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

ĐỒ THỊ VÀ MẠNG NƠRON NHIỀU LỚP

TP. HỒ CHÍ MINH, THÁNG 11/2024

1 Giới thiệu

1.1 Nội dung

Bài tập lớn số ba của môn Cấu trúc dữ liệu và giải thuật bao gồm hai nội dung chính sau đây:

Nội dung 1 (còn được gọi là TASK 1):

Nội dung này yêu cầu sinh viên phát triển cấu trúc dữ liệu Đồ thị (Graph):

- Tập tin cần hiện thực cho phần này là: `./include/graph/AbstractGraph.h`, `./include/graph/DGraphModel.h`, `./include/graph/UGraphModel.h`,
- Sinh viên đọc hướng dẫn thực hiện cho nội dung này trong **Phần 2**.

Nội dung 2 (còn được gọi là TASK 2):

Nội dung này yêu cầu sinh viên sử dụng cấu trúc dữ liệu Đồ thị để tính toán Backpropagation.

1.2 Cấu trúc dự án và Cách biên dịch dự án

Mã nguồn được cung cấp có cấu trúc tương tự như Bài tập lớn số hai.

Cách biên dịch dự án cũng tương tự như Bài tập lớn số hai. Sinh viên tham khảo Bài tập lớn số hai để biên dịch.

Các tập tin liên quan đến TASK-1 được chứa trong thư mục **./include/graph**.

2 TASK 1: Cấu trúc dữ liệu đồ thị

2.1 Tổng quan về cấu trúc dữ liệu đồ thị

Cấu trúc dữ liệu đồ thị (Graph) trong BTL này được thiết kế gồm các class như sau:

- class `IGraph`: class này định nghĩa một danh mục các APIs được hỗ trợ bởi đồ thị; các cách hiện thực đồ thị nào cũng sẽ phải hỗ trợ các APIs trong `IGraph`. `IGraph` là class cha của class `AbstractGraph`. Một số ý chi tiết:

  - `IGraph` sử dụng **template** để tham số hoá kiểu dữ liệu phần tử, và cho phép chứa trong đồ thị với kiểu bất kỳ.
  - Tất cả các APIs trong `IGraph` để ở dạng 'pure virtual method; nghĩa là các class kế thừa từ `IGraph` cần phải override tất cả các phương thức này và phương thức dạng này sẽ hỗ trợ liên kết động (tính đa hình).

- class `AbstractGraph`: Là một class trừu tượng (abstract class) kế thừa từ class interface IGraph, được thiết kế nhằm hiện thực hóa một phần giao diện của class IGraph và cung cấp các thành phần cơ bản để triển khai các mô hình đồ thị chi tiết hơn (ví dụ: đồ thị vô hướng, đồ thị có hướng). `AbstractGraph` sử dụng danh sách kề (adjacency list) để lưu trữ cấu trúc đồ thị.
- class `DGraphModel`: class DGraphModel là một hiện thực cụ thể của đồ thị có hướng (Directed Graph) dựa trên abstract class AbstractGraph. class này cung cấp các phương thức để thêm, xóa và quản lý các đỉnh và cạnh trong một đồ thị có hướng.
- class `UGraphModel`: class UGraphModel là một hiện thực cụ thể của đồ thị vô hướng (Undirected Graph) dựa trên abstract class AbstractGraph. class này cung cấp các phương thức để thêm, xóa và quản lý các đỉnh và cạnh trong một đồ thị vô hướng.

```
template<class T>
class IGraph{
public:
    virtual ~IGraph(){};
    virtual void add(T vertex)=0;
    virtual void remove(T vertex)=0;
    virtual bool contains(T vertex)=0;
    virtual void connect(T from, T to, float weight=0)=0;
    virtual void disconnect(T from, T to)=0;
    virtual bool connected(T from, T to)=0;
    virtual float weight(T from, T to)=0;
    virtual DLinkedList<T> getOutwardEdges(T from)=0;
    virtual DLinkedList<T> getInwardEdges(T to)=0;
    virtual int size()=0;
    virtual bool empty()=0;
    virtual void clear()=0;
    virtual int inDegree(T vertex)=0;
    virtual int outDegree(T vertex)=0;
    virtual DLinkedList<T> vertices()=0;
    virtual string toString()=0;
};
```

Listing 1: `IGraph<T>`: class trừu tượng định nghĩa APIs cho đồ thị

Dưới đây là mô tả cho từng pure virtual method của `IGraph`:

- `virtual void add(T vertex) = 0;`
  - Thêm một đỉnh mới vào đồ thị.
  - **Tham số**:
    - `T vertex` — Đỉnh cần thêm vào.
- `virtual void remove(T vertex) = 0;`
  - Xóa đỉnh `vertex` và tất cả các cạnh liên quan đến đỉnh này.
  - **Tham số**:
    - `T vertex` — Đỉnh cần xóa.
- `virtual bool contains(T vertex) = 0;`
  - Kiểm tra xem một đỉnh có tồn tại trong đồ thị hay không.
  - **Tham số**:
    - `T vertex` — Đỉnh cần kiểm tra.
  - **Giá trị trả về**: `true` nếu đỉnh tồn tại, ngược lại `false`.
- `virtual void connect(T from, T to, float weight = 0) = 0;`
  - Kết nối hai đỉnh `from` và `to` bằng một cạnh, với trọng số mặc định là `0`.
  - **Tham số**:
    - `T from` — Đỉnh bắt đầu.
    - `T to` — Đỉnh kết thúc.
    - `float weight` — Trọng số của cạnh (mặc định là 0).
- `virtual void disconnect(T from, T to) = 0;`
  - Xóa cạnh giữa hai đỉnh `from` và `to`.
  - **Tham số**:
    - `T from` — Đỉnh bắt đầu.
    - `T to` — Đỉnh kết thúc.
- `virtual bool connected(T from, T to) = 0;`
  - Kiểm tra xem hai đỉnh `from` và `to` có được kết nối bằng một cạnh hay không.
  - **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`.

- `virtual float weight(T from, T to) = 0;`
  - Lấy trọng số của cạnh giữa hai đỉnh `from` và `to`.
  - **Tham số**:
    - `T from` — Đỉnh bắt đầu.
    - `T to` — Đỉnh kết thúc.
  - **Giá trị trả về**: Trọng số của cạnh.
- `virtual DLinkedList<T> getOutwardEdges(T from) = 0;`
  - Lấy danh sách các đỉnh mà `from` có cạnh đi tới.
  - **Tham số**:
    - `T from` — Đỉnh xuất phát.
  - **Giá trị trả về**: `DLinkedList<T>` chứa các đỉnh đích.
- `virtual DLinkedList<T> getInwardEdges(T to) = 0;`
  - Lấy danh sách các đỉnh có cạnh đi vào `to`.
  - **Tham số**:
    - `T to` — Đỉnh đích.
  - **Giá trị trả về**: `DLinkedList<T>` chứa các đỉnh nguồn.
- `virtual int size() = 0;`
  - Trả về số lượng đỉnh trong đồ thị.
  - **Giá trị trả về**: Số lượng đỉnh.
- `virtual bool empty() = 0;`
  - Kiểm tra xem đồ thị có rỗng không.
  - **Giá trị trả về**: `true` nếu đồ thị rỗng, ngược lại `false`.
- `virtual void clear() = 0;`
  - Xóa tất cả các đỉnh và cạnh trong đồ thị, đưa đồ thị về trạng thái rỗng.
- `virtual int inDegree(T vertex) = 0;`
  - Tính số lượng cạnh đi vào đỉnh `vertex`.
  - **Tham số**:
    - `T vertex` — Đỉnh cần tính.
  - **Giá trị trả về**: Số lượng cạnh đi vào đỉnh.
- `virtual int outDegree(T vertex) = 0;`
  - Tính số lượng cạnh đi ra từ đỉnh `vertex`.

  - **Tham số**:
    * `T vertex` — Đỉnh cần tính.
  - **Giá trị trả về**: Số lượng cạnh đi ra từ đỉnh.

- `virtual DLinkedList<T> vertices() = 0;`
  - Trả về danh sách tất cả các đỉnh trong đồ thị.
  - **Giá trị trả về**: `DLinkedList<T>` chứa các đỉnh.

- `virtual string toString() = 0;`
  - Trả về chuỗi mô tả đồ thị, bao gồm danh sách đỉnh và các cạnh.
  - **Giá trị trả về**: Chuỗi biểu diễn đồ thị.

2.2 Abstract Graph

`AbstractGraph<T>` là một class cơ sở trừu tượng dùng để hiện thực các loại đồ thị (graph) như đồ thị có hướng, vô hướng, có trọng số hoặc không trọng số. Các đỉnh trong đồ thị được biểu diễn bằng kiểu dữ liệu `T`, và các cạnh được tổ chức dưới dạng danh sách kề, hỗ trợ quản lý kết nối giữa các đỉnh.

class `AbstractGraph<T>` định nghĩa các thuộc tính và phương thức cơ bản để thao tác với đồ thị, bao gồm thêm hoặc xóa đỉnh (`add`, `remove`), thêm hoặc xóa cạnh (`connect`, `disconnect`), và truy vấn thông tin như số đỉnh (`size`), kiểm tra đồ thị rỗng (`empty`), hoặc tính bậc vào và bậc ra của một đỉnh (`inDegree`, `outDegree`). Các cạnh có thể có trọng số, với giá trị trọng số được quản lý thông qua phương thức `weight`.

Tuy nhiên, việc hiện thực các phương thức `connect`, `disconnect` và `remove` sẽ phụ thuộc vào đặc điểm của đồ thị, ví dụ: trong đồ thị có hướng, cạnh sẽ được tạo hoặc xóa theo một chiều nhất định, trong khi với đồ thị vô hướng, cạnh sẽ được áp dụng hai chiều giữa hai đỉnh.

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`). Cấu trúc này hỗ trợ việc quản lý các cạnh và truy cập nhanh các thuộc tính cần thiết khi thao tác trên đồ thị.

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).

Cuối cùng, `Iterator` được cung cấp như một công cụ hỗ trợ duyệt qua các đỉnh hoặc cạnh trong đồ thị một cách tuần tự. Nó giúp đơn giản hóa việc duyệt đồ thị trong các ứng dụng khác nhau, ví dụ như tìm kiếm, sắp xếp, hoặc kiểm tra tính liên thông.

Những định nghĩa và hiện thực chi tiết có thể được tìm thấy trong tập tin **AbstractGraph.h**, trong thư mục **/include/graph**.

```
template<class T>
class AbstractGraph: public IGraph<T> {
public:
    class Edge;          // Forward declaration
    class VertexNode;    // Forward declaration
    class Iterator;      // Forward declaration

protected:
    DLinkedList<VertexNode*> nodeList; // List of nodes (adjacency list)
    bool (*vertexEQ)(T&, T&);          // Function pointer to compare vertices
    string (*vertex2str)(T&);          // Function pointer to convert vertices to string

    VertexNode* getVertexNode(T& vertex);
    string vertex2Str(VertexNode& node);
    string edge2Str(Edge& edge);

public:
    AbstractGraph(bool (*vertexEQ)(T&, T&) = 0, string (*vertex2str)(T&) = 0);
    virtual ~AbstractGraph();

    typedef bool (*vertexEQFunc)(T&, T&);
    typedef string (*vertex2strFunc)(T&);
    vertexEQFunc getVertexEQ();
    vertex2strFunc getVertex2Str();

    // IGraph API
    virtual void connect(T from, T to, float weight = 0) = 0;
    virtual void disconnect(T from, T to) = 0;
    virtual void remove(T vertex) = 0;

    virtual void add(T vertex);
    virtual bool contains(T vertex);
    virtual float weight(T from, T to);
    virtual DLinkedList<T> getOutwardEdges(T from);
    virtual DLinkedList<T> getInwardEdges(T to);
    virtual int size();
    virtual bool empty();
```

```
38      virtual void clear();
39      virtual int inDegree(T vertex);
40      virtual int outDegree(T vertex);
41      virtual DLinkedList<T> vertices();
42      virtual bool connected(T from, T to);
43      virtual string toString();
44
45      Iterator begin();
46      Iterator end();
47
48      void println();
49
50  public:
51      class VertexNode {
52      private:
53          T vertex;
54          int inDegree_, outDegree_;
55          DLinkedList<Edge*> adList;
56
57      public:
58          VertexNode();
59          VertexNode(T vertex, bool (*vertexEQ)(T&, T&), string (*vertex2str)(
        T&));
60
61          T& getVertex();
62          void connect(VertexNode* to, float weight = 0);
63          DLinkedList<T> getOutwardEdges();
64          Edge* getEdge(VertexNode* to);
65          bool equals(VertexNode* node);
66          void removeTo(VertexNode* to);
67          int inDegree();
68          int outDegree();
69          string toString();
70      };
71
72      class Edge {
73      private:
74          VertexNode* from;
75          VertexNode* to;
76          float weight;
77
78      public:
79          Edge();
```

```
80          Edge(VertexNode* from, VertexNode* to, float weight = 0);
81          bool equals(Edge* edge);
82          static bool edgeEQ(Edge*& edge1, Edge*& edge2);
83          string toString();
84      };
85
86      class Iterator {
87      private:
88          typename DLinkedList<VertexNode*>::Iterator nodeIt;
89
90      public:
91          Iterator(AbstractGraph<T>* pGraph = 0, bool begin = true);
92          Iterator& operator=(const Iterator& iterator);
93          T& operator*();
94          bool operator!=(const Iterator& iterator);
95          Iterator& operator++();
96          Iterator operator++(int);
97      };
98  };
```

Listing 2: AbstractGraph<T>: Đồ thị hiện thực bằng danh sách kề

2.2.1 Edge

Dưới đây là mô tả chi tiết về class `Edge`:

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.

2.2.2 VertexNode

Phần này cung cấp mô tả chi tiết về class `VertexNode`:

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).
   - `DLinkedList<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 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()`: Hàm khởi tạo mặc định. Khởi tạo danh sách kề với các hàm dùng để quản lý bộ nhớ và kiểm tra tính bằng nhau.
   - `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 khác `to` bằng cách tạo một cạnh với trọng số (mặc định là 0).
  - **Ngoại lệ**: Không có.
- `DLinkedList<T> getOutwardEdges()`:
  - **Chức năng**: Trả về danh sách các cạnh đi ra từ đỉnh hiện tại.
  - **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.

2.2.3 AbstractGraph

Dưới đây là mô tả chi tiết cho class `AbstractGraph`:

1. **Các thuộc tính:**
   - `DLinkedList<VertexNode*> nodeList`: Danh sách liên kết đôi chứa các đỉnh của đồ thị, được sử dụng để lưu trữ tất cả các đỉnh của đồ thị dưới dạng các nút `VertexNode`.
   - `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 một đỉnh thành chuỗi, được sử dụng để biểu diễn đỉnh dưới dạng chuỗi.
2. **Hàm khởi tạo và hàm hủy:**
   - `AbstractGraph(bool (*vertexEQ)(T&, T&)=0, string (*vertex2str)(T&)=0)`: Hàm khởi tạo class `AbstractGraph` 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`.
   - `AbstractGraph()`: Hàm hủy, giải phóng tài nguyên của đồ thị, xóa các đỉnh và các cạnh liên quan.
3. **Các phương thức:**
   - **VertexNode* 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& 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.
   - **virtual void add(T vertex)**
     - **Chức năng**: Thêm một đỉnh mới vào đồ thị.
     - **Ngoại lệ**: Không có.
   - **virtual bool contains(T vertex)**
     - **Chức năng**: Kiểm tra xem đồ thị có chứa đỉnh `vertex` hay không.
     - **Ngoại lệ**: Không có.

- **virtual 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.
- **virtual DLinkedList<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`.
- **virtual DLinkedList<T> getInwardEdges(T to)**
  - **Chức năng**: Lấy danh sách các cạnh đi vào đỉnh `to`.
  - **Hướng hiện thực**:
    (a) Tìm `VertexNode` tương ứng với đỉnh `to` thông qua hàm `getVertexNode(to)`.
    (b) Nếu không tìm thấy đỉnh `to`, ném ngoại lệ `VertexNotFoundException` với thông tin về đỉnh.
    (c) Khởi tạo một danh sách trống `list` để lưu các đỉnh có cạnh đi vào `to`.
    (d) Duyệt qua tất cả các đỉnh trong danh sách `nodeList`.
    (e) Đối với mỗi đỉnh `node`, duyệt qua các cạnh kề của đỉnh đó trong `adList`.
    (f) Nếu cạnh có đỉnh đích là `to`, thêm đỉnh nguồn của cạnh vào danh sách `list`.
    (g) Tiếp tục duyệt qua tất cả các đỉnh và cạnh cho đến khi hết.
  - **Ngoại lệ**:
    * `VertexNotFoundException`: Nếu không tìm thấy đỉnh `to`.
- **virtual int size()**
  - **Chức năng**: Trả về số lượng đỉnh trong đồ thị.
  - **Ngoại lệ**: Không có.
- **virtual bool empty()**
  - **Chức năng**: Kiểm tra xem đồ thị có rỗng hay không.
  - **Ngoại lệ**: Không có.
- **virtual void clear()**

  - **Chức năng**: Xóa tất cả các đỉnh và cạnh trong đồ thị.
  - **Ngoại lệ**: Không có.

- **virtual 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`.
- **virtual 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`.
- **virtual DLinkedList<T> vertices()**
  - **Chức năng**: Trả về danh sách tất cả các đỉnh trong đồ thị.
  - **Ngoại lệ**: Không có.
- **virtual bool connected(T from, T to)**
  - **Chức năng**: Kiểm tra xem có cạnh giữa đỉnh `from` và đỉnh `to`.
  - **Ngoại lệ**: Ném ngoại lệ `VertexNotFoundException` nếu không tìm thấy đỉnh.
- **void println()**
  - **Chức năng**: In thông tin biểu diễn của đồ thị ra màn hình.
  - **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.
- **Iterator begin()**
  - **Chức năng**: Trả về iterator để duyệt qua các đỉnh trong đồ thị bắt đầu từ đỉnh đầu tiên.
  - **Ngoại lệ**: Không có.
- **Iterator end()**
  - **Chức năng**: Trả về iterator để duyệt qua các đỉnh trong đồ thị kết thúc tại đỉnh cuối cùng.
  - **Ngoại lệ**: Không có.

2.3 Directed Graph

class `DGraphModel<T>` là một class kế thừa từ `AbstractGraph<T>` nhằm hiện thực một mô hình đồ thị có hướng (Directed Graph). class này cung cấp các phương thức để thao tác với đồ thị có hướng, bao gồm việc kết nối các đỉnh (`connect`), ngắt kết nối giữa các đỉnh (`disconnect`), và xóa đỉnh khỏi đồ thị (`remove`). Các phương thức này đảm bảo rằng các kết nối giữa các đỉnh trong đồ thị được quản lý theo chiều hướng của đồ thị có hướng.

class `DGraphModel<T>` cũng cung cấp phương thức tĩnh `create`, cho phép tạo một đối tượng `DGraphModel<T>` mới từ một danh sách các đỉnh và cạnh đã được xác định trước, hỗ trợ việc khởi tạo đồ thị từ các dữ liệu đầu vào.

```
template<class T>
class DGraphModel: public AbstractGraph<T> {
private:
public:
    DGraphModel(
            bool (*vertexEQ)(T&, T&),
            string (*vertex2str)(T&) );

    void connect(T from, T to, float weight = 0);
    void disconnect(T from, T to);
    void remove(T vertex);

    static DGraphModel<T>* create(
            T* vertices, int nvertices, Edge<T>* edges, int nedges,
            bool (*vertexEQ)(T&, T&),
            string (*vertex2str)(T&));
};
```

Listing 3: DGraphModel<T>: Đồ thị có hướng

Dưới đây là mô tả chi tiết cho class `DGraphModel`:

1. **Các hàm khởi tạo:**
   - `DGraphModel(bool (*vertexEQ)(T&, T&), string (*vertex2str)(T&))`:
     - **Chức năng**: Khởi tạo một đồ thị có hướng với hai tham số hàm:
       - `vertexEQ`: Hàm so sánh hai đỉnh để kiểm tra tính bằng nhau.
       - `vertex2str`: Hàm chuyển đỉnh sang chuỗi để hiển thị.
     - **Ngoại lệ**: Không có.
2. **Các phương thức:**

- **void connect(T from, T to, float weight = 0)**
  - **Chức năng**: Thêm một cạnh có hướng từ đỉnh `from` đến đỉnh `to` với trọng số `weight` (mặc định là 0).
  - **Hướng hiện thực**:
    (a) Lấy các `VertexNode` tương ứng với các đỉnh `from` và `to`.
    (b) Nếu bất kỳ đỉnh nào không tồn tại, ném ngoại lệ `VertexNotFoundException`.
    (c) Kết nối đỉnh `from` đến `to` bằng cách thêm cạnh mới với trọng số `weight`.
  - **Ngoại lệ**:
    * Ném ngoại lệ `VertexNotFoundException` nếu một trong hai đỉnh không tồn tại.
- **void disconnect(T from, T to)**
  - **Chức năng**: Xóa cạnh từ đỉnh `from` đến đỉnh `to`.
  - **Hướng hiện thực**:
    (a) Lấy các `VertexNode` tương ứng với các đỉnh `from` và `to`.
    (b) Nếu bất kỳ đỉnh nào không tồn tại, ném ngoại lệ `VertexNotFoundException`.
    (c) Lấy cạnh từ `from` đến `to`. Nếu cạnh không tồn tại, ném ngoại lệ `EdgeNotFoundExceptior`
    (d) Xóa cạnh này khỏi đồ thị.
  - **Ngoại lệ**:
    * `VertexNotFoundException`: Nếu một trong hai đỉnh không tồn tại.
    * `EdgeNotFoundException`: Nếu không tồn tại cạnh giữa hai đỉnh.
- **void remove(T vertex)**
  - **Chức năng**: Xóa một đỉnh và tất cả các cạnh liên quan đến đỉnh đó.
  - **Hướng hiện thực**:
    (a) Lấy `VertexNode` tương ứng với đỉnh cần xóa.
    (b) Nếu đỉnh không tồn tại, ném ngoại lệ `VertexNotFoundException`.
    (c) Duyệt qua danh sách tất cả các đỉnh trong đồ thị, xóa các cạnh kết nối đến hoặc từ đỉnh cần xóa.
    (d) Xóa đỉnh khỏi danh sách các đỉnh của đồ thị.
  - **Ngoại lệ**:
    * `VertexNotFoundException`: Nếu đỉnh không tồn tại.
- **static DGraphModel<T>* create(T* vertices, int nvertices, Edge<T>* edges, int nedges, bool (*vertexEQ)(T&, T&), string (*vertex2str)(T&))**
  - **Chức năng**: Tạo một đồ thị có hướng mới từ danh sách đỉnh và cạnh.
  - **Hướng hiện thực**:

(a) Khởi tạo một đối tượng `DGraphModel` mới.
(b) Thêm tất cả các đỉnh trong `vertices` vào đồ thị.
(c) Thêm tất cả các cạnh trong `edges` vào đồ thị.
(d) Trả về con trỏ đến đồ thị được tạo.

– **Ngoại lệ**: Không có.

2.4 Undirected Graph

class `UGraphModel<T>` là một class kế thừa từ `AbstractGraph<T>` nhằm hiện thực một mô hình đồ thị vô hướng (Undirected Graph). class này cung cấp các phương thức để thao tác với đồ thị có hướng, bao gồm việc kết nối các đỉnh (`connect`), ngắt kết nối giữa các đỉnh (`disconnect`), và xóa đỉnh khỏi đồ thị (`remove`). Các phương thức này đảm bảo rằng các kết nối giữa các đỉnh trong đồ thị được quản lý theo chiều hướng của đồ thị vô hướng.

class `UGraphModel<T>` cũng cung cấp phương thức tĩnh `create`, cho phép tạo một đối tượng `UGraphModel<T>` mới từ một danh sách các đỉnh và cạnh đã được xác định trước, hỗ trợ việc khởi tạo đồ thị từ các dữ liệu đầu vào.

```
template<class T>
class UGraphModel: public AbstractGraph<T> {
private:
public:
    UGraphModel(
            bool (*vertexEQ)(T&, T&),
            string (*vertex2str)(T&) );

    void connect(T from, T to, float weight = 0);
    void disconnect(T from, T to);
    void remove(T vertex);

    static UGraphModel<T>* create(
            T* vertices, int nvertices, Edge<T>* edges, int nedges,
            bool (*vertexEQ)(T&, T&),
            string (*vertex2str)(T&));
};
```

Listing 4: UGraphModel<T>: Đồ thị vô hướng

Dưới đây là mô tả chi tiết cho class `UGraphModel`:

1. **Mô tả tổng quan:**

- `UGraphModel`: Là một class biểu diễn đồ thị vô hướng (Undirected Graph Model), kế thừa từ class `AbstractGraph<T>`. class này cung cấp các thao tác chính trên đồ thị vô hướng như thêm đỉnh, kết nối đỉnh, ngắt kết nối, và loại bỏ đỉnh.
- **Tham số mẫu (template)**: `T`, đại diện cho kiểu dữ liệu của các đỉnh trong đồ thị.

2. **Các hàm khởi tạo:**

- `UGraphModel(bool (*vertexEQ)(T&, T&), string (*vertex2str)(T&))`:
  - **Chức năng**: Khởi tạo một đồ thị vô hướng với hai tham số hàm:
    - `vertexEQ`: Hàm so sánh hai đỉnh để kiểm tra tính bằng nhau.
    - `vertex2str`: Hàm chuyển đỉnh sang chuỗi để hiển thị.
  - **Ngoại lệ**: Không có.

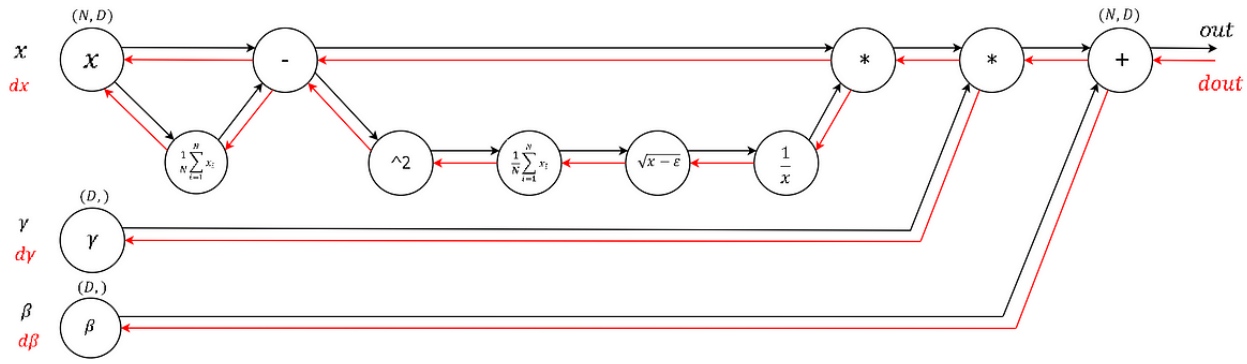
3. **Các phương thức:**

- **void connect(T from, T to, float weight = 0)**
  - **Chức năng**: Thêm một cạnh vô hướng giữa hai đỉnh `from` và `to` với trọng số `weight` (mặc định là 0).
  - **Hướng hiện thực**:
    (a) Lấy các `VertexNode` tương ứng với `from` và `to`.
    (b) Nếu bất kỳ đỉnh nào không tồn tại, ném ngoại lệ `VertexNotFoundException`.
    (c) Nếu hai đỉnh `from` và `to` trùng nhau, thêm một cạnh tự vòng (*self-loop*).
    (d) Nếu hai đỉnh khác nhau, thêm hai cạnh (một từ `from` đến `to` và ngược lại).
  - **Ngoại lệ**:
    - `VertexNotFoundException`: Nếu một trong hai đỉnh không tồn tại.
- **void disconnect(T from, T to)**
  - **Chức năng**: Xóa cạnh vô hướng giữa hai đỉnh `from` và `to`.
  - **Hướng hiện thực**:
    (a) Lấy các `VertexNode` tương ứng với `from` và `to`.
    (b) Nếu bất kỳ đỉnh nào không tồn tại, ném ngoại lệ `VertexNotFoundException`.
    (c) Lấy cạnh từ `from` đến `to`. Nếu cạnh không tồn tại, ném ngoại lệ `EdgeNotFoundExceptior`
    (d) Nếu hai đỉnh trùng nhau, xóa cạnh tự vòng (*self-loop*).
    (e) Nếu hai đỉnh khác nhau, xóa cả hai cạnh (một từ `from` đến `to` và ngược lại).
  - **Ngoại lệ**:
    - `VertexNotFoundException`: Nếu một trong hai đỉnh không tồn tại.
    - `EdgeNotFoundException`: Nếu không tồn tại cạnh giữa hai đỉnh.
- **void remove(T vertex)**

- **Chức năng**: Xóa một đỉnh và tất cả các cạnh liên quan đến đỉnh đó.
- **Hướng hiện thực**:
  (a) Lấy `VertexNode` tương ứng với đỉnh cần xóa.
  (b) Nếu đỉnh không tồn tại, ném ngoại lệ `VertexNotFoundException`.
  (c) Duyệt qua danh sách các đỉnh trong đồ thị, xóa tất cả các cạnh kết nối đến hoặc từ đỉnh cần xóa.
  (d) Xóa đỉnh khỏi danh sách các đỉnh của đồ thị.
- **Ngoại lệ**:
  * `VertexNotFoundException`: Nếu đỉnh không tồn tại.

- **static UGraphModel<T>* create(T* vertices, int nvertices, Edge<T>* edges, int nedges, bool (*vertexEQ)(T&, T&), string (*vertex2str)(T&))**
  - **Chức năng**: Tạo một đồ thị vô hướng mới từ danh sách đỉnh và cạnh.
  - **Hướng hiện thực**:
    (a) Khởi tạo một đối tượng `UGraphModel` mới.
    (b) Thêm tất cả các đỉnh trong `vertices` vào đồ thị.
    (c) Thêm tất cả các cạnh trong `edges` vào đồ thị.
    (d) Trả về con trỏ đến đồ thị được tạo.
  - **Ngoại lệ**: Không có.

3 TASK 2: Sắp xếp Topo



Hình 1: Biểu diễn topo của quá trình tính toán Backpropagation

Backpropagation là một thuật toán quan trọng trong quá trình huấn luyện mạng nơ-ron, được sử dụng để tối ưu hóa các tham số thông qua việc lan truyền gradient từ đầu ra về các lớp trước đó. Quá trình này bao gồm hai bước chính:

- **Forward Pass**: Tính toán giá trị đầu ra dựa trên các trọng số hiện tại.
- **Backward Pass**: Sử dụng đạo hàm để tính gradient của hàm mất mát theo các trọng số, sau đó cập nhật trọng số thông qua một thuật toán tối ưu, chẳng hạn như Gradient Descent.

Trong bài tập lớn (BTL) này, sinh viên được yêu cầu hiện thực một bước quan trọng của Backpropagation: **Topological Sorting (Topo Sort)**. Đây là phương pháp sắp xếp các nút của một đồ thị có hướng và không có chu trình (*Directed Acyclic Graph - DAG*) theo thứ tự mà mọi cạnh (u, v) luôn đảm bảo nút u đứng trước nút v.

Topo Sort được sử dụng trong Backpropagation để xác định thứ tự cập nhật gradient khi mạng nơ-ron có cấu trúc phức tạp, chẳng hạn như mạng với nhiều nhánh hoặc đồ thị không tuần tự.

3.1 TopoSorter

Class `TopoSorter<T>` là một class hỗ trợ việc sắp xếp topo (Topological Sorting) trên các đồ thị có hướng không chứa chu trình (*Directed Acyclic Graph - DAG*). Class này cho phép người dùng sắp xếp các đỉnh của đồ thị theo thứ tự mà mọi cạnh (u, v) đảm bảo đỉnh u đứng trước v.

Class `TopoSorter<T>` cung cấp hai phương pháp chính để sắp xếp topo: `dfsSort`, sử dụng thuật toán tìm kiếm theo chiều sâu (DFS), và `bfsSort`, sử dụng thuật toán tìm kiếm theo chiều rộng (BFS).

```
template<class T>
class TopoSorter {
public:
    static int DFS;
    static int BFS;

protected:
    DGraphModel<T>* graph;
    int (*hash_code)(T&, int);

public:
    TopoSorter(DGraphModel<T>* graph, int (*hash_code)(T&, int) = 0);
    DLinkedList<T> sort(int mode = 0, bool sorted = true);
    DLinkedList<T> bfsSort(bool sorted = true);
    DLinkedList<T> dfsSort(bool sorted = true);

```

```
17 protected:
18     //Helping functions
19     XHashMap<T, int> vertex2inDegree(int (*hash)(T&, int));
20     XHashMap<T, int> vertex2outDegree(int (*hash)(T&, int));
21     DLinkedList<T> listOfZeroInDegrees();
22 };
```

Listing 5: TopoSorter<T>

Dưới đây là mô tả chi tiết cho class `TopoSorter`:

1. **Các thuộc tính:**
   - `DGraphModel<T>* graph`: Con trỏ tới đối tượng đồ thị có hướng (directed graph) chứa các đỉnh và cạnh được sắp xếp, là cơ sở để thực hiện sắp xếp topo.
   - `int (*hash_code)(T&, int)`: Con trỏ tới hàm băm được sử dụng để ánh xạ các đỉnh của đồ thị thành các giá trị số nguyên, phục vụ các cấu trúc dữ liệu băm. Sinh viên được khuyến khích sử dụng hàm băm để tối ưu giải thuật hiện thực.
   - `static int DFS`: Giá trị hằng (0) đại diện cho chế độ sắp xếp Topo dựa trên thuật toán DFS.
   - `static int BFS`: Giá trị hằng (1) đại diện cho chế độ sắp xếp Topo dựa trên thuật toán BFS.
2. **Hàm khởi tạo:**
   - `TopoSorter(DGraphModel<T>* graph, int (*hash_code)(T&, int)=0)`:
     - **Chức năng**: Khởi tạo đối tượng `TopoSorter` với đồ thị và (tuỳ chọn) hàm băm. Nếu không cung cấp hàm băm, sử dụng giá trị mặc định là `nullptr`.
     - **Tham số**:
       * `graph`: Con trỏ tới đồ thị hướng cần sắp xếp topo.
       * `hash_code`: Hàm băm để ánh xạ các đỉnh.
3. **Các phương thức:**
   - `DLinkedList<T> sort(int mode=0, bool sorted=true)`
     - **Chức năng**: Thực hiện sắp xếp topo bằng thuật toán DFS hoặc BFS, tùy thuộc vào giá trị `mode`.
     - **Tham số**:
       * `mode`: Chế độ sắp xếp, mặc định là `DFS`.
       * `sorted`: Nếu là `true`, danh sách đỉnh được sắp xếp theo thứ tự tăng dần để phục vụ mục đích giáo dục.

  - **Trả về**: Danh sách liên kết đôi chứa các đỉnh theo thứ tự topo.
- `DLinkedList<T> bfsSort(bool sorted=true)`
  - **Chức năng**: Thực hiện sắp xếp topo bằng thuật toán BFS.
  - **Tham số**:
    * `sorted`: Nếu là `true`, sắp xếp danh sách các đỉnh theo thứ tự tăng dần trước khi xử lý.
  - **Trả về**: Danh sách liên kết đôi chứa các đỉnh theo thứ tự topo.
  - **Một số gợi ý**:
    * Sử dụng helping functions: vertex2inDegree và listOfZeroInDegrees để hiện thực giải thuật
    * Việc sắp xếp danh sách đỉnh được khuyến khích sử dụng Merge Sort để đảm bảo tính duy nhất của kết quả sắp xếp, được thiết kế sẵn từ class DLinkedListSE.
- `DLinkedList<T> bfsSort(bool sorted=true)`
  - **Chức năng**: Thực hiện sắp xếp topo bằng thuật toán BFS.
  - **Tham số**:
    * `sorted`: Nếu là `true`, sắp xếp danh sách các đỉnh theo thứ tự tăng dần trước khi xử lý.
  - **Trả về**: Danh sách liên kết đôi chứa các đỉnh theo thứ tự topo.
  - **Một số gợi ý**:
    * Sử dụng helping functions: `vertex2outDegree` và `listOfZeroInDegrees` để hiện thực giải thuật
    * Việc sắp xếp danh sách đỉnh được khuyến khích sử dụng Merge Sort để đảm bảo tính duy nhất của kết quả sắp xếp, được thiết kế sẵn từ class `DLinkedListSE`.

4. **Các phương thức hỗ trợ:**
   - `XHashMap<T, int> vertex2inDegree(int (*hash)(T&, int))`
     - **Chức năng**: Tạo một bảng băm lưu trữ bậc vào của tất cả các đỉnh trong đồ thị.
     - **Tham số**:
       * `hash`: Hàm băm để ánh xạ các đỉnh.
     - **Trả về**: Một đối tượng `XHashMap` với đỉnh làm khóa và bậc vào làm giá trị.
   - `XHashMap<T, int> vertex2outDegree(int (*hash)(T&, int))`

  - **Chức năng**: Tạo một bảng băm lưu trữ bậc ra của tất cả các đỉnh trong đồ thị.
  - **Tham số**:
    * `hash`: Hàm băm để ánh xạ các đỉnh.
  - **Trả về**: Một đối tượng `XHashMap` với đỉnh làm khóa và bậc ra làm giá trị.
- `DLinkedList<T> listOfZeroInDegrees()`
  - **Chức năng**: Tạo danh sách các đỉnh có bậc vào bằng 0.
  - **Trả về**: Danh sách liên kết đôi chứa các đỉnh có bậc vào bằng 0.

3.2 Các Class hỗ trợ

Để hỗ trợ việc hiện thực class `TopoSorter`, các class `Stack`, `Queue`, `DLinkedListSE` được cung cấp sẵn prototype và hướng dẫn hiện thực sẵn trong mã nguồn. Sinh viên được khuyến khích sử dụng các class này để hiện thực cho `TopoSorter`. Nếu không hiện thực, sinh viên giữ nguyên các file này và không sửa đổi.

4 Nộp bài

Sinh viên nộp bài trước thời hạn được đưa ra trong đường dẫn "Assignment 3 - Submission". Có một số testcase đơn giản được sử dụng để kiểm tra bài làm của sinh viên nhằm đảm bảo rằng kết quả của sinh viên có thể biên dịch và chạy được. Sinh viên có thể nộp bài bao nhiêu lần tùy ý nhưng chỉ có bài nộp cuối cùng được tính điểm. Vì hệ thống không thể chịu tải khi quá nhiều sinh viên nộp bài cùng một lúc, vì vậy sinh viên nên nộp bài càng sớm càng tốt. Sinh viên sẽ tự chịu rủi ro nếu nộp bài sát hạn chót. Khi quá thời hạn nộp bài, hệ thống sẽ đóng nên sinh viên sẽ không thể nộp nữa. Bài nộp qua các phương thức khác đều không được chấp nhận.

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. Giả sử điểm BTL mà sinh viên đạt được là **a** (theo thang điểm 10), tổng điểm các câu hỏi Harmony **b** (theo thang điểm 5). Gọi **x** là điểm của BTL sau khi Harmony, cũng là điểm BTL cuối cùng của sinh viên. Các câu hỏi cuối kì sẽ được Harmony với 50% điểm của BTL theo công thức sau:

- Nếu $a = 0$ hoặc $b = 0$ thì $x = 0$

- Nếu a và b đều khác 0 thì

$$x = \frac{a}{2} + HARM(\frac{a}{2}, b)$$

  Trong đó:

$$HARM(x, y) = \frac{2xy}{x + y}$$

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 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.

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.