Développement Efficace

Guilherme Dias da Fonseca

TP3

Ce TP est à rendre et sera noté. Vous devez rendre votre source dans un fichier zip avec soit un `Makefile`, soit un script bash `build.sh` pour compiler tous les exercices sur linux. Écrivez vos noms dans les commentaires, svp. Ne modifiez pas les fichiers que j'ai fournis et téléchargez toujours la dernière version parce qu'ils peuvent être modifiés. Vous devez rendre un fichier zip avec :

1. Le code source avec votre nom dans la première ligne
2. Un script bash appelé `build.sh` pour compiler votre code sur Linux.

Pensez que j'ai beaucoup de TPs à corriger, alors pour que je puisse rendre cette tache plus efficace je vais coder des logiciels pour faire quelques parties automatiquement. Pour ça, il faut avoir exactement les bons noms des fichiers (minuscule et majuscule c'est pas pareil). Parfois, je vois des trucs très bizarres comme des fichiers rar renommé zip pour qu'Ametice l'accepte et bien sûr que le script que j'écris pour décompresser les fichiers ne marche pas ! Le non respect des consignes peut être pénalisé.

Je vous rappelle que copier du code est interdit et risque des sanctions très sévères. Je vous prévois que je vais passer votre code dans un détecteur de plagiat de code C++ avant la correction. Vous avez le droit d'utiliser l'IA sur 2 des 6 exercices au choix, mais :

— Vous devez indiquer quelle partie du code a été produite par l'IA.
— Vous êtes responsable pour vérifier que la solution proposé par l'IA est juste (pas simplement qu'elle passe les tests).
— Vous devez être capable de m'expliquer le code.

Le but de ce TP est d'utiliser du `std::unordered_map` et du `std::unordered_set` pour représenter et réaliser des opérations et des algorithmes sur un graphe. Vous devez écrire la classe `Graph<Vertex>` qui sera appelée par mon code. Vous allez trouver mon code et les fichiers d'entrée sur Ametice.

Les sommets du graphe sont un type Vertex déclaré en template, qui sont les clés d'un `std::unordered_map<Vertex, std::unordered_set<Vertex>>`
les valeurs sont des ensembles qui correspondent à l'ensemble de voisins. Les arêtes ne sont pas orientées, alors, si 1 est voisin de 2, on a aussi que 2 est voisin de un : 1 est dans l'ensemble de voisins de 2 et 2 est dans l'ensemble de voisins de 1.

Exercice 1

Coder la classe Graph de façon à compiler et exécuter correctement le fichier `exercice1.cpp`. Ce fichier utilise les opérations :

— Constructeur d'un graphe vide.
— `void Graph<Vertex>::addVertex(Vertex v)` Ajoute le sommet `v`. Si le sommet existe déjà, ça ne fait rien. Temps : $O(1)$.
— `void Graph<Vertex>::addEdge(Vertex u, Vertex v)` Ajoute une arête entre le sommet `u` et le sommet `v`. Si l'arête existe déjà, ça ne fait rien. Si un des sommets n'existe pas encore, il sera crée automatiquement. Si `u=v`, le comportement est indéfini. Temps : $O(1)$.
— `bool Graph<Vertex>::containsVertex(Vertex v) const` Renvoie `true` si et seulement si le sommet `v` est dans le graphe. Temps : $O(1)$.
— `bool Graph<Vertex>::containsEdge(Vertex u, Vertex v) const` Renvoie `true` si et seulement si l'arête `uv` est dans le graphe. Si `u==v`, le comportement est indéfini. Temps : $O(1)$.

Exercice 2

Coder la classe Graph de façon à compiler et exécuter correctement le fichier `exercice2.cpp`. Ce fichier utilise les opérations de l'exercice 1 en plus de :
— `int Graph<Vertex>::degree(Vertex v) const` Renvoie le dégrée (nombre de voisins) du sommet `v`. Si le sommet n'est pas dans le graphe, renvoie -1. Temps : $O(1)$.
— `int Graph<Vertex>::maxDegree() const` Renvoie le dégrée maximal du graphe (le dégrée du sommet avec le plus grand dégrée). Si le graphe est vide, renvoie -1. Temps : $O(n)$ pour n sommets.
— `int Graph<Vertex>::countVertices() const` Renvoie le nombre de sommets du graphe. Temps : $O(1)$.
— `int Graph<Vertex>::countEdges() const` Renvoie le nombre d'arêtes du graphe. Temps : $O(n)$ pour n sommets.

Exercice 3

Coder la classe Graph de façon à compiler et exécuter correctement le fichier `exercice3.cpp`. Ce fichier utilise les opérations :
— `void Graph<Vertex>::removeEdge(Vertex u, Vertex v)` Supprime l'arête entre le sommet `u` et le sommet `v`. Si l'arête n'existe pas, ça ne fait rien. Si `u=v`, le comportement est indéfini. Temps : $O(1)$.
— `void Graph<Vertex>::removeVertex(Vertex v)` Supprime le sommet `v`. Si le sommet n'existe pas, ça ne fait rien. Si `v` a des arêtes, toutes ses arêtes sont aussi supprimées. Temps : $O(\delta(v))$ pour un sommet v de degree $\delta(v)$.
— `void Graph<Vertex>::clear()` Supprime tous les sommets du graphe.

Exercice 4

Coder la classe Graph de façon à compiler et exécuter correctement le fichier `exercice4.cpp`. Ce fichier utilise les opérations :

— `unordered_set<Vertex> Graph<Vertex>::vertices() const` Renvoie l'ensemble de sommets du graphe. Temps : $O(n)$ pour n sommets.
— `unordered_set<pair<Vertex,Vertex>> Graph<Vertex>::edges() const` Renvoie l'ensemble d'arêtes du graphe. chaque arête est représentée comme un `std::pair<Vertex,Ve` où le premier sommet est inférieur au deuxième. Notez que le `hash` de `std::pair` n'est pas défini automatiquement. Temps : $O(m)$ pour m arêtes.
— `const unordered_set<Vertex> & Graph<Vertex>::neighbors(Vertex v) const` Renvoie l'ensemble des voisins de `v` (sans `v`). Temps : $O(1)$.
— `unordered_set<Vertex> Graph<Vertex>::closedNeighbors(Vertex v) const` Renvoie l'ensemble des voisins de `v`, y comprit le sommet `v`. Temps : $O(\delta(v))$ pour un sommet v de degree $\delta(v)$.

Exercice 5

Coder la classe Graph de façon à compiler et exécuter correctement le fichier `exercice5.cpp`. Ce fichier utilise :

— `vector<Vertex> Graph::bfs(Vertex v, int maxv = 0) const` Renvoie un tableau avec la liste des sommets visités selon l'ordre d'un parcours en largeur à partir du sommet `v`. Si le paramètre `maxv` n'est pas 0, on renvoie seulement les premiers `maxv` éléments du tableau, sans calculer les autres. Notez que le parcours en largeur commence par le sommet `v`, suit des voisins de `v`, suit des voisins des voisins de `v`... sans jamais répéter un sommet.
— `optional<int> Graph::distance(Vertex u, Vertex v) const` Trouve la distance (nombre d'arêtes du plus court chemin) entre deux sommets `u` et `v`. Pour trouver la distance, ça suffit de modifier le code de bfs pour garder la distance de chaque sommet visité (pas besoin de Dijkstra, parce que les arêtes n'ont pas de poids). S'il n'y a pas de chemin entre `u` et `v`, ou un des sommets n'existe pas, il n'y a pas de distance. Pour cette raison, on utilise `std::optional`.

Exercice 6

Coder la classe Graph de façon à compiler et exécuter correctement le fichier `exercice6.cpp`. Ce fichier utilise les opérations :

— `Graph<Vertex>::iterator Graph<Vertex>::begin()` Renvoie un itérateur pour le premier sommet du graphe. On a choisi un `i` minuscule pour être plus compatible avec `std`. Temps : $O(1)$.
— `Graph<Vertex>::iterator Graph::end()` Renvoie un itérateur après le dernier sommet du graphe. Temps : $O(1)$.
— `Vertex Graph<Vertex>::iterator::operator*()` Renvoie le sommet du itérateur. Temps : $O(1)$.
— `void Graph<Vertex>::iterator::operator++()` Incrémente l'itérateur. Temps : $O(1)$.
— `bool Graph<Vertex>::iterator::operator==(Graph<Vertex>::iterator other)` Compare deux itérateurs. Temps : $O(1)$.

— `bool Graph<Vertex>::iterator::operator!=(Graph<Vertex>::iterator other)`
Compare deux itérateurs. Temps : $O(1)$.