh viên sử dụng lệnh sau để biên dịch:
   `g++ -o main main.cpp KnowledgeGraph.cpp -I . -std=c++17`
   Lệnh trên được sử dụng trong Command Prompt/Terminal để biên dịch chương trình. Nếu sinh viên sử dụng IDE để chạy chương trình, cần lưu ý: thêm tất cả các file vào project/workspace của IDE; chỉnh lại lệnh biên dịch trong IDE cho phù hợp. IDE thường cung cấp nút Build (biên dịch) và Run (chạy). Khi bấm Build, IDE sẽ chạy câu lệnh biên dịch tương ứng, thông thường chỉ biên dịch file `main.cpp`. Sinh viên cần tìm cách cấu hình để thay đổi câu lệnh biên dịch, cụ thể: thêm file `KnowledgeGraph.cpp`, thêm tùy chọn `-std=c++17`, và `-I .`
4. Chương trình sẽ được chấm trên nền tảng Unix. Môi trường của sinh viên và trình biên dịch có thể khác với môi trường chấm thực tế. Khu vực nộp bài trên LMS được thiết lập tương tự môi trường chấm thực tế. Sinh viên bắt buộc phải kiểm tra chương trình trên trang nộp bài, đồng thời sửa tất cả lỗi phát sinh trên hệ thống LMS để đảm bảo kết quả chính xác khi chấm cuối cùng.
5. Chỉnh sửa file `KnowledgeGraph.h` và `KnowledgeGraph.cpp` để hoàn thành bài tập, đồng thời đảm bảo hai yêu cầu sau:
   - Tất cả các phương thức được mô tả trong file hướng dẫn phải được cài đặt để chương trình có thể biên dịch thành công. Nếu sinh viên chưa hiện thực một phương thức nào đó, cần cung cấp phần hiện thực rỗng cho phương thức đó. Mỗi testcase sẽ gọi một số phương thức để kiểm tra kết quả trả về.
   - Trong file `KnowledgeGraph.h` chỉ được phép có đúng một dòng `#include "main.h"`, và trong file `KnowledgeGraph.cpp` chỉ được phép có một dòng `#include "KnowledgeGraph.h"`. Ngoài hai dòng này, không được phép thêm bất kỳ lệnh `#include` nào khác trong các file này.
   - Sinh viên không được sử dụng lệnh `#define TESTING` trong 2 file được yêu cầu chỉnh sửa.
6. Khuyến khích sinh viên được phép viết thêm các lớp, phương thức và thuộc tính phụ trợ trong các lớp yêu cầu hiện thực. Nhưng sinh viên phải đảm bảo các lớp, phương thức này không làm thay đổi yêu cầu của các phương thức được mô tả trong đề bài.
7. Sinh viên phải thiết kế và sử dụng các cấu trúc dữ liệu đã học.

8. Sinh viên bắt buộc phải giải phóng toàn bộ vùng nhớ cấp phát động khi chương trình kết thúc.

### 4.2 Thời hạn nộp bài

Thời hạn **theo thông báo trên LMS**. Sinh viên vui lòng nộp bài lên hệ thống trước thời hạn thông báo. Sinh viên tự chịu trách nhiệm nếu xuất hiện các lỗi trên hệ thống do nộp gần sát giờ quy định.

### 4.3 Chấm điểm

Toàn bộ mã nguồn của sinh viên sẽ được chấm trên bộ testcases ẩn, điểm được tính theo từng yêu cầu.

## 5 Harmony cho Bài tập lớn

Bài kiểm tra cuối kì của môn học sẽ có một số câu hỏi Harmony với nội dung của BTL.

Sinh viên phải giải quyết BTL bằng khả năng của chính mình. Nếu sinh viên gian lận trong BTL, sinh viên sẽ không thể trả lời câu hỏi Harmony và nhận điểm 0 cho BTL.

Sinh viên **phải** chú ý làm câu hỏi Harmony trong bài kiểm tra cuối kỳ. Các trường hợp không làm sẽ tính là 0 điểm cho BTL, và bị không đạt cho môn học. **Không chấp nhận giải thích và không có ngoại lệ.**

## 6 Quy định và xử lý gian lận

Bài tập lớn phải được sinh viên TỰ LÀM. Sinh viên sẽ bị coi là gian lận nếu:

- Có sự giống nhau bất thường giữa mã nguồn của các bài nộp. Trong trường hợp này, TẤT CẢ các bài nộp đều bị coi là gian lận. Do vậy sinh viên phải bảo vệ mã nguồn bài tập lớn của mình.
- Sinh viên không hiểu mã nguồn do chính mình viết, trừ những phần mã được cung cấp sẵn trong chương trình khởi tạo. Sinh viên có thể tham khảo từ bất kỳ nguồn tài liệu nào, tuy nhiên phải đảm bảo rằng mình hiểu rõ ý nghĩa của tất cả những dòng lệnh mà mình viết. Trong trường hợp không hiểu rõ mã nguồn của nơi mình tham khảo, sinh viên

được đặc biệt cảnh báo là KHÔNG ĐƯỢC sử dụng mã nguồn này; thay vào đó nên sử dụng những gì đã được học để viết chương trình.
- Nộp nhầm bài của sinh viên khác trên tài khoản cá nhân của mình.
- Sinh viên sử dụng các công cụ AI trong quá trình làm bài tập lớn dẫn đến các mã nguồn giống nhau.

Trong trường hợp bị kết luận là gian lận, sinh viên sẽ bị điểm 0 cho toàn bộ môn học (không chỉ bài tập lớn).

## KHÔNG CHẤP NHẬN BẤT KỲ GIẢI THÍCH NÀO VÀ KHÔNG CÓ BẤT KỲ NGOẠI LỆ NÀO!

Sau mỗi bài tập lớn được nộp, sẽ có một số sinh viên được gọi phỏng vấn ngẫu nhiên để chứng minh rằng bài tập lớn vừa được nộp là do chính mình làm.

Một số quy định khác:

- Mọi quyết định của giảng viên phụ trách bài tập lớn là quyết định cuối cùng.
- Sinh viên không được cung cấp testcase sau khi chấm bài.
- Nội dung Bài tập lớn sẽ được Harmony với các câu hỏi trong bài kiểm tra cuối kì với nội dung tương tự.