Autour de Dijkstra

20 novembre

Considérons un graphe $G = (V, E)$ orienté connexe. On peut pondérer les arêtes du graphe avec une fonction $p : E \to \mathbb{R}$, dite de pondération.

1 Construction d'un graphe (à faire AVANT de venir en TP)

On considère maintenant un graphe orienté G connexe et pondéré à poids positifs.
Nous allons travailler avec le type de graphe suivant :

```
struct sommetpond{int val; int poids;};
struct arete{int u; int v; int p;};

typedef struct arete arete;
typedef struct sommetpond sommetpond;


struct edgenode{
    sommetpond y;
    struct edgenode* next;
};

typedef struct edgenode edgenode;

struct graph {
    edgenode* edges[MAXV];
    int degree[MAXV];
    int nedges;
    int nbvertices;
    bool discovered[MAXV];
};

typedef struct graph graph;
```

Cette partie va permettre de construire efficacement des graphes à partir de fichiers textes. Elle doit être préparée en amont. Il conviendra de bien tester que vos fonctions fonctionnent.

▷ **Question 1.** Ecrire une fonction `void initialize_graph(graph* g)` qui initialise le graphe `g` passé par pointeur comme étant le graphe vide non orienté. ◁

▷ **Question 2.** Ecrire une fonction `void insert_edge(graph* g, int x, int y,int p)` qui insère une arête de $\{x, y, p\}$ orientée. ◁

Pour travailler sur des graphes, on va écrire une fonction permettant de lire un fichier contenant le graphe sous le format suivant :

Une première ligne contenant deux entiers, le nombre de sommets `n` et le nombre d'arêtes `k`. Ensuite `k` lignes contenant trois entiers `i`, `j` et `p` indiquant qu'il y a une arête de `i` vers `j` de poids `p`.

▷ **Question 3.** Ecrire une fonction `void read_graph(FILE* f, graph* g)` qui lit un graphe sur le flux `f` et le place dans `g` après l'avoir initialisé. ◁

▷ **Question 4.** Ecrire une fonction `void free_edges(graph* g)` qui libère les listes d'adjacence. ◁

▷ **Question 5.** Ecrire une fonction `void initialize_search(graph* g)` qui initialise les tableaux utilisés pour les fonctions de parcours. ◁

▷ **Question 6.** Ecrire une fonction `void pp_affiche(graph* g)` qui effectue un parcours en profondeur du graphe et affiche les sommets du graphe dans l'ordre du parcours. ◁

A ce stade, vous devez être en mesure de bien tester toutes vos fonctions : dessiner un graphe, le représenter dans un fichier, le construire en lisant ce fichier et faire un parcours en profondeur pour vérifier que l'affichage correspond.

2 Dijkstra : une première version (en C)

Nous avons vu l'algorithme de Dijkstra qui permet de calculer les distances à un sommet dans un graphe pondéré dont les poids sont positifs ou nuls. Il existe plusieurs manières de formuler, de prouver et d'implémenter cet algorithme.

La version la plus simple à écrire consiste à adapter le parcours en largeur.

▷ **Question 7.** Ecrire une fonction `void parcours_largeur(graph* g, int s)` qui parcourt `g` en largeur à partir du sommet `s` et affiche ses sommets. On pourra utiliser le fichier `file.h` fourni. ◁

Pour adapter ce parcours, il suffit de remplacer la file par une file de priorité en utilisant comme valeur de priorité pour un sommet `x` la distance actuellement estimée par l'algorithme entre `x` et l'origine du parcours `s`.

▷ **Question 8.** En utilisant le fichier `file_prio.h`, écrire une fonction `int* dijkstra(graph* g, int s)` qui renvoie un tableau indexé par les sommets de `g` et contenant la distance à `s` pour chacun d'entre eux. ◁

3 Récupération de données réelles

Système de coordonnées. Dans ce sujet, on fera l'approximation que la terre est une sphère et que tout point de la sphère peut-être déterminé à l'aide de deux angles appelés latitude et longitude. Par convention les latitude et longitude sont toujours données en degré d'angle, la latitude est l'angle selon l'axe nord-sud, toujours dans l'intervalle $[-90; +90]$, -90 correspond au pôle sud, $+90$ au pôle nord. La longitude est un angle selon l'axe Est-Ouest, entre $[-180; +180]$.

Distance entre points sur une sphère. Étant donné deux points p_1 et p_2 de latitude y_1 et y_2 et de longitude x_1 et x_2, on définit la distance entre p_1 et p_2 avec la formule de haversine $d(p_1, p_2)$ définie de la façon suivante :

$$d(p_1, p_2) = 2 \times R_{\text{terre}} \times \arcsin\left(\sqrt{\sin\left(\frac{y_1 - y_2}{2}\right)^2 + \cos(y_1)\cos(y_2)\sin\left(\frac{x_1 - x_2}{2}\right)^2}\right)$$

où R_{terre} est le rayon terrestre, estimé à 6371 km et les angles sont en radians. Le résultat obtenu est en km.

Carte fournie. Dans cet exercice nous avons à disposition deux fichiers `routes.txt` et `positions.txt` décrivant une partie des routes de l'Ile-de-France. Le premier fichier décrit des segments de routes qui existent

sur notre carte tandis que le second fichier donne les coordonnées (latitude et longitude) des points utilisés pour décrire les segments de routes. Plus précisément :

- le fichier `routes.txt` commence par un entier, 2 111 499, qui décrit le fait qu'il y a 2 111 499 segments de route dans notre carte. Les 2 111 499 lignes suivantes décrivent chacune un segment de route. Chaque segment de route est donné comme un quadruplet (f, t, d, n) où f est un entier qui décrit le point de départ, t est un entier qui décrit le point d'arrivée, d est un nombre flottant et donne la distance en mètres entre les points f et t tandis que n décrit le nom de la route. Ces quatre éléments seront séparés par des espaces. Voici, par exemple, les 5 premières lignes de ce fichier :

  ```
  2111499
  0 176674 864.984 Autoroute du Nord
  1 11731 522.499 Autoroute du Nord
  2 81556 1618.6 <ROUTE SANS NOM>
  3 7624 938.6759999999998 <ROUTE SANS NOM>
  ```

  Noter qu'une même route (comme l'autoroute du Nord) sera généralement décrite par plusieurs segments, chaque segment représentant une portion assez courte de la route.

- le fichier `positions.txt` commence par un entier, 842 089, décrivant le fait qu'il y a 842 089 points décrits. Les 842 089 lignes suivantes décrivent chacune un point, plus précisément, la i-ème ligne du fichier décrit le point $i-2$ en donnant d'abord sa longitude puis une espace en ensuite sa latitude. Voici, par exemple, les 5 premières lignes de ce fichier :

  ```
  1431808
  2.5511375 49.0834393
  2.5421644 49.0391224
  2.4635493 48.8840815
  2.4692871 48.8990232
  ```

  On a donc que le point 0 a pour longitude 2.5511375, que le point 2 a pour latitude 48.8840815, etc.

Attention ! La distance donnée dans le fichier `routes.txt` ne correspond pas toujours à la distance donnée par la formule d'Haversine car les routes ne sont pas forcément droites.

Squelette du code. Pour la suite de l'exercice, on vous fournit un squelette de code qu'il faudra ensuite éditer. Ce squelette contient trois fichiers :

- `main.c` qui sera le point d'entrée de notre programme;
- `lecture.c` qui sera un fichier édité plus tard dans le sujet pour lire la carte de l'Île-de-France ;
- `lecture.h` qui est le fichier header associé à `lecture.c`.

Pour compiler le programme, il faut exécuter la commande `gcc lecture.c main.c -lm`.

▷ **Question 9.** Éditer la fonction `lit_positions` du fichier `lecture.c` pour qu'elle lise et stocke les informations du fichier `positions.txt` afin que `position[i]` contienne les coordonnées du point i. Éditer ensuite la fonction `nettoie` pour qu'elle libère la mémoire allouée dans `lit_positions`. Vous pouvez vous inspirer de la fonction `lit_routes`. ◁

▷ **Question 10.** Créer les structures de données nécessaires au stockage des informations du fichier `routes.txt`. Attention, pour la compilation séparée, il faudra définir normalement les tableaux et variables dans `lecture.c` mais pour pouvoir y accéder depuis `main.c`, il faudra les redéfinir précédés du mot-clef `extern` dans `lecture.h`. Il est possible de s'inspirer de ce qui a été fait pour le tableau `position` de type `geopoint` ou de la variable `nb_routes`. ◁

▷ **Question 11.** Éditer la fonction `lit_routes` du fichier `lecture.c` pour qu'elle stocke les informations du fichier `routes.txt` dans les structures que vous venez de créer. ◁

▷ **Question 12.** Éditer le fichier `main.c` et compléter la fonction `ll_distance` pour qu'elle calcule la distance entre deux points selon la formule d'Haversine. Selon votre fonction, quelle est la longueur moyenne d'un segment de route parmi les segments fournis ? Quelle est la longueur maximale d'un des segments ? Vérifier les réponses obtenues en comparant avec les longueurs des segments. ◁

Représentation en liste d'adjacence. On veut maintenant créer un fichier contenant le graphe décrit ci-haut représenté en liste d'adjacence. Ce fichier aura sur sa première ligne le nombre de noeuds (ici 842 089) et ensuite il y aura 842 089 lignes. La $i+2$-ème ligne du fichier décrit les voisins du noeud i de la façon suivante d'abord on a un entier k, le nombre de voisins du noeud i, puis un espace suivi des $2k$ nombres séparés par des espaces $n_1 d_1 \dots n_k d_k$, décrivant les k voisins $n_1 \dots n_k$ respectivement à distance $d_1 \dots d_k$.
On remarque que le graphe, bien qu'orienté, est symétrique car les routes sont supposées à double sens.

▷ **Question 13.** Écrire une fonction `sauvegarde_graphe` qui stocke dans le fichier `graphe.txt` la représentation en liste d'adjacence du graphe routier associé aux fichiers `positions.txt` et `routes.txt`. On pourra utiliser la structure de graphe proposée dans la première partie en prenant soin de choisir une valeur convenable pour `MAXV`. Tester la fonction, combien de temps met-elle à s'exécuter ? ◁

4 Recherche de plus courts chemins (en OCAML)

Pour cet exercice, on vous fournit 4 fichiers :

- `graphe.txt` qui contient un graphe en représentation de liste d'adjacence, dans un format similaire à celui obtenu à la fin de la partie précédente. Ce graphe est non orienté (ou orienté symétrique) et sans arêtes multiples ;
- `oriente.ml` qui est le fichier principal à éditer dans cet exercice. Il contient déjà les fonctions `lit_graphe` et `lit_position` lisant les fichiers `graphe.txt` et `positions.txt`
- enfin les deux derniers fichiers sont `PQ.ml` et `PQ.mli` et correspondent à un module appelé `PQ` qui sert de file de priorité. L'interface de ce module, donnée dans `PQ.ml`, décrit comment utiliser ce module.

Pour pouvoir utiliser le module `PQ` il faut d'abord le compiler. Pour cela on va compiler l'interface du module avec la commande `ocamlc PQ.mli` qui va créer un fichier d'interface de module compilé `PQ.cmi`. Ensuite on va compiler le module lui-même, avec la commande `ocamlc PQ.ml` qui va créer un fichier `PQ.cmo`.
Ensuite pour exécuter `oriente.ml` on peut le compiler mais en précisant d'utiliser le module `PQ` dans la ligne de commande. Pour le compiler on peut faire `ocamlc PQ.cmo oriente.ml`.

▷ **Question 14.** Le graphe lu par la fonction `lit_graphe` contient-il une ou plusieurs composantes connexes ? ◁

▷ **Question 15.** Écrire une fonction qui calcule le plus court chemin entre deux points en utilisant l'algorithme de Dijkstra. La distance d'un chemin dans le graphe est la somme des distances des arêtes composants le chemin (la distance d'une arête est fournie). Il est recommandé d'utiliser le module `PQ` pour cette question.
Quelle est la distance du plus court chemin entre le point 819 913 (qui est proche de Télécom Paris) et le point 282 392 (qui est proche de la station Le guichet du RER B) ? ◁

▷ **Question 16.** On suppose maintenant que, comme dans le premier exercice, le fichier `positions.txt` contient les positions des points du graphe et donc que la distance dans le graphe entre deux points de positions p_1 et p_2 est nécessairement plus grande que la distance donnée par la formule de Haversine. Expliquer comment on peut utiliser cette information à l'aide de l'algorithme A*. Cet algorithme A* a-t-il une meilleure complexité que l'algorithme Dijkstra ? A-t-il des chances d'être plus rapide en pratique sur un graphe comme celui fourni (graphe correspondant aux routes de l'Ile-de-France). Expliquer.
Écrire une fonction calculant le plus court chemin avec l'algorithme A*. Comparer le nombre de noeuds explorés dans l'algorithme A* et dans l'algorithme Dijkstra. ◁