VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

DATA STRUCTURES AND ALGORITHMS - CO2003

Assignment 3

Developing Graph Data Structures and Artificial Neural Networks

---

Ho Chi Minh City, 11/2024

1 Introduction

1.1 Content

The third major assignment for the course *Data Structures and Algorithms* consists of the following two main tasks:

Task 1 (also referred to as TASK 1):

This task requires students to develop a Graph data structure:

- The files to be implemented for this task are: `./include/graph/AbstractGraph.h`, `./include/graph/DGraphModel.h`, `./include/graph/UGraphModel.h`.
- Students should refer to the instructions for this task in **Section 2**.

Task 2 (also referred to as TASK 2):

This task requires students to utilize the Graph data structure to compute Forward Propagation and Backward Propagation.

1.2 Project Structure and Compilation

The provided source code structure is similar to that of the second major assignment.

The project compilation process is also similar to that of the second major assignment. Students should refer to the second major assignment for compilation instructions.

The files related to TASK-1 are located in the folder **./include/graph**.

2 TASK 1: Graph Data Structure

2.1 Overview of the Graph Data Structure

The Graph data structure in this assignment is designed with the following classes:

- `IGraph` class: This class defines a set of APIs supported by the graph. Any graph implementation must support the APIs in `IGraph`. `IGraph` serves as the parent class for the `AbstractGraph` class. Key details include:

  - `IGraph` uses a **template** to parameterize the element type, allowing the graph to hold elements of any type.
  - All APIs in `IGraph` are defined as "pure virtual methods," meaning that classes inheriting from `IGraph` must override all these methods. Such methods support dynamic linking (polymorphism).

- `AbstractGraph` class: An abstract class inheriting from the `IGraph` interface. It is designed to partially implement the `IGraph` interface and provide the fundamental components for implementing detailed graph models (e.g., undirected graphs, directed graphs). `AbstractGraph` uses an adjacency list to store the graph structure.
- `DGraphModel` class: This class provides a concrete implementation of a directed graph (Directed Graph) based on the `AbstractGraph` class. It includes methods for adding, removing, and managing vertices and edges in a directed graph.
- `UGraphModel` class: This class provides a concrete implementation of an undirected graph (Undirected Graph) based on the `AbstractGraph` class. It includes methods for adding, removing, and managing vertices and edges in an undirected graph.

```
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>`: Abstract class defining APIs for the graph

Below is the description of each pure virtual method in `IGraph`:

- `virtual void add(T vertex) = 0;`
  - Adds a new vertex to the graph.
  - **Parameter**:
    - `T vertex` — The vertex to be added.
- `virtual void remove(T vertex) = 0;`
  - Removes the vertex `vertex` and all its associated edges from the graph.
  - **Parameter**:
    - `T vertex` — The vertex to be removed.
- `virtual bool contains(T vertex) = 0;`
  - Checks if a vertex exists in the graph.
  - **Parameter**:
    - `T vertex` — The vertex to check.
  - **Return value**: `true` if the vertex exists, otherwise `false`.
- `virtual void connect(T from, T to, float weight = 0) = 0;`
  - Connects two vertices `from` and `to` with an edge, with a default weight of `0`.
  - **Parameters**:
    - `T from` — The starting vertex.
    - `T to` — The ending vertex.
    - `float weight` — The weight of the edge (default is 0).
- `virtual void disconnect(T from, T to) = 0;`
  - Removes the edge between two vertices `from` and `to`.
  - **Parameters**:
    - `T from` — The starting vertex.
    - `T to` — The ending vertex.
- `virtual bool connected(T from, T to) = 0;`
  - Checks if two vertices `from` and `to` are connected by an edge.
  - **Parameters**:
    - `T from` — The starting vertex.
    - `T to` — The ending vertex.
  - **Return value**: `true` if the vertices are connected, otherwise `false`.
- `virtual float weight(T from, T to) = 0;`
  - Retrieves the weight of the edge between two vertices `from` and `to`.

  - **Parameters**:
    - `T from` — The starting vertex.
    - `T to` — The ending vertex.
  - **Return value**: The weight of the edge.

- `virtual DLinkedList<T> getOutwardEdges(T from) = 0;`
  - Retrieves a list of vertices that the vertex `from` has edges pointing to.
  - **Parameter**:
    - `T from` — The starting vertex.
  - **Return value**: A `DLinkedList<T>` containing the destination vertices.

- `virtual DLinkedList<T> getInwardEdges(T to) = 0;`
  - Retrieves a list of vertices that have edges pointing to the vertex `to`.
  - **Parameter**:
    - `T to` — The destination vertex.
  - **Return value**: A `DLinkedList<T>` containing the source vertices.

- `virtual int size() = 0;`
  - Returns the number of vertices in the graph.
  - **Return value**: The number of vertices.

- `virtual bool empty() = 0;`
  - Checks if the graph is empty.
  - **Return value**: `true` if the graph is empty, otherwise `false`.

- `virtual void clear() = 0;`
  - Removes all vertices and edges from the graph, making it empty.

- `virtual int inDegree(T vertex) = 0;`
  - Calculates the number of edges pointing to the vertex `vertex`.
  - **Parameter**:
    - `T vertex` — The vertex to calculate.
  - **Return value**: The in-degree of the vertex.

- `virtual int outDegree(T vertex) = 0;`
  - Calculates the number of edges pointing from the vertex `vertex`.
  - **Parameter**:
    - `T vertex` — The vertex to calculate.

  - **Return value**: The out-degree of the vertex.

- `virtual DLinkedList<T> vertices() = 0;`
  - Retrieves all vertices in the graph.
  - **Return value**: A `DLinkedList<T>` containing all vertices.

- `virtual string toString() = 0;`
  - Returns a string representation of the graph.
  - **Return value**: A `string` containing the graph's representation.

2.2 Abstract Graph

`AbstractGraph<T>` is an abstract base class used for implementing various types of graphs, such as directed, undirected, weighted, or unweighted graphs. The vertices in the graph are represented using the data type `T`, and edges are organized as adjacency lists to manage connections between vertices.

The `AbstractGraph<T>` class defines basic properties and methods for graph operations, including adding or removing vertices (`add`, `remove`), adding or removing edges (`connect`, `disconnect`), and querying information like the number of vertices (`size`), checking if the graph is empty (`empty`), or calculating the in-degree and out-degree of a vertex (`inDegree`, `outDegree`). Edges may have weights, which are managed through the `weight` method.

However, the implementation of methods like `connect`, `disconnect`, and `remove` depends on the graph's characteristics. For instance, in directed graphs, edges are created or removed in a specific direction, while in undirected graphs, edges are bidirectional between two vertices.

The `Edge` class represents an edge in the graph, including information about the two vertices it connects (`from` and `to`) and its weight (`weight`). This structure facilitates managing edges and accessing their attributes when working with the graph.

The `VertexNode<T>` class is used to represent a vertex and its associated edges in the adjacency list. Each vertex stores its value, a list of outward edges, and a list of inward edges.

Finally, an `Iterator` is provided as a tool to sequentially traverse the vertices or edges in the graph. This simplifies graph traversal for various applications, such as search, sorting, or connectivity checks.

Detailed definitions and implementations can be found in the file **AbstractGraph.h** within the **/include/graph** directory.

```
1 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();
    virtual void clear();
    virtual int inDegree(T vertex);
    virtual int outDegree(T vertex);
    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>: Graph implemented with adjacency lists

2.2.1 Edge

Below is a detailed description of the `Edge` class:

1. **Attributes:**
   - `VertexNode* from`: Pointer to the source vertex of the edge.
   - `VertexNode* to`: Pointer to the destination vertex of the edge.
   - `float weight`: The weight of the edge. Defaults to 0 if not specified.
2. **Constructor and Destructor:**
   - `Edge()` : Default constructor, initializes an edge without specific information about the vertices and weight.
   - `Edge(VertexNode* from, VertexNode* to, float weight = 0)` : Constructor with parameters specifying the source vertex `from`, the destination vertex `to`, and the weight of the edge `weight` (default is 0).
3. **Methods:**
   - **bool equals(Edge* edge)**
     - **Functionality**: Compares the current edge with another edge `edge`. Returns `true` if both edges have the same source and destination vertices, and `false` otherwise.
     - **Exceptions**: None.

- **static bool edgeEQ(Edge*& edge1, Edge*& edge2)**
  - **Functionality**: Compares two `Edge` objects, `edge1` and `edge2`, by calling the `equals` method of the `Edge` class. Returns `true` if the two edges are identical, and `false` otherwise.
  - **Exceptions**: None.
- **string toString()**
  - **Functionality**: Returns a string representation of the edge, including information about the source vertex `from`, the destination vertex `to`, and the weight of the edge `weight`.
  - **Exceptions**: None.

2.2.2 VertexNode

This section provides a detailed description of the `VertexNode` class:

1. **Attributes:**
   - `T vertex`: Stores the data associated with the vertex, where `T` is a generic data type.
   - `int inDegree_`: The in-degree of the vertex (number of incoming edges).
   - `int outDegree_`: The out-degree of the vertex (number of outgoing edges).
   - `DLinkedList<Edge*> adList`: A doubly linked list storing the adjacency list of edges connected to the vertex.
   - `bool (*vertexEQ)(T&, T&)`: A function pointer used to compare two vertices for equality.
   - `string (*vertex2str)(T&)`: A function pointer used to convert a vertex's data to its string representation.
2. **Constructors and Destructor:**
   - `VertexNode()`: Default constructor. Initializes the adjacency list with functions for memory cleanup and equality checking.
   - `VertexNode(T vertex, bool (*vertexEQ)(T&, T&), string (*vertex2str)(T&))`: Constructs a vertex node with the specified data `vertex`, equality function `vertexEQ`, and string conversion function `vertex2str`.
3. **Methods:**
   - `T& getVertex()`:
     - **Functionality**: Returns a reference to the vertex's data.

  - **Exceptions**: None.

- `void connect(VertexNode* to, float weight = 0)`:
  - **Functionality**: Connects this vertex to another vertex `to` by creating an edge with the specified weight (default is 0).
  - **Exceptions**: None.
- `DLinkedList<T> getOutwardEdges()`:
  - **Functionality**: Returns a list of outward edges from this vertex.
  - **Exceptions**: None.
- `Edge* getEdge(VertexNode* to)`:
  - **Functionality**: Retrieves the edge connecting this vertex to the specified vertex `to`. Returns `nullptr` if no such edge exists.
  - **Exceptions**: None.
- `bool equals(VertexNode* node)`:
  - **Functionality**: Compares this vertex with another `node` for equality using the `vertexEQ` function.
  - **Exceptions**: None.
- `void removeTo(VertexNode* to)`:
  - **Functionality**: Removes the edge connecting this vertex to the specified vertex `to`.
  - **Exceptions**: None.
- `int inDegree()`:
  - **Functionality**: Returns the in-degree of this vertex.
  - **Exceptions**: None.
- `int outDegree()`:
  - **Functionality**: Returns the out-degree of this vertex.
  - **Exceptions**: None.
- `string toString()`:
  - **Functionality**: Returns a string representation of this vertex, including its data, in-degree, and out-degree.
  - **Exceptions**: None.

2.2.3 AbstractGraph

Below is a detailed description of the `AbstractGraph` class:

1. **Attributes:**
   - `DLinkedList<VertexNode*> nodeList`: A doubly linked list containing the vertices of the graph, used to store all vertices as `VertexNode` objects.
   - `bool (*vertexEQ)(T&, T&)`: A pointer to a function for comparing two vertices, used to check vertex equality in the graph.
   - `string (*vertex2str)(T&)`: A pointer to a function for converting a vertex to a string, used to represent vertices as strings.
2. **Constructor and Destructor:**
   - `AbstractGraph(bool (*vertexEQ)(T&, T&)=0, string (*vertex2str)(T&)=0)`: Constructor for the `AbstractGraph` class with two function pointer parameters, `vertexEQ` and `vertex2str`, used for vertex comparison and representation. Defaults to `nullptr` if no values are provided.
   - `~AbstractGraph()`: Destructor, releases the graph's resources, deleting vertices and associated edges.
3. **Methods:**
   - **VertexNode* getVertexNode(T& vertex)**
     - **Functionality**: Searches for and returns a pointer to the `VertexNode` containing the vertex `vertex` in the graph. Returns `nullptr` if not found.
     - **Exceptions**: None.
   - **string vertex2Str(VertexNode& node)**
     - **Functionality**: Converts a `VertexNode` to the string representation of its vertex using the `vertex2str` function.
     - **Exceptions**: None.
   - **string edge2Str(Edge& edge)**
     - **Functionality**: Converts an `Edge` object into a string representation, including information about the source and destination vertices.
     - **Exceptions**: None.
   - **virtual void add(T vertex)**
     - **Functionality**: Adds a new vertex to the graph.
     - **Exceptions**: None.
   - **virtual bool contains(T vertex)**
     - **Functionality**: Checks whether the graph contains the vertex `vertex`.
     - **Exceptions**: None.

- **virtual float weight(T from, T to)**
  - **Functionality**: Returns the weight of the edge connecting vertex `from` to vertex `to`.
  - **Exceptions**: Throws `VertexNotFoundException` if a vertex is not found and `EdgeNotFoundException` if no edge exists between the two vertices.
- **virtual DLinkedList<T> getOutwardEdges(T from)**
  - **Function**: Retrieves the list of outward edges from the vertex `from`.
  - **Implementation**:
    (a) Find the `VertexNode` corresponding to the vertex `from` using the `getVertexNode(from)` function.
    (b) If the vertex `from` is not found, throw a `VertexNotFoundException` with information about the vertex.
    (c) If the vertex `from` exists, return the list of outward edges from this vertex using the `getOutwardEdges()` method of the `VertexNode` object.
  - **Exception**:
    * `VertexNotFoundException`: If the vertex `from` is not found.
- **virtual DLinkedList<T> getInwardEdges(T to)**
  - **Function**: Retrieves the list of inward edges to the vertex `to`.
  - **Implementation**:
    (a) Find the `VertexNode` corresponding to the vertex `to` using the `getVertexNode(to)` function.
    (b) If the vertex `to` is not found, throw a `VertexNotFoundException` with information about the vertex.
    (c) Initialize an empty list `list` to store the vertices with edges pointing to `to`.
    (d) Iterate over all vertices in the `nodeList`.
    (e) For each vertex `node`, iterate over its adjacent edges in `adList`.
    (f) If an edge has `to` as its destination, add the source vertex of the edge to the `list`.
    (g) Continue iterating through all the vertices and edges until completion.
  - **Exception**:
    * `VertexNotFoundException`: If the vertex `to` is not found.
- **virtual int size()**
  - **Functionality**: Returns the number of vertices in the graph.
  - **Exceptions**: None.

- **virtual bool empty()**
  - **Functionality**: Checks whether the graph is empty.
  - **Exceptions**: None.
- **virtual void clear()**
  - **Functionality**: Removes all vertices and edges in the graph.
  - **Exceptions**: None.
- **virtual int inDegree(T vertex)**
  - **Functionality**: Returns the in-degree (number of incoming edges) of the vertex `vertex`.
  - **Exceptions**: Throws `VertexNotFoundException` if the vertex `vertex` is not found.
- **int outDegree(T vertex)**
  - **Functionality**: Returns the out-degree (number of outgoing edges) of the vertex `vertex`.
  - **Exceptions**: Throws `VertexNotFoundException` if the vertex `vertex` is not found.
- **DLinkedList<T> vertices()**
  - **Functionality**: Returns a list of all vertices in the graph.
  - **Exceptions**: None.
- **bool connected(T from, T to)**
  - **Functionality**: Checks whether there is an edge between vertices `from` and `to`.
  - **Exceptions**: Throws `VertexNotFoundException` if a vertex is not found.
- **void println()**
  - **Functionality**: Prints the graph's representation to the console.
  - **Exceptions**: None.
- **string toString()**
  - **Functionality**: Returns a string representation of the entire graph, including the list of vertices and edges.
  - **Exceptions**: None.
- **Iterator begin()**
  - **Functionality**: Returns an iterator to traverse the vertices in the graph, starting from the first vertex.
  - **Exceptions**: None.
- **Iterator end()**

- **Functionality**: Returns an iterator to traverse the vertices in the graph, ending at the last vertex.
- **Exceptions**: None.

2.3 Directed Graph

The `DGraphModel<T>` class is a subclass of `AbstractGraph<T>` designed to implement a directed graph model. This class provides methods for manipulating directed graphs, including connecting vertices (`connect`), disconnecting vertices (`disconnect`), and removing vertices from the graph (`remove`). These methods ensure that connections between vertices are managed according to the directed nature of the graph.

The `DGraphModel<T>` class also provides a static method `create`, allowing the creation of a new `DGraphModel<T>` object from a predefined list of vertices and edges, facilitating graph initialization from input data.

```
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>: Directed Graph

Below is a detailed description of the `DGraphModel` class:

1. **Constructors:**
   - `DGraphModel(bool (*vertexEQ)(T&, T&), string (*vertex2str)(T&))`:
     - **Functionality**: Initializes a directed graph with two function parameters:

    - `vertexEQ`: Function to compare two vertices for equality.
    - `vertex2str`: Function to convert a vertex to a string for display.
  - **Exceptions**: None.

2. **Methods:**

- **void connect(T from, T to, float weight = 0)**
  - **Functionality**: Adds a directed edge from vertex `from` to vertex `to` with weight `weight` (default is 0).
  - **Implementation Steps**:
    (a) Retrieve the `VertexNode` objects corresponding to vertices `from` and `to`.
    (b) If any vertex does not exist, throw a `VertexNotFoundException`.
    (c) Connect `from` to `to` by adding a new edge with weight `weight`.
  - **Exceptions**:
    - Throws `VertexNotFoundException` if either vertex does not exist.
- **void disconnect(T from, T to)**
  - **Functionality**: Removes the edge from vertex `from` to vertex `to`.
  - **Implementation Steps**:
    (a) Retrieve the `VertexNode` objects corresponding to vertices `from` and `to`.
    (b) If any vertex does not exist, throw a `VertexNotFoundException`.
    (c) Retrieve the edge from `from` to `to`. If the edge does not exist, throw an `EdgeNotFoundException`.
    (d) Remove the edge from the graph.
  - **Exceptions**:
    - `VertexNotFoundException`: If either vertex does not exist.
    - `EdgeNotFoundException`: If no edge exists between the vertices.
- **void remove(T vertex)**
  - **Functionality**: Removes a vertex and all edges connected to it.
  - **Implementation Steps**:
    (a) Retrieve the `VertexNode` corresponding to the vertex to be removed.
    (b) If the vertex does not exist, throw a `VertexNotFoundException`.
    (c) Iterate through the graph's vertices, removing all edges connected to or from the vertex to be removed.
    (d) Remove the vertex from the graph's vertex list.
  - **Exceptions**:
    - `VertexNotFoundException`: If the vertex does not exist.

- **static DGraphModel<T>* create(T* vertices, int nvertices, Edge<T>* edges, int nedges, bool (*vertexEQ)(T&, T&), string (*vertex2str)(T&))**
  - **Functionality**: Creates a new directed graph from a list of vertices and edges.
  - **Implementation Steps**:
    (a) Initialize a new `DGraphModel` object.
    (b) Add all vertices in `vertices` to the graph.
    (c) Add all edges in `edges` to the graph.
    (d) Return a pointer to the created graph.
  - **Exceptions**: None.

2.4 Undirected Graph

The `UGraphModel<T>` class is a subclass of `AbstractGraph<T>` designed to implement an undirected graph model. This class provides methods for manipulating undirected graphs, including connecting vertices (`connect`), disconnecting vertices (`disconnect`), and removing vertices from the graph (`remove`). These methods ensure that connections between vertices are managed according to the undirected nature of the graph.

The `UGraphModel<T>` class also provides a static method `create`, allowing the creation of a new `UGraphModel<T>` object from a predefined list of vertices and edges, facilitating graph initialization from input data.

```
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>: Undirected Graph

Below is a detailed description of the `UGraphModel` class:

1. **Overview:**
   - `UGraphModel`: A class representing an undirected graph model, inheriting from `AbstractGraph<T`. This class provides essential operations on undirected graphs such as adding vertices, connecting vertices, disconnecting edges, and removing vertices.
   - **Template Parameter**: `T`, representing the data type of the vertices in the graph.
2. **Constructors:**
   - `UGraphModel(bool (*vertexEQ)(T&, T&), string (*vertex2str)(T&))`:
     - **Functionality**: Initializes an undirected graph with two function parameters:
       - `vertexEQ`: Function to compare two vertices for equality.
       - `vertex2str`: Function to convert a vertex to a string for display purposes.
     - **Exceptions**: None.
3. **Methods:**
   - **void connect(T from, T to, float weight = 0)**
     - **Functionality**: Adds an undirected edge between vertices `from` and `to` with weight `weight` (default is 0).
     - **Implementation Steps**:
       (a) Retrieve the `VertexNode` objects corresponding to `from` and `to`.
       (b) If any vertex does not exist, throw a `VertexNotFoundException`.
       (c) If `from` and `to` are the same, add a self-loop.
       (d) If `from` and `to` are different, add two edges (one from `from` to `to` and one from `to` to `from`).
     - **Exceptions**:
       - `VertexNotFoundException`: If either vertex does not exist.
   - **void disconnect(T from, T to)**
     - **Functionality**: Removes the undirected edge between vertices `from` and `to`.
     - **Implementation Steps**:
       (a) Retrieve the `VertexNode` objects corresponding to `from` and `to`.
       (b) If any vertex does not exist, throw a `VertexNotFoundException`.

(c) Retrieve the edge from `from` to `to`. If the edge does not exist, throw an `EdgeNotFoundException`.
(d) If `from` and `to` are the same, remove the self-loop.
(e) If `from` and `to` are different, remove both edges (one from `from` to `to` and one from `to` to `from`).

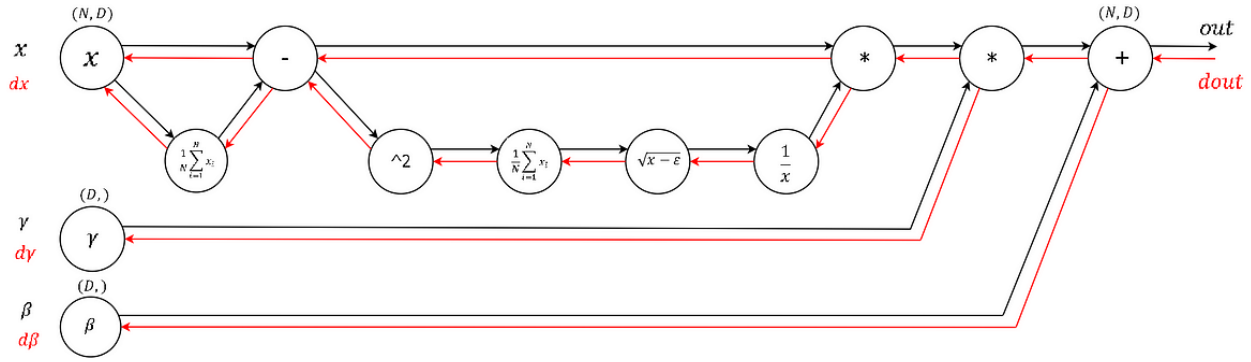
– **Exceptions**:
* `VertexNotFoundException`: If either vertex does not exist.
* `EdgeNotFoundException`: If no edge exists between the vertices.

- **void remove(T vertex)**
  – **Functionality**: Removes a vertex and all edges connected to it.
  – **Implementation Steps**:
    (a) Retrieve the `VertexNode` corresponding to the vertex to be removed.
    (b) If the vertex does not exist, throw a `VertexNotFoundException`.
    (c) Iterate through the graph's vertices, removing all edges connected to or from the vertex to be removed.
    (d) Remove the vertex from the graph's vertex list.
  – **Exceptions**:
    * `VertexNotFoundException`: If the vertex does not exist.
- **static UGraphModel<T>* create(T* vertices, int nvertices, Edge<T>* edges, int nedges, bool (*vertexEQ)(T&, T&), string (*vertex2str)(T&))**
  – **Functionality**: Creates a new undirected graph from a list of vertices and edges.
  – **Implementation Steps**:
    (a) Initialize a new `UGraphModel` object.
    (b) Add all vertices in `vertices` to the graph.
    (c) Add all edges in `edges` to the graph.
    (d) Return a pointer to the created graph.
  – **Exceptions**: None.

3 TASK 2: Topological Sorting

Backpropagation is a critical algorithm in neural network training, used to optimize parameters by propagating gradients from the output back to previous layers. This process consists of two main steps:

- **Forward Pass**: Calculates the output values based on the current weights.



Hình 1: Topological representation of the Backpropagation process

- **Backward Pass**: Uses derivatives to compute the gradient of the loss function with respect to the weights, followed by updating the weights using an optimization algorithm such as Gradient Descent.

In this assignment, students are required to implement a key step of Backpropagation: **Topological Sorting (Topo Sort)**. This is a method of ordering the vertices of a Directed Acyclic Graph (DAG) such that for every directed edge (u, v), vertex u appears before vertex v.

Topo Sort is used in Backpropagation to determine the order of gradient updates when the neural network has a complex structure, such as networks with multiple branches or non-sequential graphs.

3.1 TopoSorter

The `TopoSorter<T>` class facilitates topological sorting on Directed Acyclic Graphs (DAGs). This class enables users to sort the vertices of a graph in an order that ensures for every edge (u, v), vertex u precedes v.

The `TopoSorter<T>` class provides two primary methods for topological sorting: `dfsSort`, which uses Depth-First Search (DFS), and `bfsSort`, which uses Breadth-First Search (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);

protected:
    //Helping functions
    XHashMap<T, int> vertex2inDegree(int (*hash)(T&, int));
    XHashMap<T, int> vertex2outDegree(int (*hash)(T&, int));
    DLinkedList<T> listOfZeroInDegrees();
};
```

Listing 5: TopoSorter<T>

Below is a detailed description of the `TopoSorter` class:

1. **Attributes:**
   - `DGraphModel<T>* graph`: Pointer to the directed graph object containing the vertices and edges to be sorted, serving as the basis for topological sorting.
   - `int (*hash_code)(T&, int)`: Pointer to the hash function used to map graph vertices to integer values, facilitating hashed data structures. Students are encouraged to use the hash function to optimize their implementation.
   - `static int DFS`: Constant value (0) representing the DFS-based Topo Sort mode.
   - `static int BFS`: Constant value (1) representing the BFS-based Topo Sort mode.
2. **Constructor:**
   - `TopoSorter(DGraphModel<T>* graph, int (*hash_code)(T&, int)=0)`:
     - **Functionality**: Initializes a `TopoSorter` object with a graph and (optionally) a hash function. If no hash function is provided, the default value is `nullptr`.
     - **Parameters**:
       * `graph`: Pointer to the directed graph to be topologically sorted.
       * `hash_code`: Hash function for mapping vertices.
3. **Methods:**
   - `DLinkedList<T> sort(int mode=0, bool sorted=true)`
     - **Functionality**: Performs topological sorting using DFS or BFS, depending on the `mode`.

  - **Parameters**:
    - `mode`: Sorting mode, defaults to `DFS`.
    - `sorted`: If `true`, sorts the vertex list in ascending order for educational purposes.
  - **Returns**: A doubly linked list containing the vertices in topological order.
- `DLinkedList<T> bfsSort(bool sorted=true)`
  - **Functionality**: Performs topological sorting using BFS.
  - **Parameters**:
    - `sorted`: If `true`, sorts the vertex list in ascending order before processing.
  - **Returns**: A doubly linked list containing the vertices in topological order.
  - **Hints**:
    - Use helping functions such as `vertex2inDegree` and `listOfZeroInDegrees` to implement the algorithm.
    - Sorting the vertex list is recommended to ensure result consistency, using the pre-implemented `Merge Sort` in the `DLinkedListSE` class.
- `DLinkedList<T> dfsSort(bool sorted=true)`
  - **Functionality**: Performs topological sorting using DFS.
  - **Parameters**:
    - `sorted`: If `true`, sorts the vertex list in ascending order before processing.
  - **Returns**: A doubly linked list containing the vertices in topological order.
  - **Hints**:
    - Use helping functions such as `vertex2outDegree` and `listOfZeroInDegrees` to implement the algorithm.
    - Sorting the vertex list is recommended to ensure result consistency, using the pre-implemented `Merge Sort` in the `DLinkedListSE` class.

4. **Supporting Methods:**
   - `XHashMap<T, int> vertex2inDegree(int (*hash)(T&, int))`
     - **Functionality**: Creates a hash map storing the in-degrees of all vertices in the graph.
     - **Parameters**:
       - `hash`: Hash function for mapping vertices.
     - **Returns**: An `XHashMap` object with vertices as keys and their in-degrees as values.
   - `DLinkedList<T> listOfZeroInDegrees()`

- **Functionality**: Creates a list of vertices with in-degrees equal to zero.
- **Returns**: A doubly linked list containing vertices with zero in-degrees.

3.2 Supporting Classes

To support the implementation of `TopoSorter`, the `Stack`, `Queue`, and `DLinkedListSE` classes are provided with prototypes and guidance in the source code. Students are encouraged to utilize these classes in their implementation. If unused, students must retain these files without modifications.

4 Submission

Students are required to submit files before the deadline specified in the "Assignment 3 - Submission" path. There are some simple testcases used to check the students' work to ensure that the results can be compiled and run. Students can submit their work as many times as they want, but only the last submission will be graded. Because the system cannot handle the load when too many students submit their work at the same time, students should submit their work as early as possible. Students will bear the risk if they submit their work close to the deadline. Once the deadline for submission has passed, the system will be closed, and students will not be able to submit anymore. Submissions through other methods will not be accepted.

5 Assignment Harmony

The final exam for this course will include some Harmony questions based on the content of the assignment. Assume the assignment score achieved by a student is **a** (out of 10), and the total score for the Harmony questions is **b** (out of 5). Let **x** be the final assignment score after Harmony. The final score will be combined with 50% of the Harmony score as follows:

- If $a = 0$ or $b = 0$, then $x = 0$
- If both a and b are non-zero, then

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

where:

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

Students must complete the assignment independently. If a student cheats on the assignment, they will be unable to answer the Harmony questions and will receive a score of 0 for the assignment.

Students **must** answer the Harmony questions on the final exam. Failure to answer will result in a score of 0 for the assignment and course failure. **No explanations or exceptions will be accepted.**

6 Handling Cheating

The assignment must be completed independently. Cheating includes the following cases:

- Unusual similarities between code submissions. In this case, ALL submissions will be considered cheating. Students must protect their assignment code.
- Students cannot explain the code they submitted, except for the pre-existing code provided in the initial program. Students may refer to any resource, but they must understand every line of code they write. If a student does not fully understand the code from a source, they are strongly advised NOT to use it; instead, rely on their learning.
- Submitting another student's work under their account.

If cheating is confirmed, students will receive a 0 for the entire course (not just the assignment).

NO EXPLANATIONS
OR EXCEPTIONS WILL BE ACCEPTED!

After each assignment is submitted, some students will be randomly selected for an interview to prove that the assignment submitted is their own work.