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

# KNOWLEDGE GRAPH IMPLEMENTATION USING GRAPH DATA STRUCTURES

---

HO CHI MINH CITY, 11/2025

# ASSIGNMENT'S SPECIFICATION

**Version 1.0**

## 1 Assignment's outcome

Upon completion of this assignment, students will be able to:

- Master Object-Oriented Programming (OOP).
- Develop graph data structures.
- Utilize graph data structures to construct a Knowledge Graph.

## 2 Introduction

In previous assignments, students successfully built a *VectorStore* with storage and search capabilities based on vector similarity, utilizing data structures ranging from basic (Lists) to advanced (AVL Trees, Red-Black Trees). However, in real-world knowledge management problems, data does not merely exist independently or link solely through mathematical distance; it is also interconnected by direct semantic relationships (e.g., articles citing each other, products within the same category, or related concepts).

Assignment 3 requires students to upgrade the current system by integrating the **Graph** data structure to construct a **Knowledge Graph**. In this model, each string will serve as a vertex of the graph, and the connections between them will be represented by edges.

This transformation allows the system not only to answer questions like "what is most similar to this" but also to address structural inquiries such as "which entities are related to each other" or "what is the connection path between any two entities." Through this assignment, students will complete the comprehensive picture of applying Data Structures and Algorithms to solve multi-dimensional knowledge storage and mining problems.

## 3 Description

### 3.1 Graph Data Structures

The graph data structure (Graph) in this assignment is designed with a simplified approach, consisting of the following main classes:

- class `DGraphModel<T>`: The main class for implementing the directed graph (Directed Graph). This class contains all methods to add, remove, and manage vertices and edges in the directed graph. It does not use complex inheritance models or interfaces; all logic is integrated directly into this class.
- class `Edge`: Represents an edge in the graph, including information about the two vertices it connects (source and destination) along with the weight.
- class `VertexNode<T>`: Represents a vertex in the graph along with all related edges (adjacency list).

### 3.1.1 Edge

The `Edge` class represents an edge in the graph, including information about the two vertices it connects (`from` and `to`) along with the weight (`weight`).

1. **Attributes:**
   - `VertexNode* from`: Pointer to the source vertex of the edge.
   - `VertexNode* to`: Pointer to the destination vertex of the edge.
   - `float weight`: Weight of the edge. Defaults to 0 if not specified.
2. **Constructors and Destructors:**
   - `Edge()`: Default constructor, initializes an edge without specific information about vertices and weight.
   - `Edge(VertexNode* from, VertexNode* to, float weight = 0)`: Constructor with parameters specifying the source vertex `from`, destination vertex `to`, and edge weight `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.
     - **Exception**: 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.

- **Exception**: None.

- **string toString()**
  - **Function**: Returns a string representation of the edge, including the source vertex `from`, the destination vertex `to`, and the edge weight `weight`.
  - **Exceptions**: None.
  - **Format**: `(<value of from vertex>, <value of to vertex>, <weight>)`
    **Example:** For the graph in Figure 1

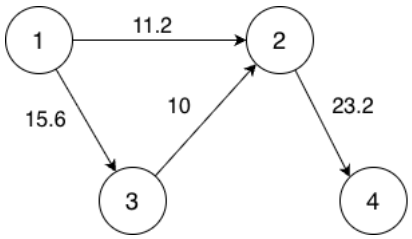


Figure 1: Illustration of a directed graph

The return format of the edge (1,2) is: `(1, 2, 11.2)`

### 3.1.2 VertexNode

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

1. **Attributes:**
   - `T vertex`: Data stored in the vertex, where `T` is a generic data type.
   - `int inDegree_`: In-degree of the vertex (number of incoming edges).
   - `int outDegree_`: Out-degree of the vertex (number of outgoing edges).
   - `vector<Edge*> adList`: List storing the adjacency list of edges connected to the vertex.
   - `bool (*vertexEQ)(T&, T&)`: Function pointer used to compare if data stored in two vertices are equal.
   - `string (*vertex2str)(T&)`: Function pointer used to convert vertex data into a string.
2. **Constructor:**
   - `VertexNode(T vertex, bool (*vertexEQ)(T&, T&), string (*vertex2str)(T&))`: Constructor, creates a vertex with data `vertex`, equality check function `vertexEQ`, and string conversion function `vertex2str`.

3. **Methods:**
   - `T& getVertex()`
     - **Functionality**: Returns a reference to the vertex data.
     - **Exception**: None.
   - `void connect(VertexNode* to, float weight = 0)`
     - **Functionality**: Connects the current vertex with vertex `to` by creating an edge with a weight (default is 0).
     - **Exception**: None.
   - `Edge* getEdge(VertexNode* to)`
     - **Functionality**: Returns a pointer to the edge connecting from the current vertex to vertex `to`. Returns `nullptr` if the edge is not found.
     - **Exception**: None.
   - `bool equals(VertexNode* node)`
     - **Functionality**: Compares the current vertex with another vertex `node` based on the `vertexEQ` function.
     - **Exception**: None.
   - `void removeTo(VertexNode* to)`
     - **Functionality**: Removes the connecting edge from the current vertex to vertex `to`.
     - **Exception**: None.
   - `int inDegree()`
     - **Functionality**: Returns the in-degree of the vertex.
     - **Exception**: None.
   - `int outDegree()`
     - **Functionality**: Returns the out-degree of the vertex.
     - **Exception**: None.
   - `string toString()`
     - **Function**: Returns a string representation of the vertex, including its data value, in-degree, out-degree, and all adjacent edges (in the order stored in `adList`).
     - **Exceptions**: None.
     - **Format**: `(<value>, <in-degree>, <out-degree>, [<adjacent edge list>])`. Each edge is printed using its `toString` method. If the vertex has no adjacent edges, an empty list `[]` is printed.

       **Example:** For the graph in Figure 1

The printed format of vertex (3) is: `(3, 1, 1, [(1, 3, 15.6), (3, 2, 10)])`

### 3.1.3 Directed Graph

The `DGraphModel<T>` class directly implements a directed graph model (Directed Graph). This class provides methods to add, remove, and manage vertices and edges in a directed graph. It does not use interfaces or abstract classes; all logic is integrated directly into this class.

1. **Attributes:**
   - `vector<VertexNode<T>*> nodeList`: List containing all vertices of the graph.
   - `bool (*vertexEQ)(T&, T&)`: Function pointer to a function comparing two vertices, used to check for equality of vertices in the graph.
   - `string (*vertex2str)(T&)`: Function pointer to a function converting vertex data into a string, used to represent the vertex as a string.
2. **Constructors and Destructors:**
   - `DGraphModel(bool (*vertexEQ)(T&, T&)=0, string (*vertex2str)(T&)=0)`: Constructor for the `DGraphModel` class with two function pointer parameters `vertexEQ` and `vertex2str` used for comparing and representing vertices. If no values are provided, these parameters will be `nullptr`.
   - `DGraphModel()`: Destructor, releases graph resources, deletes vertices and related edges.
3. **Methods:**
   - `VertexNode<T>* getVertexNode(T& vertex)`
     - **Functionality**: Searches for and returns a pointer to the `VertexNode` containing the vertex `vertex` in the graph. If not found, returns `nullptr`.
     - **Exception**: None.
   - `string vertex2Str(VertexNode<T>& node)`
     - **Functionality**: Converts a `VertexNode` into the string representation of the vertex via the `vertex2str` function.
     - **Exception**: None.
   - `string edge2Str(Edge& edge)`
     - **Functionality**: Converts an `Edge` object into the string representation of the edge, including information about the source and destination vertices.
     - **Exception**: None.

- `void add(T vertex)`
  - **Functionality**: Adds a new vertex to the graph.
  - **Exception**: None.
- `bool contains(T vertex)`
  - **Functionality**: Checks if the graph contains the vertex `vertex`.
  - **Exception**: None.
- `float weight(T from, T to)`
  - **Functionality**: Returns the weight of the edge connecting from vertex `from` to vertex `to`.
  - **Exception**: Throws `VertexNotFoundException` if the vertex is not found, and `EdgeNotFoundException` if the edge between the two vertices is not found.
- `vector<Edge<T>*> getOutwardEdges(T from)`
  - **Functionality**: Retrieves a list of outward edges from vertex `from`.
  - **Implementation**:
    (a) Find the `VertexNode` corresponding to vertex `from` via the function `getVertexNode(fro`
    (b) If vertex `from` is not found, throw `VertexNotFoundException` with information about the vertex.
    (c) If vertex `from` exists, return the list of edges going out from this vertex via the `getOutwardEdges()` method of the `VertexNode` object.
  - **Exception**:
    * `VertexNotFoundException`: If vertex `from` is not found.
- `void connect(T from, T to, float weight = 0)`
  - **Functionality**: Connects two vertices `from` and `to` with an edge, with a default weight of `0`.
  - **Implementation**:
    (a) Check if vertex `from` already exists in the graph. If not, throw `VertexNotFoundExceptio` with information about vertex `from`.
    (b) Check if vertex `to` already exists in the graph. If not, throw `VertexNotFoundException` with information about vertex `to`.
    (c) Retrieve pointers to the `VertexNode` of vertices `from` and `to`.
    (d) Call the `connect` method of vertex `from` to create a connecting edge to vertex `to` with weight `weight`.
  - **Parameters**:
    * `T from`: Starting vertex.

    - `T to`: Ending vertex.
    - `float weight`: Weight of the edge (default is 0).
  - **Exception**:
    - `VertexNotFoundException`: If vertex `from` or vertex `to` is not found.

- `void disconnect(T from, T to)`
  - **Functionality**: Removes the edge between two vertices `from` and `to`.
  - **Implementation**:
    (a) Check if vertex `from` already exists in the graph. If not, throw `VertexNotFoundExceptio` with information about vertex `from`.
    (b) Check if vertex `to` already exists in the graph. If not, throw `VertexNotFoundException` with information about vertex `to`.
    (c) Retrieve pointers to the `VertexNode` of vertices `from` and `to`.
    (d) Call the `removeTo` method of vertex `from` to remove the connecting edge to vertex `to`.
  - **Parameters**:
    - `T from`: Starting vertex.
    - `T to`: Ending vertex.
  - **Exception**:
    - `VertexNotFoundException`: If vertex `from` or vertex `to` is not found.

- `bool connected(T from, T to)`
  - **Functionality**: Checks if two vertices `from` and `to` are connected by an edge.
  - **Implementation**:
    (a) Find the `VertexNode` corresponding to vertex `from` via the function `getVertexNode(fro`
    (b) If vertex `from` is not found, throw `VertexNotFoundException` with information about the vertex.
    (c) Find the `VertexNode` corresponding to vertex `to` via the function `getVertexNode(to)`.
    (d) If vertex `to` is not found, throw `VertexNotFoundException` with information about the vertex.
    (e) Call the `getEdge` method of vertex `from` to check if there is an edge to vertex `to`.
    (f) Return `true` if the edge is found, otherwise return `false`.
  - **Parameters**:
    - `T from`: Starting vertex.
    - `T to`: Ending vertex.

  - **Return Value**: `true` if the two vertices are connected, otherwise `false`.
  - **Exception**:
    * `VertexNotFoundException`: If vertex `from` or vertex `to` is not found.

- `int size()`
  - **Functionality**: Returns the number of vertices in the graph.
  - **Exception**: None.
- `bool empty()`
  - **Functionality**: Checks if the graph is empty.
  - **Exception**: None.
- `void clear()`
  - **Functionality**: Removes all vertices and edges in the graph.
  - **Exception**: None.
- `int inDegree(T vertex)`
  - **Functionality**: Returns the in-degree (number of incoming edges) of vertex `vertex`.
  - **Exception**: Throws `VertexNotFoundException` if vertex `vertex` is not found.
- `int outDegree(T vertex)`
  - **Functionality**: Returns the out-degree (number of outgoing edges) of vertex `vertex`.
  - **Exception**: Throws `VertexNotFoundException` if vertex `vertex` is not found.
- `vector<T> vertices()`
  - **Functionality**: Returns a list of all vertices in the graph.
  - **Exception**: None.
- `string toString()`
  - **Function**: Returns a string representation of the entire graph, including the list of vertices in the exact order stored in `nodeList`. Each vertex is printed using the `toString()` method of `VertexNode`.
  - **Exceptions**: None.
  - **Format**: `[<vertexNode1>, <vertexNode2>, ..., <vertexNodeN>]`. If the graph is empty, prints `[]`.

    **Example:** For the graph in Figure 1

    The returned format of the graph is: `[(1, 0, 2, [(1, 2, 11.2), (1, 3, 15.6)]), (2, 2, 1, [(1, 2, 11.2), (3, 2, 10), (2, 4, 23.2)]), (3, 1, 1, [(1, 3, 15.6), (3, 2, 10)]), (4, 1, 0, [(2, 4, 23.2)])]`

- `string bfs(T start)`
  - **Functionality**: Traverses the graph using Breadth-First Search (BFS) starting from vertex `start`, returns a string representing the list of visited vertices in BFS order, where each vertex is converted to a string using the `vertex2Str` function.
  - **Parameters**:
    - `T start`: Vertex to start BFS traversal.
  - **Return Value**: `string` containing the string representation of visited vertices in BFS order starting from `start`.
  - **Format**: `[<vertexNode1>, <vertexNode2>, ...]`. If the graph is empty, prints `[]`. For each vertex, the `toString` method of the `VertexNode` class is used for printing.
  - **Exception**:
    - `VertexNotFoundException`: If vertex `start` is not found in the graph.
- `string dfs(T start)`
  - **Functionality**: Traverses the graph using Depth-First Search (DFS) starting from vertex `start`, returns a string representing the list of visited vertices in DFS order, where each vertex is converted to a string using the `vertex2Str` function.
  - **Parameters**:
    - `T start`: Vertex to start DFS traversal.
  - **Return Value**: `string` containing the string representation of visited vertices in DFS order starting from `start`.
  - **Format**: `[<vertexNode1>, <vertexNode2>, ...]`. If the graph is empty, prints `[]`. For each vertex, the `toString` method of the `VertexNode` class is used for printing.
  - **Exception**:
    - `VertexNotFoundException`: If vertex `start` is not found in the graph.

## 3.2 Knowledge Graph

### 3.2.1 Introduction

A Knowledge Graph is a model for representing information in the form of a graph, in which entities such as people, places, events, or concepts are represented by nodes, and the relation-

ships between them are represented by edges. Unlike traditional databases that store discrete tabular data, a Knowledge Graph organizes data into a connected network with clear structure and semantics.

The operational mechanism of a Knowledge Graph begins with aggregating data from various sources, followed by processing to accurately identify entities and determine the relationships between them. Natural language processing and machine learning algorithms are applied to extract, clean, and consolidate information. This graph enables intelligent querying based on data relationships and context, rather than simple keyword-based search.

Through the Knowledge Graph, information retrieval, analysis, and mining become more efficient, supporting artificial intelligence applications, recommendation systems, decision support, and various other fields such as e-commerce, healthcare, education, and enterprise data governance.

### 3.2.2 KnowledgeGraph Class

The `KnowledgeGraph` class represents a knowledge graph, consisting of the following components.

**Attributes**

- `DGraphModel<string> graph`: Base graph object storing entities and relationships.
- `vector<string> entities`: List of all entities in the graph (synchronized with `graph`).

**Methods**

- `KnowledgeGraph()`
  - **Functionality**: Default constructor of the `KnowledgeGraph` class.
- `void addEntity(string entity)`
  - **Functionality**: Adds a new entity to the Knowledge Graph.
  - **Implementation**:
    1. Check if the entity already exists; if so, throw `EntityExistsException`.
    2. Add the new entity to the knowledge graph.
    3. Add `entity` to the end of the entity list.
  - **Parameters**:
    - `string entity`: Entity name.

  - **Return Value**: None.
  - **Exception**:
    * `EntityExistsException`: If the entity already exists.

- `void addRelation(string from, string to, float weight = 1.0f)`
  - **Functionality**: Adds a directed relationship with a specific relationship type.
  - **Implementation**:
    1. Check if both `from` and `to` exist.
    2. If not, throw `EntityNotFoundException`.
    3. Call the function to connect a new edge.
  - **Parameters**:
    * `string from`: Source entity.
    * `string to`: Destination entity.
    * `float weight`: Weight (default 1.0).
  - **Return Value**: None.
  - **Exception**:
    * `EntityNotFoundException`: If `from` or `to` does not exist.

- `vector<string> getAllEntities()`
  - **Functionality**: Returns a list of all entities in the graph.
  - **Return Value**: `vector<string>` containing all entities.

- `vector<string> getNeighbors(string entity)`
  - **Functionality**: Returns a list of directly adjacent entities (out-neighbors).
  - **Implementation**:
    1. Check if the graph contains the entity `entity`; if not, throw `EntityNotFoundException`.
    2. Return the list of adjacent entities.
  - **Parameters**:
    * `string entity`: Entity to find neighbors for.
  - **Return Value**: `vector<string>` of adjacent entities.
  - **Exception**:
    * `EntityNotFoundException`: If the entity does not exist.

- `string bfs(string start)`
  - **Functionality**: Performs BFS traversal from the start entity, returns the traversal order string. Each node

  - **Implementation**:
  - **Parameters**:
    - `string start`: Starting entity.
  - **Return Value**: `string` BFS traversal order.
  - **Exception**:
    - `EntityNotFoundException`: If the entity does not exist.

- `string dfs(string start)`
  - **Functionality**: Performs DFS traversal from the start entity, returns the traversal order string.
  - **Parameters**:
    - `string start`: Starting entity.
  - **Return Value**: `string` DFS traversal order.
  - **Exception**:
    - `EntityNotFoundException`: If the entity does not exist.

- `bool isReachable(string from, string to)`
  - **Functionality**: Checks if there is a path from the source entity to the destination.
  - **Parameters**:
    - `string from`: Source entity.
    - `string to`: Destination entity.
  - **Return Value**: `bool` - true if a path exists.
  - **Exception**:
    - `EntityNotFoundException`: If one of the two does not exist.

- `string toString()`
  - **Functionality**: Represents the general overview of the Knowledge Graph as a string.
  - **Return Value**: `string` representing the graph.

- `vector<string> getRelatedEntities(string entity, int depth = 2)`
  - **Functionality**: Finds all entities related to an entity within a range of `depth` steps.
  - **Implementation**:
    1. Check if the vertex `entity` exists; if not, throw `EntityNotFoundException`.
    2. Use BFS with depth limit `depth`, collecting all visited vertices.
    3. Remove duplicates and return the list of related entities.
  - **Parameters**:

    - `string entity`: Central entity.
    - `int depth`: Search depth (default 2).
  - **Return Value**: `vector<string>` of related entities.
  - **Exception**:
    - `EntityNotFoundException`: If the entity does not exist.

- `string findCommonAncestors(string entity1, string entity2)`
  - **Functionality**: Finds the Lowest Common Ancestor (LCA) of 2 entities (KG feature: reasoning about shared relationships).
  - **Implementation**:
    1. Find all ancestors of `entity1` and `entity2` using reverse DFS.
    2. Find the intersection of the 2 ancestor sets, select the nearest ancestor (shortest path to both).
  - **Parameters**:
    - `string entity1`: First entity.
    - `string entity2`: Second entity.
  - **Return Value**: `string` name of the lowest common ancestor or "No common ancestor".
  - **Exception**:
    - `EntityNotFoundException`: If one of the two does not exist.

# 4 Requirements and Grading

## 4.1 Requirements

To complete this assignment, students need to:

1. Read this entire description file carefully.
2. Download the **initial.zip** file and extract it. After extracting, students will obtain the following files: `utils.h`, `main.cpp`, `main.h`, `KnowledgeGraph.h`, `KnowledgeGraph.cpp`, and a folder containing sample outputs. Students only need to submit two files: `KnowledgeGraph.h` and `KnowledgeGraph.cpp`. Therefore, students are not allowed to modify the `main.h` file when testing the program.
3. Use the following command to compile:
   `g++ -o main main.cpp KnowledgeGraph.cpp -I . -std=c++17`

This command should be used in Command Prompt/Terminal to compile the program. If students use an IDE to run the program, note that they must: add all files to the IDE's project/workspace; modify the build command in the IDE accordingly. IDEs usually provide a Build button and a Run button. When clicking Build, the IDE runs the corresponding compile command, which typically compiles only `main.cpp`. Students must configure the compile command to include `KnowledgeGraph.cpp`, and add the options `-std=c++17` and `-I .`

4. The program will be graded on a Unix-based platform. Students' environments and compilers may differ from the actual grading environment. The submission area on LMS is configured similarly to the grading environment. Students must check their program on the submission page and fix all errors reported by LMS to ensure correct final results.
5. Edit the `KnowledgeGraph.h` and `KnowledgeGraph.cpp` files to complete the assignment, while ensuring the following two requirements:
   - All methods described in this guide must be implemented so that the program can compile successfully. If a method has not yet been implemented, students must provide an empty implementation for that method. Each test case will call certain methods to check their return values.
   - The file `KnowledgeGraph.h` must contain exactly one line `#include "main.h"`, and the file `KnowledgeGraph.cpp` must contain exactly one line `#include "KnowledgeGraph.h"`. Apart from these, no other `#include` statements are allowed in these files.
   - Students are not allowed to use the directive `#define TESTING` in the two files that are required to be modified.
6. Students are encouraged to write additional supporting classes, methods, and attributes within the classes they are required to implement. However, these additions must not change the requirements of the methods described in the assignment.
7. Students must design and use data structures learned in the course.
8. Students must ensure that all dynamically allocated memory is properly freed when the program terminates.

## 4.2 Submission Deadline

The deadline is **as announced on LMS**. Students must submit their assignments to the system before the specified deadline. Students are fully responsible for any issues arising from submissions made too close to the deadline.

### 4.3 Grading

All student source code will be evaluated using hidden test cases, and scores will be calculated based on each specific requirement.

## 5 Harmony Questions

The final exam for the course will include several "Harmony" questions related to the content of the Assignment.

Students must complete the Assignment by their own ability. If a student cheats in the Assignment, they will not be able to answer the Harmony questions and will receive a score of 0 for the Assignment.

Students **must** pay attention to completing the Harmony questions in the final exam. Failing to do so will result in a score of 0 for the Assignment, and the student will fail the course. **No explanations and no exceptions.**

## 6 Regulations and Handling of Cheating

The Assignment must be done by the student THEMSELVES. A student will be considered cheating if:

- There is an unusual similarity between the source code of submitted projects. In this case, ALL submissions will be considered as cheating. Therefore, students must protect their project source code.
- The student does not understand the source code they have written, except for the parts of code provided in the initialization program. Students can refer to any source of material, but they must ensure they understand the meaning of every line of code they write. If they do not understand the source code from where they referred, the student will be specifically warned NOT to use this code; instead, they should use what has been taught to write the program.
- Submitting someone else's work under their own account.
- Students use AI tools during the Assignment process, resulting in identical source code.

If the student is concluded to be cheating, they will receive a score of 0 for the entire course (not just the assignment).

**NO EXPLANATIONS WILL BE ACCEPTED AND THERE WILL BE NO EXCEPTIONS!**

After the final submission, some students will be randomly selected for an interview to prove that the submitted project was done by them.

Other regulations:

- All decisions made by the lecturer in charge of the assignment are final decisions.
- Students are not provided with test cases after the grading of their project.
- The content of the Assignment will be harmonized with questions in the exam that has similar content.

# 7 Changelog

- Updated the return format of the `toString` methods.
- Updated the return type of the `getOutwardEdges` method from `vector<T>` to `vector<Edge<T>*>`.
- Updated the return format of the `DFS` and `BFS` methods.

**——————————THE END——————————**