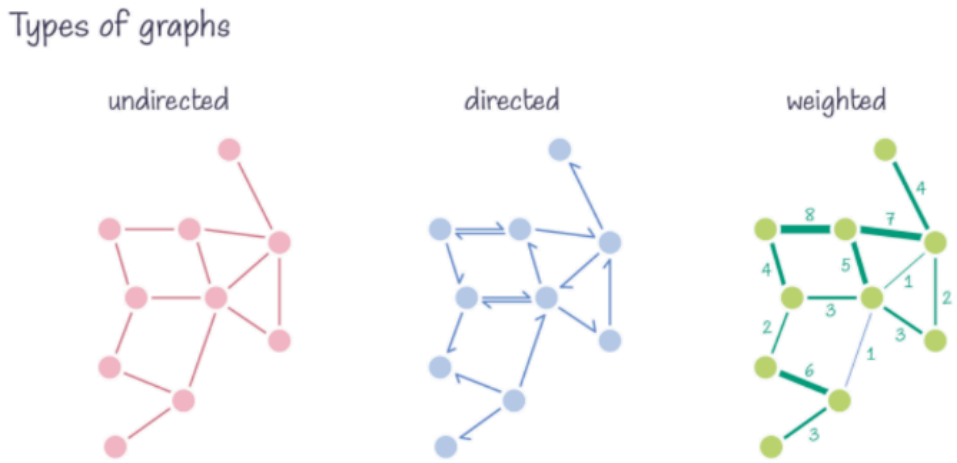
Mục lục

1 Lý Thuyết về graph

1.1 Cấu trúc đồ thị

Cấu trúc đồ thị bao gồm tập hợp các đỉnh (vertices) và các cạnh (edges) nối giữa chúng. Đồ thị có thể được sử dụng để mô hình hóa nhiều vấn đề trong thực tế như mạng lưới giao thông, mạng máy tính, quan hệ xã hội, và nhiều lĩnh vực khác. Dưới đây là một số khái niệm cơ bản:

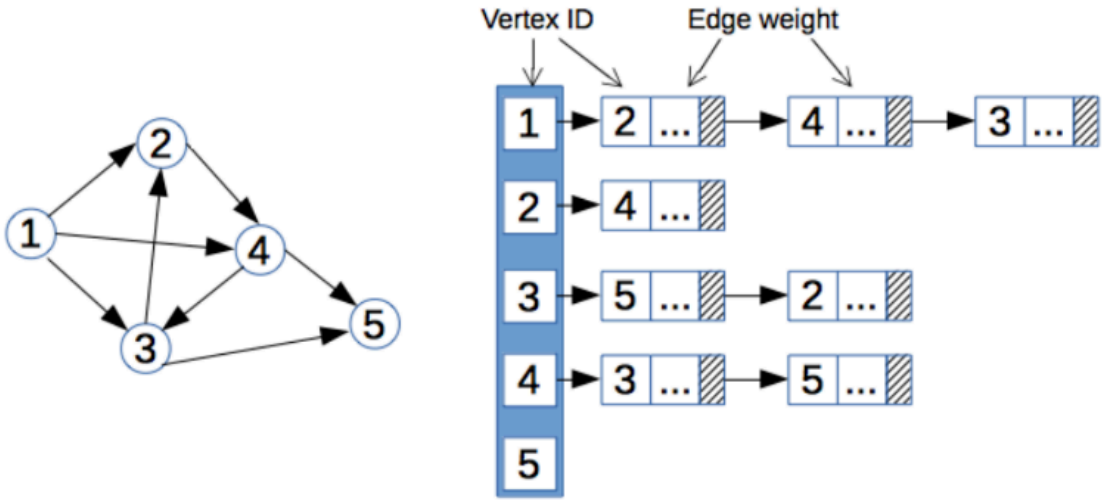


Hình 1: *Type of graphs*

- **Đồ thị (Graph):** Là một cặp $G = (V, E)$, trong đó:
  - V là tập hợp các đỉnh (vertices).
  - E là tập hợp các cạnh (edges), mỗi cạnh kết nối hai đỉnh (có thể có hướng hoặc không có hướng).
- **Đồ thị vô hướng (Undirected Graph):** Các cạnh không có hướng. Nếu $e = (u, v)$ là một cạnh thì u và v có thể đi qua lại.
- **Đồ thị có hướng (Directed Graph):** Các cạnh có hướng, nghĩa là $e = (u, v)$ chỉ cho phép đi từ u đến v.
- **Đồ thị có trọng số (Weighted Graph):** Mỗi cạnh được gán một giá trị (trọng số), thường biểu diễn chi phí, khoảng cách hoặc thời gian.
- **Bậc của đỉnh (Degree):** Số cạnh liên kết với một đỉnh. Với đồ thị có hướng:
  - **In-degree:** Số cạnh đi vào một đỉnh.
  - **Out-degree:** Số cạnh đi ra từ một đỉnh.

1.2 Danh sách kề (Adjacency List)

Danh sách kề (Adjacency List) là một cách phổ biến để biểu diễn đồ thị trong lập trình, đặc biệt phù hợp với các đồ thị thưa (sparse graph), nơi số lượng cạnh ít so với số đỉnh. Cách biểu diễn này tiết kiệm bộ nhớ và dễ dàng thao tác trên các đỉnh kề của đồ thị.



Hình 2: *Adjacency List*

- **Mảng của danh sách:** Sử dụng một mảng, trong đó mỗi phần tử là một danh sách chứa các đỉnh kề.
- **Danh sách liên kết:** Sử dụng danh sách liên kết để lưu các đỉnh kề.
- **HashMap (hoặc Dictionary):** Sử dụng một Map (hoặc HashMap) với khóa là đỉnh và giá trị là danh sách hoặc tập hợp các đỉnh kề.

Giả sử có đồ thị vô hướng với 4 đỉnh $V=\{0,1,2,3\}$ và các cạnh:

- (0,1)
- (0,2)
- (1,2)
- (2,3)

Danh sách kề của đồ thị sẽ như sau:

- Đỉnh 0: [1,2]
- Đỉnh 1: [0,2]
- Đỉnh 2: [0,1,3]
- Đỉnh 3: [2]

2 TASK 1 BTL

2.1 Ý tưởng dùng Danh sách kề (giả định vector là danh sách liên kết cho dễ hiểu)

Dưới đây là ví dụ về biểu diễn đồ thị bằng cấu trúc danh sách kề sử dụng `vector<vector<Edge»`:

```cpp
#include <iostream>
#include <vector>
using namespace std;

// Định nghĩa cấu trúc cạnh
struct Edge {
    char from, to;   // Dữ liệu của đỉnh
    float weight;    // Trọng số của cạnh
};

int main() {
    // Danh sách kề cho đồ thị
    vector<vector<Edge>> graph(4);

    // Thêm các cạnh vào danh sách kề
    graph[0].push_back({'A', 'B', 2.5}); // A -> B
    graph[0].push_back({'A', 'C', 1.0}); // A -> C
    graph[1].push_back({'B', 'D', 3.2}); // B -> D
    graph[2].push_back({'C', 'D', 4.5}); // C -> D

    // Hiển thị danh sách kề
    char vertices[] = {'A', 'B', 'C', 'D'};
    for (int i = 0; i < graph.size(); i++) {
        cout << "Vertex " << vertices[i] << ": ";
        for (const auto& edge : graph[i]) {
            cout << "(" << edge.to << ", " << edge.weight << ") ";
        }
        cout << endl;
    }

    return 0;
}
```

Kết quả đầu ra của chương trình sẽ như sau:

```
Vertex A: (B, 2.5) (C, 1.0)
Vertex B: (D, 3.2)
Vertex C: (D, 4.5)
Vertex D:
```

2.2 API của DLinkedList dùng trong task 1

1. **add(T e)**: Thêm phần tử `e` vào cuối danh sách.
2. **add(int index, T e)**: Thêm phần tử `e` vào vị trí chỉ định bởi `index`.
3. **removeAt(int index)**: Xóa và trả về phần tử tại vị trí `index`.
4. **removeItem(T item, void (*removeItemData)(T)=0)**: Xóa phần tử `item` khỏi danh sách và có thể xóa dữ liệu kèm theo.
5. **empty()**: Trả về `true` nếu danh sách rỗng, ngược lại trả về `false`.
6. **size()**: Trả về số lượng phần tử trong danh sách.
7. **clear()**: Xóa tất cả các phần tử, đưa danh sách về trạng thái ban đầu.
8. **get(int index)**: Trả về tham chiếu đến phần tử tại vị trí `index`.
9. **indexOf(T item)**: Trả về vị trí của phần tử `item`, hoặc `-1` nếu không tìm thấy.
10. **contains(T item)**: Trả về `true` nếu danh sách chứa phần tử `item`, ngược lại trả về `false`.
11. **toString(string (*item2str)(T&)=0)**: Trả về chuỗi mô tả danh sách bằng cách chuyển từng phần tử thành chuỗi.

2.4 AbstractGraph

AbstractGraph Là một class trừu tượng (abstract class) kế thừa từ class interface IGraph, sử dụng danh sách kề (adjacency list) để lưu trữ cấu trúc đồ thị.

2.4.1 Edge

```
class Edge{
private:
    VertexNode* from;
    VertexNode* to;
    float weight;
    friend class VertexNode;
    friend class AbstractGraph;
public:
    Edge(){}
    Edge(VertexNode* from, VertexNode* to, float weight=0){
        this->from = from;
        this->to = to;
        this->weight = weight;
    }

    bool equals(Edge* edge){
        //TODO
    }

    static bool edgeEQ(Edge*& edge1, Edge*& edge2){
        return edge1->equals(edge2);
    }
    string toString(){
        stringstream os;
        os << "E("
                << this->from->vertex
                << ","
                << this->to->vertex
                << ","
                << this->weight
                << ")";
        return os.str();
    }
};
```

2.3 IGraph

class IGraph 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

Có thể bỏ qua or đọc trong pdf

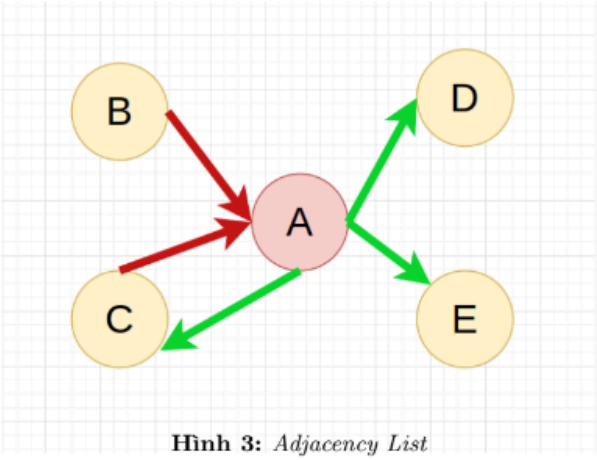
1. **Các thuộc tính:**
   - **VertexNode* from:** Con trỏ tới đỉnh nguồn của cạnh.
   - **VertexNode* to:** Con trỏ tới đỉnh đích của cạnh.
   - **float weight:** Trọng số của cạnh. Mặc định là 0 nếu không được chỉ định.
2. **bool equals(Edge* edge)** So sánh cạnh hiện tại với một cạnh khác edge. Trả về true nếu cả hai cạnh có cùng đỉnh nguồn và đỉnh đích, và false nếu không. **gọi hàm equals trong VertexNode để so sánh 2 đỉnh from và to**
3. **Các hàm còn lại** đọc pdf **241_CO2003__Assignment_3**

```
class VertexNode{
private:
    template<class U>
    friend class UGraphModel; //UPDATED: added
    T vertex;
    int inDegree_, outDegree_;
    DLinkedList<Edge*> adList;
    friend class Edge;
    friend class AbstractGraph;

    bool (*vertexEQ)(T&, T&);
    string (*vertex2str)(T&);

public:
    VertexNode():adList(&DLinkedList<Edge*>::free, &Edge::edgeEQ){}
    VertexNode(T vertex, bool (*vertexEQ)(T&, T&), string (*vertex2str)(T&))
        :adList(&DLinkedList<Edge*>::free, &Edge::edgeEQ){
        this->vertex = vertex;
        this->vertexEQ = vertexEQ;
        this->vertex2str = vertex2str;
        this->outDegree_ = this->inDegree_ = 0;
    }
    T& getVertex(){
        return vertex;
    }
    void connect(VertexNode* to, float weight=0){
        //TODO
    }
    DLinkedList<T> getOutwardEdges(){
        //TODO
    }

    Edge* getEdge(VertexNode* to){
        //TODO
    }
    bool equals(VertexNode* node){
        //TODO
    }

    void removeTo(VertexNode* to){
        //TODO
    }
    int inDegree(){
        //TODO
    }
    int outDegree(){
        //TODO
    }
    string toString(){
        stringstream os;
        os << "V("
                << this->vertex << ", "
                << "in: " << this->inDegree_ << ", "
                << "out: " << this->outDegree_ << ")";
```

```
        return os.str();
    }
};
```

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. **void connect(VertexNode* to, float weight = 0):** Kết nối đỉnh hiện tại với đỉnh khác to bằng cách tạo một cạnh với trọng số (mặc định là 0). **thêm vào cuối danh sách adList và cập nhật outDegree_**
3. **DLinkedList<T> getOutwardEdges():** Trả về danh sách các cạnh đi ra từ đỉnh hiện tại. **Tạo ra danh sách trả về, sau đó duyệt qua các phần tử trong adList nếu Edge đó có from bằng với vertex thì đẩy vào danh sách liên kết**
4. **Edge* getEdge(VertexNode* to):** Trả về con trỏ tới cạnh nối từ đỉnh hiện tại tới đỉnh to. Trả về nullptr nếu không tìm thấy cạnh. **Duyệt qua các phần tử trong adList nếu Edge đó có to bằng với to truyền vào thì trả về**
5. **bool equals(VertexNode* node):** So sánh đỉnh hiện tại với một đỉnh khác node dựa trên hàm vertexEQ. **dùng vertexEQ và vertex**
6. **void removeTo(VertexNode* to):** Xóa cạnh nối từ đỉnh hiện tại tới đỉnh to. **Duyệt qua các phần tử trong adList nếu Edge đó có to bằng với to thì gọi hàm xóa trong danh sách liên kết tại vị trí đó và giảm outDegree_**
7. **int inDegree():** Trả về bậc vào của đỉnh.
8. **int outDegree():** Trả về bậc ra của đỉnh.

Ví Dụ:



Hình 3: *Adjacency List*

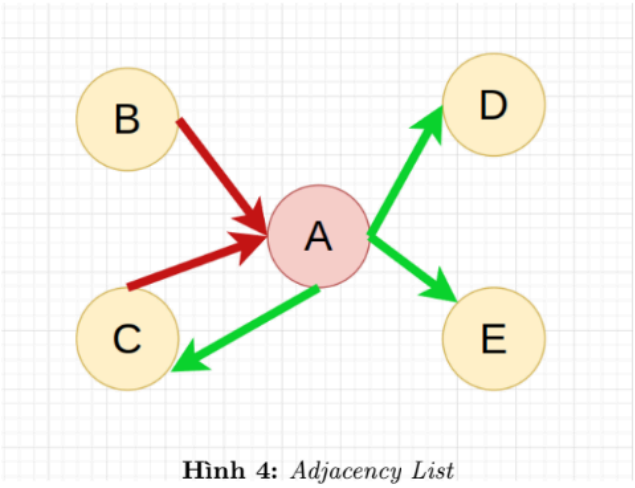
1. **Các thuộc tính:**
    - **T vertex=** A
    - **int inDegree_ =** 2
    - **int outDegree_ =** 3
    - **DLinkedList<Edge*> adList=** (A,C)->(A,D)->(A,E)->null
2. **void connect(VertexNode* to, float weight = 0):**
    - connect(B)
    - **DLinkedList<Edge*> adList=** (A,C)->(A,D)->(A,E)->(A,B)->null
    - **int outDegree_ =** 4
3. **DLinkedList<T> getOutwardEdges():**
    - **return** C->D->E->null
4. **Edge* getEdge(VertexNode* to):**
    - getEdge(C) => (A,C)
    - getEdge(B) => null
5. **void removeTo(VertexNode* to):**
    - connect(C)
    - **DLinkedList<Edge*> adList=** (A,D)->(A,E)->null
    - **int outDegree_ =** 2
6. **int inDegree():** 2
7. **int outDegree():** 3

2.4.3 Abstract Graph

```
template<class T>
class AbstractGraph: public IGraph<T>{
protected:
    DLinkedList<VertexNode*> nodeList;
    bool (*vertexEQ)(T&, T&);
    string (*vertex2str)(T&);
    ...
    ...
    ...
    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) {
        //TODO
    }
    virtual bool contains(T vertex){
        //TODO
    }
    virtual float weight(T from, T to){
        //TODO
    }
    virtual DLinkedList<T> getOutwardEdges(T from){
        //TODO
    }
    virtual DLinkedList<T>  getInwardEdges(T to){
        //TODO
    }
    virtual int size() {
        //TODO
    }
    virtual bool empty(){
        //TODO
    };
    virtual void clear(){
        //TODO
    }
    virtual int inDegree(T vertex){
        //TODO
    }
    virtual int outDegree(T vertex){
        //TODO
    }
    virtual DLinkedList<T> vertices(){
        //TODO
    }
    virtual bool connected(T from, T to){
        //TODO
    }
    ...
    ...
};
```

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. **Phương thức có sẵn**
    - **getVertexNode** Tìm kiếm và trả về con trỏ đến nút VertexNode chứa đỉnh vertex trong đồ thị. Nếu không tìm thấy, trả về nullptr.
    - **vertex2Str** : Chuyển đổi một nút VertexNode thành chuỗi biểu diễn của đỉnh thông qua hàm vertex2str.
    - **edge2Str** : Chuyển đổi một đối tượng Edge thành chuỗi biểu diễn của cạnh, bao gồm thông tin về đỉnh nguồn và đỉnh đích
    - **Contructor VertexNode có param** dùng để khởi tạo VertexNode mới, dùng trong add
    - **getEdge trong VertexNode** dùng để lấy được trọng số weight
    - **inDegree trong VertexNode** bậc vào
    - **outDegree trong VertexNode** bậc ra
3. **Các phương thức cần hiện thực đọc pdf thầy, kì này thầy ghi khá rõ và coi ví dụ ở phần sau**

Ví Dụ:



Hình 4: *Adjacency List*

1. **Các thuộc tính:**
   - **DLinkedList<VertexNode*> nodeList=**

     ```
     A : (A,C)->(A,D)->(A,E)->null
     B : (B,A)->null
     C : (C,A)->null
     D : null
     E : null
     null
     ```

2. **virtual void add(T vertex = F):**

   ```
   A : (A,C)->(A,D)->(A,E)->null
   B : (B,A)->null
   C : (C,A)->null
   D : null
   E : null
   F : null
   null
   ```

3. **virtual bool contains(T vertex):**
   - add(A) => true
   - add(X) => false
4. **virtual float weight(T from, T to):**
   - weight(A,B) => 0
   - weight(B, X) => VertexNotFoundException(vertex2Str(X))
   - weight(B, C) => EdgeNotFoundException(edge2Str(Edge(B,C)))
5. **virtual DLinkedList<T> getOutwardEdges(T from = A):**

   ```
   C->B->E->null
   ```

6. **virtual DLinkedList<T> getInwardEdges(T from = A):**

   ```
   B->C->null
   ```

7. **virtual DLinkedList<T> vertices():**

   ```
   A->B->C->D->E->null
   ```

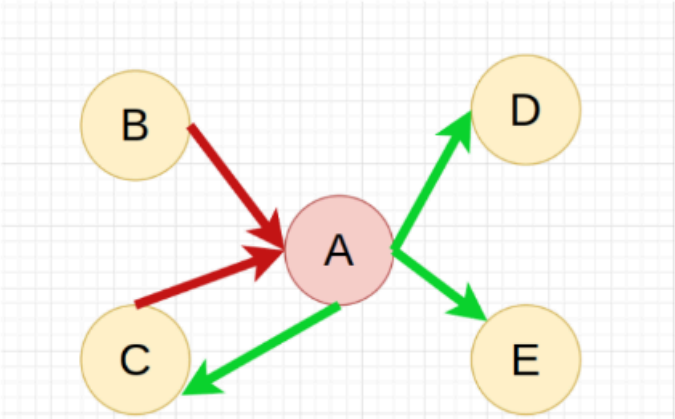
2.5 DGraphModel

class DGraphModel là một hiện thực cụ thể của đồ thị có hướng (Directed Graph) dựa trên abstract class AbstractGraph

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

    void connect(T from, T to, float weight=0){
        //TODO
    }
    void disconnect(T from, T to){
        //TODO
    }
    void remove(T vertex){
        //TODO
    }

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

Các phương thức cần hiện thực đọc pdf thầy, kì này thầy ghi khá rõ và coi ví dụ ở phần sau

Ví Dụ:



Hình 5: *Adjacency List*

1. **Các thuộc tính:**
   - **DLinkedList<VertexNode*> nodeList=**

     `A : (A,C)->(A,D)->(A,E)->null`

```
B : (B,A)->null
C : (A,C)->null
D : null
E : null
null
```

2. **void connect(T from = B, T to = C, float weight = 0)**

```
A : (A,C)->(A,D)->(A,E)->null
B : (B,A)->(B,C)->null
C : (A,C)->null
D : null
E : null
null
```

3. **void disconnect(T from = B, T to = A)**

```
A : (A,C)->(A,D)->(A,E)->null
B : null
C : (A,C)->null
D : null
E : null
null
```

4. **void remove(T vertex = C)**

```
A : (A,D)->(A,E)->null
B : (B,A)->null
D : null
E : null
null
```

2.6 UGraphModel

class UGraphModel là một hiện thực cụ thể của đồ thị vô hướng (Undirected Graph) dựa trên abstract class AbstractGraph

Các phương thức cần hiện thực đọc pdf thầy, kì này thầy ghi khá rõ và xem các ví dụ trước đó