TP15.5: Graphes, trajet de métro

MP2I Lycée Pierre de Fermat

Graphes pondérés

Les premières questions sont identiques à celles de la fin du TP15, vous pouvez reprendre votre code et les sauter si vous y avez déjà répondu.

On considère maintenant des graphes pondérés.Pour un graphe $G = (S, A, w)$ pondéré, on rappelle que les listes d'adjacence stockent dans ce cas des une liste de couples (voisin, poids de l'arête) pour chaque sommet $u \in S$, et que les matrices d'adjacences stockent les coefficients $w(u, v)$ pour $u, v \in S$. On considèrera des poids réels, pas forcément entiers ni positifs.

Question 1. Créer un nouveau fichier C, y recopier les types des listes d'adjacence du TP précédent, et faire les modifications nécessaires pour pouvoir gérer des graphes pondérés. On appelera le type `LADJ` et plus `GRAPHE`.

Question 2. Proposer un type pour les matrices d'adjacence, que l'on appelera `MADJ`.

Question 3. Écrire deux fonctions permettant de transformer une matrice d'adjacence en liste d'adjacence et inversement.

L'algorithme de Floyd-Warshall permet de calculer en $\mathcal{O}(n^3)$ la matrice des distances d'un graphe $G = (S, A, w)$, c'est à dire la donnée de la distance entre toute paire de sommet, pour un graphe ne contenant pas de cycle négatif. Faire une itération de l'algorithme de plus que nécessaire permet de détecter l'existence de cycles négatifs.
On rappelle le principe: pour un graphe $G = (S, A, w)$ avec $S = \{0, \dots, n-1\}$, on calcule pour $i, j \in [\![0, n-1]\!]$ et $k \in [\![0, n]\!]$ la quantité $M_{ij}^k =$ le poids d'un plus court chemin entre i et j ne passant que par des sommets parmis 0 à $k-1$. On a la formule de récurrence suivante pour $k \geq 0$:

$$M_{ij}^{k+1} = \min(M_{ij}^k, M_{ik}^k + M_{kj}^k)$$

Cette formule exprime qu'un plus court chemin entre i et j peut soit passer par k, soit pas.

Question 4. Que vaut M_{ij}^0 pour $i, j \in [\![0, n-1]\!]$?

Question 5. Écrire une fonction `float ** floyd_warshall(MADJ* g)` qui applique l'algorithme de Floyd-Warshall et renvoie la matrice des distances calculée.

Fin du TP précédent

On se propose maintenant d'implémenter une version simple l'algorithme de Dijkstra, sans structure de file de priorité particulière. On en rappelle le pseudo-code:

Algorithme 1 : PCC: Dijkstra

Entrée(s) : $G = (S, A, w)$ graphe pondéré avec $s = \{0, \dots, n-1\}$, $s \in S$
Sortie(s) : **d** tableau des distances depuis s, et **Pred** tableau des prédecesseurs

```
1  d ← tableau de taille n initialisé à [+∞, ..., +∞];
2  Pred ← tableau de taille n initialisé à [−1, ..., −1];
3  d[s] = 0;
4  Q ← ensemble contenant chaque élément de S;
5  tant que Q non vide faire
6      u ← extraire sommet de Q avec d[u] minimal;
7      pour v voisin de u faire
8          si d[u] + w(u, v) < d[v] alors
9              d[v] ← d[u] + w(u, v);
10             Pred[v] ← u;
11 retourner d, Pred
```

On utilisera des graphes par listes d'adjacence. De plus, on utilisera la structure suivante pour stocker le couple (**d**, **Pred**):

```
1 typedef struct {
2   int* d;
3   int* pred;
4 } RES_DIJK
```

Dans la librairie `<math.h>`, vous trouverez une valeur nommée `INFINITY`, qui est un flottant particulier, se comportant comme $+\infty$. Pour compiler un programme utilisant `<math.h>`, il faut rajouter l'argument `-lm` **à la fin de la commande de compilation**. Par exemple:

```
gcc  toto.c bla.c -o bla -g -lm
```

Question 6. Écrire une fonction `RES_DIJK dijkstra(LADJ* g, int s)` qui lance l'algorithme de Dijkstra depuis le sommet s et renvoie les tableaux des distances et des prédecesseurs.

Question 7. En déduire une fonction `int* pcc(LADJ* g, int s, int t)` qui renvoie un plus court chemin de s à t sous la forme d'un tableau. s et t seront respectivement le premier élément et le dernier élément utile du tableau, ce qui permettra de le manipuler sans connaître sa taille.

Question 8. Tester extensivement toutes vos fonctions si ce n'est pas déjà fait.

Avancé: Dictionnaire d'adjacence

On se propose d'utiliser l'algorithme de Dijkstra pour calculer des plus courts trajets sur la carte des transports en commun parisiens.[1] Afin de de rendre le code plus lisible, nous allons implémenter les graphes par **dictionnaire d'adjacence**. Cette implémentation est identique à celle des listes d'adjacences, mais utilise un dictionnaire géré par table de hachage afin de pouvoir manipuler des sommets qui sont des chaînes de caractères.

L'archive du TP contient quatre paires de fichiers .h/.c:

[1] Le métro de Toulouse est plus local mais moins intéressant d'un point de vue topologie de graphe...

- `stof.h` et `stof.c`: implémentent des dictionnaires **S**tring **TO** **F**loat, autrement dit dont les clés sont des chaînes de caractères `char*` et dont les valeurs sont des flottants `float`.
- `stos.h` et `stos.c`: implémentent des dictionnaires **S**tring **TO** **S**tring, autrement dit dont les clés sont des chaînes de caractères `char*` et dont les valeurs sont des `char*` aussi.
- `adj_dict.h` et `adj_dict.c`: implémentent les graphes par dictionnaire d'adjacence.
- `hash.h` et `hash.c`: définitions utiles pour le hachage.

La documentation de ces fonctions est recopiée en annexe à la fin de ce sujet.

L'archive contient également un fichier `demo.c` montrant comment utiliser les différentes fonctions de ces interfaces. Ce fichier contient notamment une fonction `void print_graphe(GRAPHE* g)` qui permet d'afficher les informations d'un graphe, en utilisant l'interface de `adj_dict.h`. Pour compiler ce fichier, tapez `gcc stos.c stof.c hash.c demo.c -o demo`.

Si vous êtes très rapide, avant de continuer vous pouvez essayer de recoder tous les .c uniquement à partir de la documentation donnée dans les .h.

Question 9. Lire attentivement le fichier `adj_dict.h`, ainsi que la partie du code dans `demo.c` qui utilise les fonctions sur les dictionnaires d'adjacence, afin de vous familiariser avec la structure de cette implémentation. Pour l'instant, ne regardez pas `adj_dict.c`: vous n'avez besoin que de lire les spécifications des fonctions pour les utiliser.

Question 10. Écrire une fonction `int degre(GRAPHE* g, char* u)` qui renvoie le degré de u dans g.

Afin de stocker de tels graphes, on propose le format suivant: pour représenter un graphe $G = (S, A, w)$ pondéré à n sommets et m arêtes, on écrira dans un fichier:

- Sur la première ligne, n et m;
- Sur les n lignes suivantes, les noms des différents sommets, qui peuvent comporter des espaces, mais pas de signe $;
- Sur les m lignes suivantes, des triplets de la forme `u$v$d` où (u, v) est une arête et $d = w(u, v)$. Il n'y a pas d'espace autour des dollars.

Vous trouverez dans l'archive du TP un fichier "bebe_graphe.txt" représentant un petit graphe sous ce format.

Rappel: la fonction `int fgetc(FILE* f)` renvoie un caractère lu dans f, ou bien renvoie `EOF` s'il n'y a plus rien à lire.

Question 11. Écrire une fonction `bool lire_ajouter_arete (GRAPHE* g, FILE* f)` qui lit une ligne dans f et ajoute l'arête correspondante au graphe g. Cette fonction s'occupera aussi de lire le '
`n`' de la fin de ligne. On pourra supposer qu'aucun nom de sommet ne fait plus de 50 caractères.

Question 12. Écrire une fonction `GRAPHE* lire_graphe(char* filename)` permettant de lire les informations d'un graphe dans un fichier donné, ainsi qu'une fonction `void ecrire_graphe(char* filename)` qui sauvegarde les informations d'un graphe sous le format spécifié plus haut.

Question 13. Écrire une fonction `float arete(GRAPHE* g, char* u, char* v)` qui renvoie le poids de l'arête (u, v) dans g, ou bien $+\infty$ si cette arête n'existe pas.

Question 14. Quelle est la complexité de cette fonction ?

Dans la suite, on s'interdit d'utiliser cette fonction car elle est trop coûteuse.

Passons maintenant à l'implémentation de l'algorithme de Dijkstra sur cette structure. Il s'agira d'adapter le code de la section précédente en utilisant la structure de dictionnaire d'adjacence proposée ici. On utilisera des dictionnaires pour stocker **d** et **Pred**, ainsi l'algorithme commencera comme suit:

Algorithme 2 : PCC: Dijkstra

Entrée(s) : $G = (S, A, w)$ graphe pondéré à n sommets, $s \in S$
Sortie(s) : **d** tableau des distances depuis s, et **Pred** tableau des prédecesseurs

1 **d** $\leftarrow$ dictionnaire avec S comme clés, et ∞ pour toutes les valeurs;
2 **Pred** $\leftarrow$ dictionnaire vide;
3 $d[s] = 0$;
4 ... ;

On propose d'utiliser dans le code C deux tableau, chacun de taille n, en plus des dictionnaires **d** et **Pred**, qui permettront d'implémenter Q:

- Un tableau `char** sommets` stockant la liste des sommets du graphe;
- Un tableau de booléens `in_Q` tel que `in_Q[i]` indique si `sommets[i]` est encore dans Q.

On pourra également introduire une variable q stockant le nombre d'éléments restant dans Q.

Enfin, on introduit une fonction très utile de la librairie `<string.h>`: `char* strdup(char* s)`. Cette fonction renvoie une copie de s **et alloue toute seule de la mémoire pour stocker cette copie**. Il faut bien évidemment libérer le pointeur renvoyé lorsque l'on a fini de l'utiliser, et libérer s séparément si besoin. Par exemple:

```
1 char* u = malloc(5*sizeof(char));
2 strcpy(u, "toto"); // écrit "toto" dans la mémoire allouée pour u
3
4 free(u); // si on commente cette ligne, on crée une fuite mémoire.
5
6 u = strdup("tatitata"); // réserve un nouvel emplacement et y écrit "tatitata"
7 free(u);
```

Question 15. Lire attentivement les fichiers `stos.h` et `stof.h`, ainsi que la partie du fichier `demo.c` qui utilise les fonctions de ces deux fichiers, afin de vous familiariser avec l'utilisation des dictionnaires.

Question 16. Implémenter une fonction `STOS_TBL* dijkstra(GRAPHE* g, char* s)` appliquant l'algorithme de Dijkstra à partir de s dans g, et renvoyant le dictionnaire des prédecesseurs.

Question 17. En déduire une fonction `char** plus_court_chemin(GRAPHE* g, char* s, char* t)` qui renvoie un tableau représentant un plus court chemin entre s et t dans g. s sera le premier élément du tableau et t le deuxième.

Question 18. Écrire une fonction `total(GRAPHE* g, char** chemin, char* s, char* t)` qui calcule la distance totale parcourue en suivant le chemin donné de s à t dans g.

Trajet en métro

L'archive du TP contient un dossier "metro", contenant lui même 14 fichiers "`ligne_1.txt`", "`ligne_2.txt`", ..., "`ligne_14.txt`" contenant les informations des 14 lignes de métro parisiennes (lignes bis exclues) sous la forme de graphes. On souhaite pour commencer réunir ces informations dans un unique graphe.

Question 19. Écrire une fonction `GRAPHE* fusion_graphes(GRAPHE** L, int p)` qui prend en entrée une liste L de p graphes, et renvoie un graphe dont:

- L'ensemble des sommets est l'union de l'ensemble des sommets des graphes de L;
- L'ensemble des arêtes est l'union de l'ensemble des arêtes des graphes de L.

Ainsi, une station présente sur plusieurs lignes sera représentée par un unique sommet.

Question 20. Sur le graphe construit, chercher le plus court chemin entre `Porte d'Auteuil` et `Pyramides`. Vous pouvez vérifier la cohérence de vos résultats sur un plan du métro en ligne. Combien de changements y-at'il ?

On veut maintenant modéliser le fait qu'un changement n'est pas instantané, et que les jours fériés et les week-ends, un changement peut même être très coûteux. Pour cela, on propose un opérateur sur les graphes que l'on appellera **opérateur de correspondance**.

Étant donné p graphes $G_1, \dots G_p$ et $t \geq 0$ un temps d'attente, le graphe des correspondances de $G_1, \dots G_p$ est le graphe $G = \mathbf{Corresp}^t(G_1, \dots G_p)$ obtenu en mettant côte à côte les graphes $G_1, \dots G_p$ sans aucune arête inter-graphe, puis en connectant tous les sommets de même nom, par une arête de poids t.

Question 21. Dessiner trois graphes F, G et H ayant quelques noms de sommets en commun, puis dessiner le graphe $\mathbf{Corresp}^t(F, G, H)$.

Afin de représenter cet opérateur dans notre programme, il faut donner un nom aux nouveaux sommets créés. Pour la ligne numéro i, une station u aura dans le graphe des correspondances le nom $u\#i$. Par exemple, la station `Gare Montparnasse` est sur les lignes 4, 6, 12 et 13. Dans le graphe des correspondances elle sera alors remplacée par 4 sommets: `Gare Montparnasse_4`, ..., `Gare Montparnasse_13`.

Rappel: la fonction `sprintf` sert à écrire dans un string en utilisant les mêmes mécanismes que `printf`. Par exemple, `sprintf(s, "ligne_%d.txt", 12)` va remplacer s par `ligne_12.txt`. Il faut cependant avoir réservé assez d'espace dans s.

Question 22. Écrire une fonction `GRAPHE* correspondances(GRAPHE** L, int p, float t)` qui prend en entrée une liste $L = [G_1, \dots, G_p]$ de p graphes ainsi qu'un temps d'attente t, et crée le graphe des correspondances $G = \mathbf{Corresp}^t(G_1, \dots, G_p)$.

Nous avons maintenant tous les outils nécessaires pour écrire un programme donnant des trajets de métro efficaces. Avant de continuer, séparez votre code pour l'algorithme de Dijkstra en un .h et un .c (si vous avez un `main` pour les tests, renommez la fonction en `test`).

Question 23. Écrire un programme `trajet`, que l'on pourra exécuter comme suit:

```
./trajet depart destination t
```

Ceci aura pour effet d'afficher le trajet optimal entre la station de départ et la station destination, en supposant que le temps de correspondance au sein d'une station est donné par t. Le programme affichera également le temps total de trajet.

Question 24. Tester votre programme en vérifiant les résultats sur la carte du métro, et vérifiez qu'en augmentant le temps d'attente aux stations, votre programme privilégie les trajets ayant peu de changements.

Question 25. Améliorez votre programme pour qu'il affiche seulement les lignes à prendre et les changements à effectuer, ainsi que le temps passé sur chaque ligne, plutôt que la liste des stations. On pourra imiter le format suivant:

```
Trajet de Porte d'Auteuil à Pyramides (Temps total: 24 minutes):

1) Ligne 9 de Porte d'Auteuil jusqu'à Chaussée d'Antin La Fayette (17 minutes)

2) Changement à Chaussée d'Antin La Fayette (4 minutes)

3) ligne 7 de Chaussée d'Antin La Fayette jusqu'à Pyramides (3 minutes)
```

(Il se peut que vous ne trouviez pas les mêmes valeurs sur cet exemple.)

Annexe: Documentation

Dictionnaires string to string

```c
/* Crée une table de hachage string to string vide */
STOS_TBL* stos_tbl_vide();

/* Renvoie true si `clef` est une clef de t, false sinon */
bool stos_mem(STOS_TBL* t, char* clef);

/* Renvoie la valeur associée à `clef` dans t. Si
   t ne contient pas `clef`, affiche un message d'erreur
   et arrête l'exécution.
   La chaîne renvoyée est stockée dans une zone fraîchement
   allouée, il faut la libérer. */
char* stos_get(STOS_TBL* t, char* clef);

/* Assigne à `clef` la valeur `valeur` dans t. Si
   `clef` est déjà présente, modifie la valeur précédente */
void stos_set(STOS_TBL* t, char* clef, char* valeur);

/* Libère l'espace alloué pour la table t. */
void stos_free(STOS_TBL* t);
```

Dictionnaires string to float

```c
/* Crée une table de hachage string to string vide */
STOF_TBL* stof_tbl_vide();

/* Renvoie true si `clef` est une clef de t, false sinon */
bool stof_mem(STOF_TBL* t, char* clef);

/* Renvoie la valeur associée à `clef` dans t. Si
   t ne contient pas `clef`, affiche un message d'erreur
   et arrête l'exécution. */
float stof_get(STOF_TBL* t, char* clef);

/* Assigne à `clef` la valeur `valeur` dans t. Si
   `clef` est déjà présente, modifie la valeur précédente */
void stof_set(STOF_TBL* t, char* clef, float valeur);

/* Libère l'espace alloué pour la table t. */
void stof_free(STOF_TBL* t);
```

Graphes

```c
/* Crée un graphe vide */
GRAPHE* graphe_vide();

/* Renvoie le nombre de sommets de g */
int taille(GRAPHE* g);

/* Renvoie la liste des sommets de g. Les chaînes renvoyées
  sont allouées dans de la mémoire fraîche, il faut les libérer
  ainsi que la liste elle même. */
char** liste_sommets(GRAPHE* g);

/* Ajoute au graphe g le sommet id. Si le sommet
   est déjà présent, ne fait rien. */
void ajouter_sommet(GRAPHE* g, char* id);

/* Ajoute au graphe g l'arête (id1, id2) avec un poids `dist`.
   Les deux sommets doivent figurer dans g. Cette fonction
   ne vérifie pas si (i1, id2) est déjà une arête du graphe. */
void ajouter_arete(GRAPHE* g, char* id1, char* id2, float dist);

/* Renvoie un pointeur vers le premier voisin de id dans g
   dans l'ordre de stockage */
VOISIN* premier_voisin(GRAPHE* g, char* id);

/* Renvoie le nom du sommet d'arrivée de l'arête représentée par v.
   Il ne faut PAS appeler free sur le résultat de cette fonction
   car le pointeur renvoyé est égal à celui stocké dans la structure GRAPHE */
char* id_voisin(VOISIN* v);

/* Renvoie le poids de l'arête représentée par v */
float dist_voisin(VOISIN* v);

/* Renvoie le voisin suivant v, NULL si v est le dernier
  voisin dans l'ordre de stockage. */
VOISIN* voisin_suivant(VOISIN* v);

/* Libère la mémoire allouée pour le graphe g */
void free_graphe(GRAPHE* g);
```