Lycée Loges

Chapitre 11

Théorie des Graphes TP

Simon Dauguet
simon.dauguet@gmail.com

mercredi 10 juin, 2026

Exercice 1 :

On souhaite déterminer si un graphe ni orienté ni pondéré défini dans un fichier est connexe. Le fichier `CPGE-graphes-G1.txt` contient la définition d'un graphe $\mathcal{G}_1$ dont les sommets sont numérotés par des entiers.
Ci-contre les 4 premières lignes du fichier.
La 2è ligne énumère l'ensemble des sommets triés dans l'ordre, puis suit à partir de la 4è ligne l'ensemble des arêtes.
Ainsi nous savons que le graphe possède 7 sommets et qu'une arête existe entre le sommet 0 et 2.

```
Sommets
0, 1, 2, 3, 4, 5, 6
Aretes
0, 2
⋮
```

1. Ouvrir le fichier et déterminer les matrices S et A représentant le sommet et les arêtes du graphe.
2. Définir une fonction `lectureGraphe()` qui renvoie les deux matrices S et A définies à partir du fichier `CPGE-graphes-G1.txt`.
3. Définir une fonction `matriceAdjacence(S: list, A: list) -> list`, qui prend en argument S et A et renvoie la matrice d'adjacence M.
4. Définir une fonction `existeArete(i: int, j: int, M: list) -> bool`, qui prend en argument deux sommets et la matrice d'adjacence M et renvoie `True` si l'arête reliant les sommets i et j existe ; `False` sinon.
5. Définir une fonction `existeChaine(i: int, j: int, M: list) -> tuple`, qui prend en argument deux sommets et la matrice d'adjacence M et renvoie le couple $(0,0)$ si aucune existe ; (p,m) où p est la longueur de la plus petite chaîne entre i et j ; m le nombre de telles chaînes.
6. Pour chacun des algorithmes, spécifier le coût temporel des algorithmes.

le graphe $\mathcal{S} = \{A,B,C,D,E,F\}$ et $\mathcal{A} = \{(A,B,5),(C,D,7),(C,E,8),(C,F,9)\}$ représenté par la matrice $M = \begin{pmatrix} 0 & 5 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7 & 8 & 9 \\ 0 & 0 & 7 & 0 & 0 & 0 \\ 0 & 0 & 8 & 0 & 0 & 0 \\ 0 & 0 & 9 & 0 & 0 & 0 \end{pmatrix}$ peut également être représentée par la liste A=[(0,1,5),(2,3,7),(2,4,8),(2,5,9)] ou même mieux par la liste (de listes) L={[(1,5)],[],[(3,7),(4,8),(5,9)]} *i.e.* $\begin{pmatrix} (1,5) & & \\ () & & \\ (3,7) & (4,8) & (5,9) \end{pmatrix}$.

Remarque : afin que $|L| = |\mathcal{S}| = n$, on peut compléter la liste par des sous-listes vides : dans ce cours, on appellera cela une liste complétée.

Sous python cela donne :

```
L=[[(1,5)],[],[(3,7),(4,8),(5,9)]]                    # liste simple
# ou completee par des sous-listes vides :
L2=[[(1,5)],[],[(3,7),(4,8),(5,9)],[],[],[]]          # liste completee
```

Exercice 2 :

Pour cet exercice, vous pouvez compléter le fichier **CPGE-graphes-G2.py**.

On considère le graphe pondéré non orienté : $\mathcal{S} = \{A, B, C, D, E\}$ et $\mathcal{A} = \{(A, B, 2), (A, D, 6), (B, C, 4), (B, D, 2), (C, D, 1), (C, E, 4), (D, E, 5)\}$

1. Tracer le graphe et préciser sa matrice d'adjacence.
2. Appliquer l'algorithme de Dijkstra pour trouver la chaîne de poids minimal entre les sommets A et E.
3. Définir une fonction **listeSommets(A: list) -> list**, qui prend en argument une liste **A** représentant les arêtes d'un graphe connexe non orienté et qui renvoie la liste des sommets du graphe.
4. Définir une fonction **index(s:str, sommets:list) -> int** qui renvoie le numéro du sommet **s** dans la liste de sommets **sommets**.
5. Définir une fonction **matAdj(A:list) -> list** qui prend en argument la liste des arêtes d'un graphe connexe non orienté et qui renvoie la matrice d'adjacence correspondante.
6. Définir une fonction **dijkstraMat(ini: int, fin: int, A: list) -> list**, qui prend en argument deux sommets **ini**, **fin** et une liste d'arêtes pondérées **A**, et renvoie la chaîne de poids minimal entre les sommets **ini** et **fin** ainsi que son poids.
7. En fin de fichier, vous avez un graphe connexe dont les sommets sont composés de plusieurs capitales sud-américaines dans l'ordre alphabétique (à savoir Asunción, Bogota, Brasilia, Buenos Aires, Caracas, Lima, Montevideo, Quito, Santiago, Sucre). Les poids associés aux arêtes représentent le temps (arrondi à l'heure près) pour relier deux capitales par la route. Ce graphe est représentée dans le fichier **CPGE-graphes-Villes.pdf**. Tester votre fonction et vérifier les résultats (*i.e.* Asunción-Caracas=126 ; Buenos Aires-Quito=85 ; Montevideo-Quito=90).
8. Préciser la complexité temporelle de chacun des algorithmes précédents.

Exercice 3 :

Pour cet exercice, vous pouvez compléter le fichier **CPGE-graphes-G3.py**.

On reprend le même graphe qu'à l'exercice précédent mais on le traite sous forme de liste afin de limiter l'espace mémoire utilisé (utile en cas de matrice creuse).

1. Définir une fonction **convertList(L: list) -> list**, qui prend en argument une liste et renvoie une liste de longueur $n = |\mathcal{S}|$ avec possiblement des sous-listes vides.
2. Définir une fonction **areteSommetP(s: int, L: int) -> list**, qui prend en argument un sommet **s** et un graphe sous forme de liste **L** ; et renvoie la liste des arêtes du sommet sous la forme (s_j, p_{ij}).
3. Considérant la ligne des arêtes partant de **sc**, définir une fonction **retireSommetLigneP(s : int, ligne : list) -> list**, qui prend en argument un sommet **s** et une ligne ; et renvoie les arêtes sans le sommet **s**.
4. Définir une fonction **retireSommetListeP(s: int, L: list) -> list**, qui prend en argument un sommet **s** et la liste **L** des arêtes du graphe et renvoie les arêtes sans le sommet **s**.
5. Définir une fonction récursive **dijkstraR(sc: int, sf: int, L: list, S: list, P: list) -> list**, qui prend en argument un sommet courant et final, un graphe **L**, une liste de sommets et un liste **P** de

couples (sommet précédent, poids minimal) défini par l'algorithme de Dijkstra. (On pourra réutiliser la fonction **sommetPoidsMin** de l'exo précédent.)

Remarque : on peut arrêter l'algorithme proposé dès que $fin \notin S$.

6. Définir une fonction `dijkstraRinit(si: int, sf: int, L: list) -> tuple`, qui prend en argument deux sommets **si**, **sf**, un graphe **L** qui fait l'appel principal de **dijkstraR()** et renvoie le poids minimal de la chaîne entre les sommets **si** et **sf** ainsi que cette chaîne.
7. En fin de fichier, vous avez un graphe dont les sommets sont composés d'un grand nombre de capitales européennes dans l'ordre alphabétique (à savoir Amsterdam, Athènes, Belgrade, Berlin, Berne, Bruxelles, Bucarest, Budapest, Copenhague, Helsinski, Kiev, Lisbonne, Londres, Luxembourg, Madrid, Moscou, Oslo, Paris, Prague, Rome, Sarajevo, Sofia, Stockholm, Varsovie, Vienne, Zagreb). Les poids associés aux arêtes représentent le temps (arrondi à l'heure près) pour relier deux capitales. Ce graphe est représentée dans le fichier **CPGE-graphes-Villes.pdf**.
   (a) Peut-on savoir, par une simple lecture de la liste, le temps permettant de relier Berlin à Paris ? Bucarest à Kiev ? Bruxelles à Sarajevo ? Si oui, préciser cette durée.
   (b) Le graphe fourni est-il connexe ? (on pourra adapter la fonction **dijkstraR()**)
   (c) Calculer la durée minimale ainsi que le parcours correspondant, pour relier Lisbonne à Prague ; Athènes à Oslo.