Mục lục

1 Cách Chạy test case

Các file code trước đó

- DLinkedList.h - BTL1
- xMap.h - BTL2
- AbstractGraph.h - task 1
- DGraphModel.h - task 1
- UGraphModel.h - task 1

Các file hiện thực

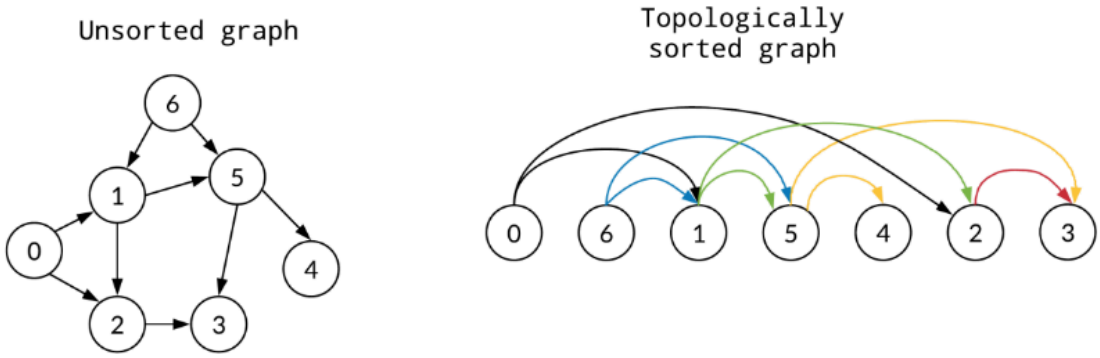
- TopoSorter.h - task 2
- Queue.h - task 2 (Hiện thực các hàm dùng trong bfs thôi)
- Stack.h - task 2 (Hiện thực các hàm dùng trong dfs thôi)

Test Case thì anh có đưa cách chạy trong file main và 1 số test case trước của tasks 1 đã cập nhật và task 2 (anh sẽ đăng thêm sau)

PHẦN SORT VÀ DLinkedListSE.h ANH SẼ THÔNG BÁO CẬP NHẬT SAU

2 Giải Thuật Topo Sort

Topo Sort (sắp xếp topo) là một kỹ thuật sắp xếp các đỉnh của đồ thị có hướng (Directed Acyclic Graph - DAG) sao cho, với mọi cung (u, v), đỉnh u luôn xuất hiện trước đỉnh v trong danh sách sắp xếp. Đây là một kỹ thuật thường được sử dụng trong các bài toán như xác định thứ tự công việc, lập lịch trình, hoặc phân tích phụ thuộc.



Hình 1: *Togo Sort*

Ý tưởng

1. Đảm bảo rằng đồ thị là DAG (không có chu trình).
2. Bắt đầu từ các đỉnh không có cung đi vào (*in-degree* = 0).
3. Sử dụng phương pháp duyệt lần lượt để xóa các đỉnh đã xử lý khỏi đồ thị và giảm bậc (*in-degree*) của các đỉnh liên quan.

```cpp
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) {
    this->graph = graph;
    this->hash_code = hash_code;
  }
  DLinkedList<T> sort(int mode = 0, bool sorted = true) {
    return (mode == BFS) ? bfsSort(sorted) : dfsSort(sorted);
  }
};  // TopoSorter
template <class T>
int TopoSorter<T>::DFS = 0;
template <class T>
int TopoSorter<T>::BFS = 1;
```

Giải Thích

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. **Hàm** `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.

3 Topo Sort Dùng dfs

3.1 Lý Thuyết

Giải thuật Topo Sort dùng DFS với Stack không đệ quy thay thế đệ quy bằng một stack để quản lý trạng thái duyệt. Cách tiếp cận này hữu ích trong các môi trường hạn chế đệ quy hoặc để tránh tràn ngăn xếp trong trường hợp đồ thị lớn.

1. **Khởi tạo dữ liệu**
   - **Graph Representation**: Đồ thị được biểu diễn dưới dạng danh sách kề (adjacency list), tức là mỗi đỉnh sẽ chứa danh sách các đỉnh kề.
   - **Visited Array**: Một mảng boolean `visited[]` sẽ được sử dụng để đánh dấu các đỉnh đã được thăm hay chưa. Mảng này sẽ có kích thước bằng số lượng đỉnh của đồ thị.
   - **Result Stack**: Một ngăn xếp `resultStack` sẽ lưu trữ các đỉnh theo thứ tự Topological Sort. Các đỉnh sẽ được đưa vào `resultStack` sau khi tất cả các đỉnh kề của chúng đã được duyệt.
   - **DFS Stack**: Một ngăn xếp phụ `dfsStack` dùng để hỗ trợ việc duyệt theo chiều sâu của đồ thị mà không cần dùng đệ quy.
2. **Duyệt qua tất cả các đỉnh**

   Ta sẽ duyệt tất cả các đỉnh trong đồ thị. Đối với mỗi đỉnh chưa được thăm, ta sẽ bắt đầu duyệt từ đỉnh đó bằng cách sử dụng stack `dfsStack`.
3. **Duyệt đồ thị theo chiều sâu (DFS)**
   - Đỉnh đầu tiên được lấy từ stack `dfsStack`. Nếu đỉnh chưa được thăm, ta đánh dấu đỉnh đó là đã thăm.
   - Sau đó, ta sẽ kiểm tra tất cả các đỉnh kề của đỉnh hiện tại. Nếu đỉnh kề chưa được thăm, đẩy chúng vào `dfsStack`.
   - Khi tất cả các đỉnh kề của đỉnh hiện tại đã được duyệt, đỉnh này sẽ được đưa vào `resultStack`.
4. **Xử lý kết quả**

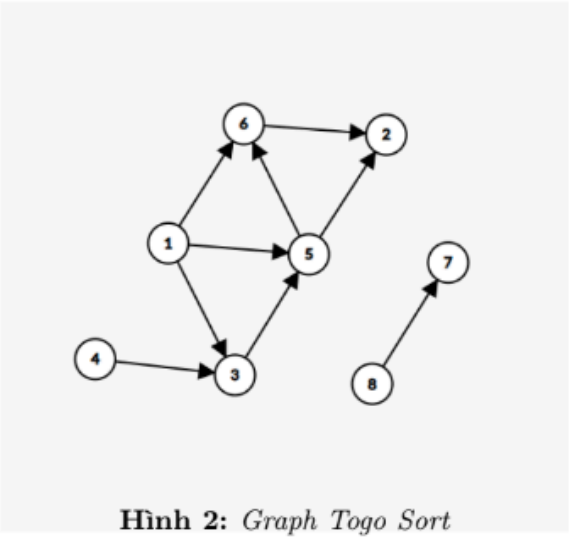
   Sau khi duyệt qua tất cả các đỉnh, `resultStack` sẽ chứa các đỉnh theo thứ tự ngược lại của sắp xếp topo. Vì vậy, ta cần chuyển các đỉnh trong `resultStack` sang một vector `result[]` để trả về kết quả đúng thứ tự sắp xếp topo.

Code dùng stack (dùng đệ quy vô làm harmony thở oxi)

```cpp
// Hàm thực hiện DFS không đệ quy
void dfsIterative(int start, const vector<vector<int>>& graph, vector<bool>&
    visited, stack<int>& resultStack) {
    resultStack.push(start);

    while (!resultStack.empty()) {
        int u = resultStack.top();

        if (!visited[u]) {
            visited[u] = true; // Đánh dấu đã thăm
        }

        bool allNeighborsVisited = true;
        for (int v : graph[u]) {
            if (!visited[v]) {
                resultStack.push(v); // Đẩy đỉnh kề vào stack
                allNeighborsVisited = false;
                break; // Tiếp tục duyệt đỉnh kề
```

```
            }
        }

        if (allNeighborsVisited) {
            resultStack.pop();
            resultStack.push(u); // Đẩy đỉnh vào stack kết quả
        }
    }
}

// Hàm thực hiện Topological Sort
vector<int> topoSort(const vector<vector<int>>& graph) {
    int n = graph.size(); // Số lượng đỉnh
    vector<bool> visited(n, false); // Mảng đánh dấu đã thăm
    stack<int> resultStack;

    // Duyệt qua tất cả các đỉnh
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            dfsIterative(i, graph, visited, resultStack);
        }
    }

    // Kết quả sắp xếp topo
    vector<int> result;
    while (!resultStack.empty()) {
        result.push_back(resultStack.top());
        resultStack.pop();
    }
    return result;
}
```

Ví Dụ



Hình 2: *Graph Togo Sort*

. **Đỉnh 1:**

- Khởi động từ đỉnh 1:
  - Đẩy 1 vào `resultStack`: `resultStack` = [1].
  - Đánh dấu `visited[1] = true`.

- Duyệt các đỉnh kề của 1:
  - Đỉnh kề đầu tiên là 3, đẩy 3 vào `resultStack`: `resultStack` = [1, 3].
  - Đánh dấu `visited[3] = true`.
- Duyệt các đỉnh kề của 3:
  - Đỉnh kề đầu tiên là 5, đẩy 5 vào `resultStack`: `resultStack` = [1, 3, 5].
  - Đánh dấu `visited[5] = true`.
- Duyệt các đỉnh kề của 5:
  - Đỉnh kề đầu tiên là 2, đẩy 2 vào `resultStack`: `resultStack` = [1, 3, 5, 2].
  - Đánh dấu `visited[2] = true`.
- Đỉnh 2 không có đỉnh kề:
  - Pop 2 khỏi `resultStack`, đẩy vào `resultStack`: `resultStack` = [1, 3, 5], `resultStack` = [2].
- Quay lại đỉnh 5, xử lý đỉnh kề còn lại (6):
  - Đẩy 6 vào `resultStack`: `resultStack` = [1, 3, 5, 6].
  - Đánh dấu `visited[6] = true`.
- Duyệt các đỉnh kề của 6:
  - Đỉnh kề duy nhất là 2, nhưng 2 đã thăm, bỏ qua.
- Đỉnh 6 không còn đỉnh kề:
  - Pop 6 khỏi `resultStack`, đẩy vào `resultStack`: `resultStack` = [1, 3, 5], `resultStack` = [2, 6].
- Quay lại đỉnh 5, không còn đỉnh kề:
  - Pop 5 khỏi `resultStack`, đẩy vào `resultStack`: `resultStack` = [1, 3], `resultStack` = [2, 6, 5].
- Quay lại đỉnh 3, không còn đỉnh kề:
  - Pop 3 khỏi `resultStack`, đẩy vào `resultStack`: `resultStack` = [1], `resultStack` = [2, 6, 5, 3].

- Quay lại đỉnh 1, không còn đỉnh kề:
  - Pop 1 khỏi `resultStack`, đẩy vào `resultStack`: `resultStack` = [], `resultStack` = [2, 6, 5, 3, 1].

2. **Đỉnh 4:**
   - Khởi động từ đỉnh 4:
     - Đẩy 4 vào `resultStack`: `resultStack` = [4].
     - Đánh dấu `visited[4] = true`.
   - Duyệt các đỉnh kề của 4:
     - Đỉnh kề đầu tiên là 3, nhưng 3 đã thăm, bỏ qua.
   - Đỉnh 4 không còn đỉnh kề:
     - Pop 4 khỏi `resultStack`, đẩy vào `resultStack`: `resultStack` = [], `resultStack` = [2, 6, 5, 3, 1, 4].
3. **Đỉnh 8:**
   - Khởi động từ đỉnh 8:

- Duyệt các đỉnh kề của 8:
  - Đỉnh kề đầu tiên là 7, đẩy 7 vào `resultStack`: `resultStack` = [8, 7].
  - Đánh dấu `visited[7] = true`.
- Đỉnh 7 không còn đỉnh kề:
  - Pop 7 khỏi `resultStack`, đẩy vào `resultStack`: `resultStack` = [8], `resultStack` = [2, 6, 5, 3, 1, 4, 7].
- Quay lại đỉnh 8, không còn đỉnh kề:
  - Pop 8 khỏi `resultStack`, đẩy vào `resultStack`: `resultStack` = [], `resultStack` = [2, 6, 5, 3, 1, 4, 7, 8].

3.2 Các API có thể dùng

1. **xMap or XHashMap**
   - **Contructor truyền hash_code**
   - **V put(K key, V value)**
   - **V& get(K key)**
   - **void clear()**
2. **Stack**
   - **void push(T item)**
   - **T pop()**
   - **T& peek()**
   - **bool empty()**
   - **void clear()**
3. **DLinkedList**
   - **Iterator và BWDIterator**
4. **DGraphModel**
   - **DLinkedList<T> getOutwardEdges(T from)**
   - **DLinkedList<T> vertices()**

STACK DÙNG HÀM GÌ THÌ HIỆN THỰC HÀM NẤY CÁC HÀM ANH SÀI PUSH, POP, PEEK, EMPTY

4 Topo Sort Dùng bfs

4.1 Lý Thuyết

Đây là một phương pháp để thực hiện sắp xếp topo trên đồ thị có hướng (directed graph) sử dụng thuật toán tìm kiếm theo chiều rộng (BFS). Giải thuật này dựa trên việc duyệt qua các đỉnh theo thứ tự có thể duy trì một danh sách của các đỉnh đã được xử lý, dựa trên số lượng cạnh đến các đỉnh này (được gọi là độ vào, in-degree).

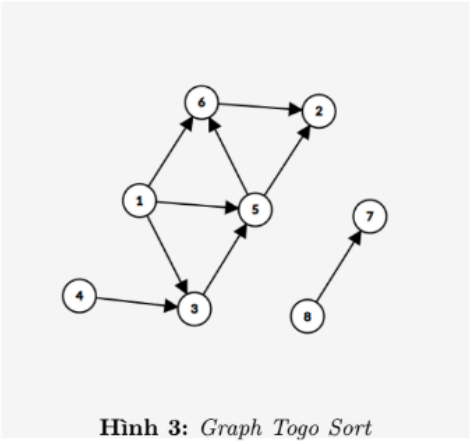
1. **Khởi tạo**:
    - Tạo một danh sách lưu trữ số lượng cạnh đến mỗi đỉnh (in-degree) từ các đỉnh khác. Nếu đỉnh không có cạnh đến từ đỉnh khác, độ vào của nó sẽ là 0.
    - Tạo một hàng đợi (queue) để lưu trữ các đỉnh có độ vào bằng 0, nghĩa là những đỉnh không có bất kỳ đỉnh kề nào trỏ tới.
    - Một danh sách kết quả (result) để lưu trữ các đỉnh theo thứ tự topo.
2. **Bước 1: Khởi tạo các giá trị**:
    - Tính toán độ vào (in-degree) của tất cả các đỉnh trong đồ thị.
    - Thêm tất cả các đỉnh có độ vào bằng 0 vào trong hàng đợi (queue).
3. **Bước 2: Duyệt qua đồ thị**: Lặp lại quá trình khi hàng đợi chưa rỗng:
    - Lấy ra một đỉnh từ hàng đợi (dequeue).
    - Thêm đỉnh đó vào danh sách kết quả.
    - Duyệt qua tất cả các đỉnh kề của đỉnh vừa lấy ra:
        - Giảm độ vào của các đỉnh kề.
        - Nếu độ vào của một đỉnh kề trở thành 0, thêm nó vào hàng đợi (queue).

Code dùng queue

```cpp
// Hàm để thực hiện sắp xếp topo sử dụng BFS (Kahn's Algorithm)
vector<int> topologicalSortBFS(const vector<vector<int>>& graph, int numVertices) {
    // Độ vào (in-degree) của các đỉnh
    vector<int> inDegree(numVertices, 0);

    // Tính độ vào cho mỗi đỉnh
    for (int u = 0; u < numVertices; ++u) {
        for (int v : graph[u]) {
            inDegree[v]++;
        }
    }

    // Khởi tạo hàng đợi (queue) và thêm các đỉnh có độ vào bằng 0 vào hàng đợi
    queue<int> q;
    for (int i = 0; i < numVertices; ++i) {
        if (inDegree[i] == 0) {
            q.push(i);
        }
    }

    vector<int> topoOrder;  // Kết quả sắp xếp topo
    while (!q.empty()) {
        int node = q.front();
        q.pop();
```

```cpp
        topoOrder.push_back(node);

        // Duyệt các đỉnh kề của node
        for (int neighbor : graph[node]) {
            inDegree[neighbor]--;
            if (inDegree[neighbor] == 0) {
                q.push(neighbor);
            }
        }
    }
    return topoOrder;
}
```

Ví Dụ



Hình 3: *Graph Togo Sort*

1. **Khởi tạo**
   - Số đỉnh: 8 (đánh số từ 0 đến 7).
   - Đồ thị được biểu diễn dưới dạng danh sách kề: $\text{graph}[u] = \{v_1, v_2, \dots\}$.
   - Tính độ vào (*in-degree*) cho mỗi đỉnh. Ban đầu, tất cả độ vào của các đỉnh là 0.
2. **Tính độ vào**
   - Độ vào của đỉnh 0 là 0.
   - Độ vào của đỉnh 1 là 1 (do $4 \to 3$).
   - Độ vào của đỉnh 2 là 3 (do $5 \to 2$, $6 \to 2$).
   - Độ vào của đỉnh 3 là 1 (do $4 \to 3$).
   - Độ vào của đỉnh 4 là 0.
   - Độ vào của đỉnh 5 là 2 (do $1 \to 5$, $3 \to 5$).
   - Độ vào của đỉnh 6 là 2 (do $1 \to 6$, $5 \to 6$).
   - Độ vào của đỉnh 7 là 1 (do $8 \to 7$).
3. **Khởi tạo hàng đợi** Tất cả các đỉnh có độ vào bằng 0 được đưa vào hàng đợi (queue). Ở đây, đỉnh 1 và đỉnh 4 có độ vào bằng 0, nên $\text{queue} = [0, 3]$.

4. **Duyệt đồ thị Bước đầu tiên**: Lấy đỉnh từ hàng đợi và xử lý.
    - Pop đỉnh 0:

        - Đẩy các đỉnh kề của 0 vào hàng đợi (3, 5, 6).
        - Giảm độ vào của các đỉnh kề 3, 5, 6 và đưa vào hàng đợi nếu độ vào của chúng bằng 0.
        - queue = $[3, 5, 6]$.
    - Pop đỉnh 3:
        - Đẩy các đỉnh kề của 3 vào hàng đợi (5).
        - Giảm độ vào của đỉnh 5 và đưa vào hàng đợi nếu độ vào của nó bằng 0.
        - queue = $[5, 6]$.
    - Pop đỉnh 5:
        - Đẩy các đỉnh kề của 5 vào hàng đợi (2, 6).
        - Giảm độ vào của đỉnh 2, đưa vào hàng đợi nếu độ vào của nó bằng 0.
        - queue = $[6, 2]$.
    - Pop đỉnh 6:
        - Đẩy các đỉnh kề của 6 vào hàng đợi (2).
        - Giảm độ vào của đỉnh 2, đưa vào hàng đợi nếu độ vào của nó bằng 0.
        - queue = $[2]$.

- Pop đỉnh 2:
  - Đỉnh 2 không có đỉnh kề.
  - queue = [].
- Pop đỉnh 4:
  - Đỉnh 4 không có đỉnh kề.
  - queue = [].

4.2 Các API có thể dùng

1. **Hàm** `vertex2inDegree`
   - **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.
2. **Hàm** `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. **xMap or XHashMap**
   - **Contructor truyền hash_code**
   - **V put(K key, V value)**
   - **V& get(K key)**
   - **void clear()**
4. **Queue**
   - **void push(T item)**

- **T pop()**
- **T& peek()**
- **bool empty()**
- **void clear()**

5. **DLinkedList**
   - **Iterator và BWDIterator**
6. **DGraphModel**
   - **DLinkedList<T> getInwardEdges(T to)**
   - **DLinkedList<T> vertices()**
   - **int inDegree(T vertex)**

QUEUE DÙNG HÀM GÌ THÌ HIỆN THỰC HÀM NẤY CÁC HÀM ANH SÀI PUSH, POP, EMPTY

5 Liên Quan tới sort và DLinkedListSE

UPDATE SAU