Projet sur les graphes

Projet à rendre avant le lundi 27 janvier 2025 sous la forme d'un repository github de nom `graph` auquel vous me mettez collaborateur.

introduction

Nos graphes seront composés de sommets et d'arêtes étiquetées et orientées connectant ces sommets.

Qu'il s'agisse de graphes pour représenter des réseaux aériens, routiers, ferrés, électroniques, sociaux, d'eau ou de télécommunications... nous sommes amenés à nous poser des questions très générales, comme: les sommets atteignables à partir d'un sommet donné; les plus courts chemins entre les sommets du graphe...

Ce modèle a donné lieu à une théorie informatique où des structures de données et des algorithmes ont été proposés pour résoudre ces questions: des parcours de graphes (*depth first search* et *breadth first search*), des calculs de plus courts chemins (*Dijkstra*, *Floyd-Warshall*)...

mise en garde

Le sujet est simple et très classique et vous trouverez sur Internet ou par des IA de très bons codes déjà implémentés pour les graphes, ceci est un projet pour que vous progressiez en programmation aussi je vous demande de ne prendre de code tout fait mais de profiter de ce sujet pour vous entraînez à réfléchir sur les structures de données et les algos.

description des graphes

Nos graphes sont dirigés i.e. les transitions vont d'un sommet de départ à un sommet d'arrivée, on ne peut pas *remonter* une transition.

Un sommet `sj` est adjacent à un sommet `si` si il existe une arête reliant `si` à `sj` dont `si` est le sommet de départ `si -> sj`

Une arête des graphes contient une valeur qui sera un `double` positif.

représentation d'un graphe en mémoire

Il existe deux manières principales pour représenter un graphe dans la mémoire d'un programme.

1. la liste d'adjacence:
    - *vous avez un conteneur de sommets et chaque case de ce conteneur contient les sommets adjacents à ce sommet*
    - cette structure permet, à partir d'un sommet, d'accéder *rapidement* à la liste des sommets adjacents à ce sommet
    - pour stocker vos sommets, si ils sont numérotés vous pouvez les mettre dans un vecteur (`std::vector`) si ils sont nommés par des chaînes de caractères (`std::string`) vous pouvez les mettre dans un dictionnaire `std::unordered_map` (voir exemple de dictionnaire annexe)

2. par matrice d'adjacence : *vous avez une matrice où `Mat[i, j]` est l'arête qui va du sommet `si` au sommet `sj`. Si il n'y a pas d'arrête entre `si` et `sj` vous devez mettre une valeur qui n'existe pas dans le graphe.*

quelle structure choisir ?

Ça dépend: suivant la représentation choisie, la place mémoire occupée par le graphe ainsi que les algorithmes et leur complexité seront différents.

Le projet

question 1

Implémentez la représentation par sommets adjacents d'un graphe.

question 2

Implémentez le parcours du graphe en profondeur de manière récursive.

Parcourir un graphe consiste à *passer* par tous ses sommets et toutes ses arêtes.

Le parcours en **profondeur** depuis le sommet `si` d'un graphe, s'exprime très simplement d'une manière récursive:

1. vous lancez la *visite* sur chacun des sommets `si` du graphe
    - si `si` n'a pas encore été *visité*
        - vous *marquez* le sommet `si` comme ayant été *visité* (pour qu'il ne soit pas *revisité*)
        - vous considérez chacun des sommets `sj` adjacents à `si`
            - c'est la première fois que vous voyez l'arête `si -> sj`
              vous l'affichez avec sa valeur
            - si le sommet `sj` n'a pas encore été visité, vous lancez sa visite
    - quand tous les sommets atteignables à partir de `si` ont été visités, le parcours en profondeur depuis le sommet `si` du graphe se termine, vous continuez la visite des sommets du graphe (1.)

question 3

Si on réfléchit, pendant le parcours récursif, les sommets en cours de visite, sont *conservés* sur la pile d'exécution du programme et c'est cela qui réalise un parcours en profondeur *récursif*.

Si, au lieu de laisser la pile d'exécution du programme stocker les sommets, on utilise un conteneur avec un comportement de pile (`std::stack` voir annexe), on peut implémenter un parcours en profondeur *itératif*.

Implémentez le parcours en profondeur de manière itérative dans vos deux représentations de graphes.

question 4

Maintenant, si on remplace ce mécanisme de pile par un de *file* (*file*: premier entré = premier sorti `std::queue` voir annexe) l'algorithme va changer et devenir un parcours en largeur d'abord.

Implémentez le parcours en largeur.

question subsidiaire

Implémentez un graphe sous forme de matrice d'adjacence et implémentez l'Algorithme de Floyd-Warshall https://fr.wikipedia.org/wiki/AlgorithmedeFloyd-Warshall qui calcule les plus courts chemins entre tous les sommets d'un graphe.

Annexe

exemple de main

Ce sont juste des exemples, vous faites comme vous voulez !

```
// vos classes ici
// exemple d'un main

int main() {
    Graph g;
    // plutôt dictionnaire
    g.addEdge("Paris", "Lyon", 100);
    g.addEdge("Paris", "Marseille", 150);
    g.addEdge("Lyon", "Nice", 80);
    g.addEdge("Marseille", "Nice", 70);
    g.addEdge("Nice", "Paris", 200);
    g.printGraph();

    // ou plus c++
    std::cout << g << std::endl; // redéfinir operator<<

    g.dfs(); // parcours du graphe en profondeur d'abord
             // depth first search
    g.bfs(); // parcours du graphe en largeur d'abord
             // breath first search

    // même chose à partir d'un sommet
    g.dfs("Paris");
    g.bfs("Paris");

    return 0;
}
```

```
// vos classes ici
// exemple d'un main

int main() {
    Graph g;
    // plutôt vecteur
    g.addEdge(1, 2, 100);
    g.addEdge(1, 3, 150);
```

```
    g.addEdge(2, 0, 80);
    g.addEdge(3, 0, 70);
    g.addEdge(0, 1, 200);
    g.printGraph();

    // ou plus c++
    std::cout << g << std::endl; // redéfinir operator<<

    g.dfs(); // parcours du graphe en profondeur d'abord
             // depth first search
    g.bfs(); // parcours du graphe en largeur d'abord
             // breath first search

    // même chose à partir d'un sommet
    g.dfs(1);
    g.bfs(1);

    return 0;
}
```

un dictionnaire en C++

```
#include <iostream>
#include <unordered_map>
#include <string>
#include <vector>

int main()
{
    // on crée un dictionnaire pour stocker les noms des sommets
    // et ici un simple entier
    // (mais il faudrait mettre un conteneur d'arrêtes adjacentes)
    std::unordered_map<std::string, int> my_graph;

    // on donne des valeurs aux sommets
    // (mais il faudrait leur donner leurs arrêtes adjacentes)
    my_graph["s1"] = 1;
    my_graph["s2"] = 2;
    my_graph["s3"] = 3;

    // On accède à un sommet
    std::cout << my_graph["s1"] << std::endl;

    // On accède à tous les sommets
    // notons que les couples <key, value> sont des std::pair

    // 1) accès par first et second
    for (auto &e : my_graph)
    {
        std::cout << "key: " << e.first
                  << " value: " << e.second << std::endl;
```

```
    }

    // 2) accès par std::get<0> et std::get<1>
    // à préférer à first et second
    for (auto &e : my_graph)
    {
        std::cout << "key: " << std::get<0>(e)
                  << " value: " << std::get<1>(e) << std::endl;
    }

    return 0;
}
```

une pile en c++

```
#include <iostream>
#include <stack>

int main() {
    // on crée une pile
    std::stack<int> stack;

    // on empile des éléments dans la pile
    stack.push(10);
    stack.push(20);
    stack.push(30);

    // on affiche le sommet de la pile
    std::cout << "haut de la pile " << stack.top() << std::endl;

    // on dépile un élément
    // (on remarque que la fonction pop() ne retourne rien dans cette
bibliothèque)
    stack.pop();
    std::cout << "haut de pile après pop" << stack.top() << std::endl;

    // on vérifie si la pile est vide
    if (stack.empty()) {
        std::cout << "elle est vide" << std::endl;
    } else {
        std::cout << "elle n'est pas vide" << std::endl;
    }

    return 0;
}
```

une *file* (first in first out conteneur) en c++

```cpp
#include <iostream>
#include <queue>

int main()
{
    // on crée une file (first in first out conteneur)
    std::queue<int> fifo;

    // on ajoute des éléments à l'arrière de la file
    fifo.push(10);
    fifo.push(20);
    fifo.push(30);

    // son mécanisme consiste à ajouter des éléments à l'arrière
    // et à les retirer à l'avant avec pop
    while (!fifo.empty())
    {
        std::cout << "front " << fifo.front() << std::endl;
        fifo.pop();
    }

    return 0;
}
```