Université Claude Bernard - Lyon 1 — Licence Sciences et Technologies - L3
Algorithmique, Programmation et Complexité - LIFAP6 — 2021 - 2022
http : //liris.cnrs.fr/ ∼ rchaine/COURS/LIFAP6/

TP 9 et 10 - Graphe d'un terrain
Recherche de chemins entre deux points

Dans le cadre de l'UE, vous êtes amenés à programmer en **C++** en utilisant les références pour réaliser vos passages de paramètres, les constructeurs, les destructeurs et les surcharges de l'opérateur d'affectation. Néanmoins, il sera important que vous sachiez toujours basculer dans un autre langage de programmation. Pour cela, il est important que vous établissiez votre raisonnement au niveau du pseudo-langage algorithmique en utilisant les spécificités du langage utilisé seulement quand vous effectuez la mise en œuvre.

1 Graphe structuré en grille rectangulaire

On considère un graphe non orienté dont les sommets sont plongés dans une grille rectangulaire et sont dotés d'une altitude. Le sommet de coordonnées (i, j), positionné sur la $i$ème ligne et la $j$ième colonne de la grille, est muni d'une altitude $h(i, j)$. Les seuls voisins possibles du sommet (i, j) sont les sommets à sa gauche $(i, j-1)$, à sa droite $(i, j+1)$, au dessus $(i-1, j)$ et en dessous de lui $(i+1, j)$ (on les désignera sous l'appellation voisins Ouest, Est, Nord et Sud), mais cela n'a pas besoin d'être stocké de manière explicite. Cela signifie qu'on ne va pas utiliser une représentation par matrice ou par liste d'adjacence pour stocker le graphe. Les sommets du bord de la grille ont bien entendu moins de voisins. La valuation d'une arête correspond à la distance Euclidienne entre ses deux extrémités 3D (ainsi la valuation de l'arête joignant le sommet (i, j) à son voisin Nord $(i-1, j)$ est $sqrt(1 + sqr(h(i, j) - h(i-1, j)))$. On désigne par L et C le nombre de lignes et de colonnes de la grille.

On se propose de mettre en œuvre un tel module graphe représentant une Grille, en utilisant uniquement un tableau 1D de taille $L \times C$ contenant des informations de hauteur. Le sommet (i, j) $(0 \leq i \leq L-1,\ 0 \leq j \leq C-1)$ est représenté par l'indice global $i * C + j$ de la case du tableau qui contient la valeur de son altitude $h(i, j)$. Il est inutile de créer une structure de donnée arête, puisqu'on connaît la position et l'altitude des voisins d'un sommet (i, j).

Parmi les procédures d'initialisation d'une Grille, en prévoir une pour charger un graphe sauvegardé dans un fichier.

Format de stockage d'un graphe dans un fichier :

```
15 24    // Dimensions du graphe L (largeur) et H (hauteur)
12 34 5 .... //Altitudes des L*H sommets listés ligne par ligne
             // (du haut vers le bas et de la gauche vers la droite)
```

En plus des opérations classiques d'initialisation, d'affectation et de testament de graphe, prévoir des fonctions d'accès à l'indice global d'un sommet en fonction de ses indices de ligne ou de colonne, d'accès à l'altitude d'un sommet (en fonction de son indice global), d'accès à l'indice global du voisin Nord (resp. Sud, Est et Ouest) d'un sommet (avec pour précondition que le voisin existe), d'accès à la distance d'un sommet à son voisin Nord (resp. Sud, Est et Ouest), s'il existe, ainsi que des procédures de modification de l'altitude d'un sommet et une procédure d'affichage de la grille de hauteur.

2 Chercher son chemin : l'algorithme A*

On considère à présent que la grille modélise un site géographique et on considère les coordonnées d'un sommet de départ (LYON) sur ce site. L'algorithme de Dijkstra vu en cours permet

de trouver les plus courts chemins entre LYON et tous les autres sommets du graphe. Si jamais vous désirez vous rendre de LYON à PARIS, vous n'aurez que faire d'un algorithme si méthodique qui vous donnera non seulement le chemin optimal, mais aussi les plus courts chemins permettant d'aller dans toutes les autres villes de France.

Idée générale de A*

L'algorithme de recherche A* est une variante de Dijkstra qui se concentre sur la recherche rapide d'un chemin *raisonnablement* court entre le sommet initial LYON et un sommet cible PARIS fourni par l'utilisateur. Le déroulement de cet algorithme vise à ce que la première solution trouvée soit l'une des meilleures. C'est un algorithme qu'il convient d'utiliser dans des applications privilégiant les temps de calcul plutôt que l'exactitude des résultats.

Comme Dijkstra, l'algorithme A* procède par exploration progressive des sommets autour du sommet de départ LYON. L'algorithme A* maintient, pour chaque sommet n visité, une **estimation** de la longueur du meilleur chemin entre LYON et Paris **en passant par** n, en **sommant** la longueur d'un chemin connu menant de LYON à n avec une estimation de la longueur restant à parcourir entre n et PARIS.

Comme Dijkstra, l'algorithme A* procède par étapes successives au cours de chacune desquelles il fige la situation pour un des sommets visités et procéde au relâchement des arcs/arêtes issues de ce sommet : cela permet de découvrir de nouveaux sommets ou de diminuer la distance connue de LYON à certains sommets. Dans Dijkstra, c'est le sommet visité non encore figé le plus proche de LYON qui est choisi, alors que dans A* il s'agit cette fois-ci du sommet visité mais non encore figé qui semble le plus apte à rapprocher la source de la destination finale qui est choisi, au sens de l'estimation de la longueur totale du chemin.

Pour chaque sommet n visité, on stocke donc la **distance connue** le séparant de LYON (distance du chemin que l'on peut retrouver si on stocke l'information de prédécesseur) et on peut également stocker, si on ne souhaite pas en refaire le calcul, l'**approximation de la distance restant à parcourir** pour atteindre PARIS.

Plusieurs approximations sont possibles pour la distance restant à parcourir entre le sommet n de coordonnées (i, j) et PARIS de coordonnées (i_P, j_P) :

- Le voyageur pressé qui n'aime pas les variations de dénivelé prend en compte la distance à vol d'oiseau, c'est à dire la distance Euclidienne entre les positions 3D de n $(i, j, h(i, j))$ et PARIS $(i_P, j_P, h(i_P, j_P))$.
- Le voyageur un peu plus aventurier est prêt à traverser des montagnes et il approxime la distance restant à parcourir par la distance Euclidienne entre les positions 2D sur la grille de n (i, j) et PARIS (i_P, j_P).

Implantation

Au niveau de l'implantation, on sépare les sommets en 3 ensembles, comme dans Dijkstra :

- ceux qui n'ont pas encore été atteints par un chemin (ensemble Blanc),
- ceux qui ont été atteints, mais pour lesquels on n'a pas encore figé le chemin depuis LYON (ensemble Gris),
- et ceux pour lesquels on ne va plus faire évoluer le chemin les séparant de LYON (ensemble Noir).

Au démarrage de l'algorithme, tous les sommets sont blancs, sauf le sommet de départ qui est colorié en gris.

L'algorithme A* est un algorithme glouton qui considère à chaque étape le sommet gris s offrant le meilleur chemin entre LYON et PARIS (en sommant la distance connue entre LYON et s et l'approximation de la distance restante jusqu'à PARIS). Cette sélection pourra être effectuée en rangeant les sommets gris dans une file de priorité (possibilité de faire votre propre module ou

d'utiliser la `priority_queue` de la STL) ou dans un tableau (mais l'efficacité de la recherche du meilleur sommet gris s est alors moindre).

Les arêtes issues de s dont l'autre extrémité est blanche ou grise sont relâchées de manière à mettre à jour les chemins vers ces exrémités et à colorier en gris celles qui sont blanches. Le sommet s est ensuite colorié en noir.

L'algorithme s'interrompt lorsque PARIS est coloriée en noir ou qu'il n'y a plus de sommet gris (dans ce cas, il n'y a pas de chemin menant de LYON à PARIS, mais ce n'est pas possible sur notre graphe structuré en grille).

Il est possible de stocker les infos relatives au parcours du graphe (couleur, prédécesseur, longueurs de chemin) dans des tableaux 1D de taille $L \times C$, indexés dans le même ordre que les $L \times C$ sommets du graphe.

3 Organisation de votre code

Vous accorderez un soin particulier à l'écriture de vos modules et de l'interface (fonctions membres publiques) de votre classe de graphe.

4 Programme de test

— Soyez attentifs à ce que votre programme de test puisse prendre un fichier en entrée, de manière à ce qu'il soit facile pour l'utilisateur de modifier la valeur de hauteur de certains sommets, ainsi que les coordonnées de LYON et PARIS. Le programme écrira sur la sortie standard le chemin qu'il propose entre LYON et PARIS.

— Permettez à l'utilisateur de choisir entre différentes stratégies d'estimation de la distance séparant un sommet n de PARIS. Dans l'implantation, vous pourrez pour cela recourir à des pointeurs de fonction et vous êtes libres si vous en avez la fantaisie de proposer des stratégies différentes des deux seules qui sont proposées dans le sujet. On peut par exemple imaginer la prise en compte de l'orientation du vent, la difficulté à se déplacer dans une direction donnée, etc...

— Permettez un mode bavard d'exécution de votre application, avec une visualisation des couleurs du graphe à chaque étape de l'algorithme.

5 Conditions de rendu

Vous déposerez une archive `Nom1_Nom2.tgz` (`Nom1` et `Nom2` étant les noms des 2 étudiants composant le binôme) de votre travail sur Tomuss avant le **mardi 14 décembre** à 23h (le dépôt sera ensuite fermé). **Les deux membres du binôme DOIVENT APPARTENIR AU MEME GROUPE DE TP et il est interdit de se greffer au travail d'un étudiant dans un autre groupe**. Il est possible qu'un groupe de TP contienne un monôme.

Votre archive doit contenir un répertoire `Nom1_Nom2` avec les sources de votre application. Les sources sont bien évidemment tous les fichiers `.cpp` et `.hpp`, mais aussi le fichier `Makefile`, ainsi que tout autre fichier (`README`, etc.) utile à la compréhension de votre programme, ainsi qu'à sa compilation et à son exécution.

Le fichier `Makefile` doit modéliser clairement les dépendances entre les fichiers et permettre une recompilation partielle, en cas de modification partielle.

Attention !

— Votre archive NE DOIT PAS contenir de fichiers objets (.o) ni d'exécutable.

— Le travail rendu doit être issu de la collaboration entre 2 personnes et toute récupération flagrante du code d'autrui sera sanctionnée.

— Votre programme doit pouvoir être compilé sans erreur ni avertissement avec `g++` sans nécessiter l'installation de bibliothèques supplémentaires sur une des machines de la salle de TP.
— une attention particulière devra être consacrée à la mise en œuvre d'une programmation modulaire débouchant sur la proposition de véritables types abstraits.
— l'interface de vos modules doit offrir toutes les informations utiles à l'utilisation des types et des fonctionnalités offertes.
— soignez bien votre programme principal (`main.cpp`) qui doit illustrer que les opérations à coder ont été faites correctement.

Le **mercredi 15 décembre**, vous serez amenés à faire une petite présentation au chargé ou à la chargée de TP au cours de laquelle des questions seront posées à chacun des 2 membres du binôme. Cette présentation sera précédée d'une mini-interrogation sur papier portant sur le travail que vous aurez rendu la veille.

Profitez du mercredi 8 décembre après-midi pour avancer sur votre TP avec votre binôme. Il n'y aura alors pas de séance de TP encadrée pour que vous puissiez avancer sur votre travail.