TP6: Graphes pondérés, trajet de métro

MP2I Option SI: Informatique tronc commun

1 Algorithme de Dijkstra

On considère des graphes pondérés. On rappelle que l'on peut adapter les représentations des graphes par matrices d'adjacence et par listes d'adjacence pour stocker également les poids de chaque arête. On manipulera dans ce TP exclusivement des graphes représentés par listes d'adjacence. Un graphe sera donc représenté dans nos programmes par un dictionnaire, associant à chaque sommet la liste des couples (successeur, poids de l'arête). Par exemple :

```
g = {
  "A": [("B", 3), ("C", 2), ("E", 1)], # A a trois voisins. L'arête (A, B) a un poids 3
  "B": [("C", 2)], # B a un seul voisin C, et l'arête (B, C) a un poids 2
  "C": [("D", 2)],
  "D": [("A", 1), ("E", 3)],
  "E": [("A", 3), ("B", 2), ("D", 2)]
}
```

Question 1. Écrire une fonction `degres(g)` qui renvoie un dictionnaire dont les clés sont les sommets du graphe g et dont les valeurs sont les degrés sortants des sommets, c'est à dire leur nombre de successeurs.

On s'intéresse au calcul de plus courts chemins dans des graphes pondérés.

Question 2. Écrire une fonction `longueur(g, l)` qui prend en entrée un graphe g et une liste l de sommets de g représentant un chemin, et qui renvoie la longueur totale de ce chemin.

On rappelle le pseudo-code de l'algorithme de Dijkstra, qui permet de calculer des plus courts chemins d'un sommet source vers tous les autres sommets. Plus précisément, cet algorithme construit un dictionnaire des prédécesseurs, comme les algorithmes de parcours vus précédemment, ce qui permet ensuite de reconstruire facilement les chemins :

Algorithme 1 : PCC : Dijkstra

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

```
d ← dictionnaire avec S comme clés, et ∞ pour toutes les valeurs;
Pred ← dictionnaire vide;
d[s] = 0;
Q ← ensemble contenant chaque élément de S;
tant que Q non vide faire
    u ← extraire sommet de Q avec d[u] minimal;
    pour v voisin de u faire
        si d[u] + w(u, v) < d[v] alors
            d[v] ← d[u] + w(u, v);
            Pred[v] ← u;
retourner d, Pred
```

Le tableau d renvoyé est tel que pour $u \in S$, $d[u]$ la longueur d'un plus court chemin de s à u, et **Pred**$[u]$ contient le prédécesseur de u dans un plus court chemin de s à u.

On rappelle les opérations de base des ensembles en python :

```
A = set() # création d'un ensemble vide
B = set([2, 3, 4]) # création d'un ensemble à partir d'une liste
A.add(8) # ajout d'un élément à A
B.remove(3) # suppression d'un élément de B
C = A.union(B) # création d'un nouvel ensemble, union de A et B
A.update(B) # modification de A en lui ajoutant les éléments de B
```

Question 3. Écrire une fonction `extraire_min(Q, d)` qui prend en entrée un ensemble de sommets Q, ainsi qu'un dictionnaire d dont les éléments de Q sont des clés, et qui renvoie un élément $u \in Q$ tel que $d[u]$ est minimal. Cette fonction modifiera l'ensemble Q pour lui retirer u.

Question 4. Écrire une fonction `dijkstra(g, s)` qui lance l'algorithme de Dijkstra depuis le sommet s et renvoie les tableaux des distances et des prédécesseurs sous la forme d'un couple (s, p).

Question 5. En déduire une fonction `pcc(g, s, t)` qui renvoie un plus court chemin de s à t dans g sous la forme d'une liste.

Question 6. Créer quelques graphes et tester votre algorithme.

2 Trajet dans le métro

On se propose d'écrire un programme Python qui calcule des trajets optimaux dans le métro parisien, en utilisant l'algorithme de Dijkstra.

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. Plus précisément, chaque fichier contient :

- Sur la première ligne, deux entiers n et m indiquant le nombre de sommets et d'arêtes du graphe
- Sur les n lignes suivantes, les noms des sommets du graphe
- Sur les m lignes suivantes, des triplets $(u, v, w(u, v))$ indiquant les arêtes du graphe. Les trois composantes des triplets sont séparées par un \$.

Vous trouverez dans l'archive du TP une base de code, contenant les trois fonctions suivantes :

```
def charger_graphe(filename):
    """
    Entrée: filename un nom de fichier ou un chemin vers un fichier
            contenant les informations d'une ligne de métro au format
            spécifié dans le TP
    Sortie: graphe construit à partir du fichier lu, sous forme de dictionnaire
            d'adjacence
    """

def creer_dictionnaire_positions(filename):
    """
    Entrée: filename nom d'un fichier CSV contenant les coordonnées GPS
    des stations de métro
    Sortie: dictionnaire associant à chaque nom de station son couple (latitude, longitude)
    """

def afficher_graphe(G, positions):
    """
    Entrées:
    - graphe G (par dictionnaire d'adjacence)
    - positions dictionnaire associant à chaque sommet de G ses coordonnées
    Sortie: Dessine le graphe G avec chaque sommet à sa position indiquée
    """
```

La première fonction, `charger_graphe`, est incomplète, et contient des commentaires indiquant les étapes à rajouter.

Question 7. Compléter la fonction `charger_graphe`. Tester en chargeant le graphe de la ligne 1 et en l'affichant, à l'aide des trois fonctions.

Question 8. Charger le graphe de la ligne 1 et vérifier sur un plan en ligne que le plus court chemin renvoyé par votre algorithme entre Concorde et Bastille est cohérent.

Pour commencer, nous allons réunir les informations des différentes lignes en un seul graphe.

Question 9. Écrire une fonction `fusion_graphes(L)` qui prend en entrée une liste L de 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.

Deux sommets ayant le même nom dans deux graphes distincts de L seront donc fusionnés en un seul et unique sommet. Ainsi, une station présente sur plusieurs lignes donnera lieu à un seul sommet.

Question 10. Sur le graphe construit à partir de toutes les lignes, 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 a-t-il ?

On veut maintenant modéliser le fait qu'un changement n'est pas instantané, et que les jours fériés et les weekends, un changement peut même être très coûteux.

Pour cela, nous allons représenter une station S présente sur k lignes par k sommets $S_1, \dots S_k$. Ces k sommets sont tous reliés entre eux, par des arêtes dont le poids sera le temps de correspondance. Plus précisément :

Définition 1. Étant donné p graphes $G_1, \dots G_p$ et $t \geq 0$ un temps d'attente, le graphe fusion de $G_1, \dots G_p$ avec temps de correspondance t, noté $G = \mathbf{Corresp}^t(G_1, \dots G_p)$, est le graphe obtenu en mettant côte à côte les graphes $G_1, \dots G_p$, puis en connectant tous les sommets de même nom, par une arête de poids t.

Question 11. 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 cette opérations 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 ainsi remplacée par 4 sommets : `Gare Montparnasse#4`, ..., `Gare Montparnasse#13`.

Question 12. Écrire une fonction `correspondances(L, t)` qui prend en entrée une liste $L = [G_1, \dots, G_p]$ de graphes ainsi qu'un temps d'attente t, et crée le graphe $G = \mathbf{Corresp}^t(G_1, \dots, G_p)$.

Nous avons maintenant tous les outils nécessaires pour écrire un programme donnant des trajets de métro efficaces.

Question 13. Écrire une fonction `trajet(depart, arrivee, t)` qui prend en entrée deux noms de stations, ainsi que le temps d'attente t estimé à chaque changement et qui affiche la suite des stations sur un plus court chemin. Votre programme affichera aussi le temps total de trajet.

Question 14. 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, même s'ils empruntent plus de stations.

Question 15. 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: 22 minutes):

1) Ligne 9 de Porte d'Auteuil jusqu'à Chaussée d'Antin La Fayette (15 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)
```

Question 16. Enfin, rajoutez à votre programme la possibilité d'afficher le chemin sur le plan du métro (par exemple en le traçant d'une autre couleur que le reste des arêtes).