TP sur les graphes

Stéphane Grandcolas, François Brucker, Valentin Emiya - Aix-Marseille Université

L2 MPCI - Novembre 2024

Objectif. L'objectif de ce TP est de coder des structures de données appropriées pour les graphes orientés pondérés et quelques fonctionnalités telles que le calcul des plus courts chemins par l'algorithme de Dijkstra.

Structure de graphe. Pour coder un graphe pondéré dans l'esprit de la représentation par listes d'adjacence vue en cours, on utilisera un dictionnaire de dictionnaires. Le graphe est un dictionnaire `g` dont chaque clé `u` est un sommet, et la valeur `g[u]` est un dictionnaire. Pour tout sommet `u`, le dictionnaire `g[u]` a pour clés les sommets `v` voisins de `u` et pour valeurs `g[u][v]` les poids des arcs `(u,v)`. Comme vu en cours, l'intérêt de cette représentation est de permettre une grande efficacité quand il faut parcourir le graphe, par exemple pour les algorithmes de plus courts chemins. Le recours à un dictionnaire plutôt qu'à une liste est plus naturel : l'ordre des sommets est arbitraire et leur nom n'est pas forcément un entier compris entre 0 et le nombre de sommets, comme ce serait le cas si l'on devait considérer les indices d'une liste. Vous trouverez sur Ametice un petit tutoriel `graph_dict.html` à ce sujet.

Remarque : dans la deuxième partie du TP, on réutilisera ce principe dans un cadre plus général de programmation objet.

Organisation. Vous mettrez tous les fichiers dans un même répertoire pour éviter les problèmes d'importation.

***** Les exercices marqués d'une étoile sont optionnels : faites ceux qui vous intéressent le plus, en fonction du temps que vous avez.

1 Mise en œuvre simple avec un dictionnaire

Vous utiliserez trois fichiers `graph_dict.py` pour les définitions de fonctions (fourni, à compléter), `main_graph_dict.py` pour le code utilisant ces fonctions et `test_graph_dict.py` pour les tests.

Exercice 1 (Création d'un graphe)**.** Dans un fichier `main_graph_dict.py`, créez le graphe de la Figure 1, vu en TD, en utilisant un dictionnaire de dictionnaires.

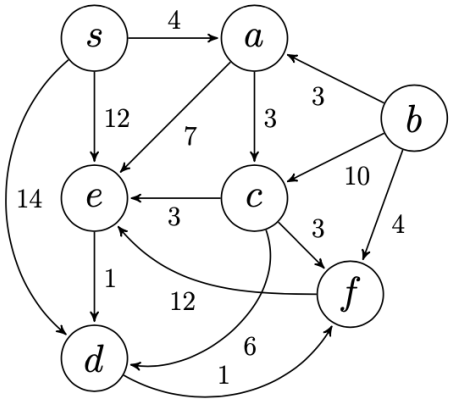

FIGURE 1 – Exemple de graphe vu en TD

Exercice 2 (Nombre de sommets et d'arcs)**.** Dans le fichier `graph_dict.py`, écrivez une fonction `get_n_vertices(graph_dict)` qui renvoie le nombre de sommets dans un graphe et une fonction `get_n_edges(graph_dict)` qui renvoie le nombre d'arcs.

Exercice 3 (Algorithme de Dijkstra)**.** Écrivez une fonction `dijkstra(graph_dict, source_vertex)` qui prend en argument un graphe et un sommet source et renvoie un dictionnaire `dist` dont les clés sont les sommets du graphe et les valeurs sont les longueurs des plus courts chemins depuis le sommet source jusqu'au sommet clé (`math.inf` si le sommet n'est pas atteignable). Vous testerez votre fonction en l'appliquant au graphe de l'exercice 1.

Exercice 4 (Prédécesseur)**.** Modifiez votre fonction `dijkstra` pour qu'elle renvoie deux dictionnaires : le dictionnaire des longueurs des plus courts chemins et le dictionnaire des sommets prédécesseurs (avec `pred[source_vertex]=None`). Pour ce dernier dictionnaire, faites un test sur l'exemple du graphe précédent.

Exercice 5 (Reconstruction d'un plus court chemin)**.** Ajoutez une fonction `shortest_path(target_vertex, pred)` qui prend en argument un sommet cible `target_vertex` et un dictionnaire de prédécesseurs `pred` et renvoie la liste composée des sommets d'un plus court chemin depuis le sommet source utilisé pour construire `pred` jusqu'au sommet cible `target_vertex`. Testez la fonction sur l'exemple précédent.

Exercice 6 (Tests unitaires)**.** Dans le fichier `test_graph_dict.py`, si ce n'est pas déjà fait, reprenez les codes de test que vous avez développés précédemment pour en faire plusieurs fonctions de test à utiliser avec `pytest`. N'hésitez pas à ajouter tout test qui vous semble utile.

2 Mise en œuvre avec une classe

Dans la partie précédente, vous avez manipulé des graphes comme des dictionnaires de dictionnaires. On souhaite à présent conserver ce principe tout en créant une classe spécifique `DiGraph` pour ces graphes orientés (*directed graph*). Autrement dit, tout graphe étant avant tout un dictionnaire, on souhaite que cette classe hérite de `dict`.

Vous organiserez votre code en séparant les classes/fonctions, les fichiers scripts d'utilisation et les tests (à ajouter des tests systématiquement, ils n'ont pas besoin d'être très élaborés).

Remarque : il s'agit là d'une implémentation proche de la classe homonyme `DiGraph` proposée dans `networkx`. La principale différence est que les graphes de `networkx` sont des imbrications de dictionnaires à trois niveaux alors que la vôtre est une imbrication à deux niveaux, avec les poids des arêtes directement insérés comme les valeurs des dictionnaires de successeurs, au second niveau.

Exercice 7 (Étude de `DiGraph` : graphes orientés)**.** Étudiez le code de la classe `DiGraph` dans le fichier `graph_obj.py`. Il n'est pas utile de changer le constructeur ni d'ajouter des attributs.

Exercice 8 (Fonctionnalités de base.)**.** En recyclant les fonctionnalités que vous avez déjà codées, implémentez les propriétés [1] `n_nodes` (nombre de sommets) et `n_edges` (nombre d'arêtes), et complétez les méthodes, `add_node`, `add_edge`, `remove_node` et `remove_edge`. Recréez aussi l'exemple de la figure 1 en utilisant les méthodes disponibles.

Exercice 9 (Algorithme de Dijkstra)**.** Écrivez la méthode `dijkstra` et une méthode statique `compute_shortest_path` en réutilisant votre code.

Exercice 10 (* Autres fonctionnalités)**.** Ajoutez les fonctionnalités suivantes :

- méthode spéciale `__str__` : renvoie une chaîne de caractères du type

  `"Directed graph with 6 nodes and 7 arcs."`

- `get_transpose` : crée et renvoie un nouveau graphe correspondant au graphe transposé du graphe courant.
- `get_induced_subgraph` : crée et renvoie le sous-graphe induit par un sous-ensemble de sommets.
- `get_adjacency_matrix` : crée et renvoie la matrice d'adjacence au format liste de liste ou `nd-array` de `numpy`, dont les coefficients sont les poids des arêtes, et `math.inf` en l'absence d'arête ; la méthode doit également renvoyer la liste des sommets correspondant à l'ordre utilisé pour définir la matrice.

Exercice 11 (* Génération de graphes aléatoires)**.** Pour faire des simulations de plus grande ampleur que sur l'exemple de la Figure 1, on peut générer des graphes aléatoirement avec des tailles arbitrairement grandes. Une possibilité est de générer des graphes *binomiaux* (ou de Erdös-Rényi) : à partir d'un nombre de sommets n et d'une probabilité $p \in [0, 1]$, on crée un graphe sans arc et pour chaque paire de sommets (u, v), on ajoute un arc de u à v avec une probabilité p (tirage selon une loi de Bernoulli de paramètre p), le poids de l'arc étant 1. Créez une méthode statique `create_binomial_graph(n, p)` qui renvoie un tel graphe.

Remarque : le graphe obtenu n'est pas forcément fortement connexe. Vous pouvez le modifier pour le rendre fortement connexe en ajoutant des arcs : par exemple, si les sommets sont ordonnés, pour tout sommet u, créer un arc (u, v) et un arc (x, u) en choisissant v et x aléatoirement tels que $v < u$ et $x < u$ (vous pouvez montrer que le graphe est alors fortement connexe, et il est inutile d'ajouter des arcs s'il en existe déjà qui satisfont l'inégalité).

1. Voir la notion de propriété dans le document *Compléments de programmation Python* fourni lors du premier TP et disponible sur Ametice.

Exercice 12 (* Version de Dijkstra avec file de priorité)**.** Ajoutez une méthode `dijkstra_heap(self, source_vertex)` qui met en œuvre l'algorithme de Dijkstra avec une file de priorité. En utilisant `heapq`, vous devrez gérer la difficulté de diminuer la priorité d'une entrée de la file, ce qui n'est pas proposé dans cette classe. Une possibilité est d'ajouter un nouvel élément avec la nouvelle priorité, en créant donc un doublon, et d'ignorer l'ancien élément. Pour cela, utilisez un ensemble initialement vide pour stocker tous les sommets extraits de la file de priorité : lors de l'extraction du sommet de poids minimal, ajoutez ce sommet à l'ensemble s'il n'y est pas déjà et ignorez-le sinon en passant directement à l'itération suivante ; en cas de doublon, l'occurrence avec le poids le plus petit sera prise en compte et les autres occurrences seront ignorées.

Exercice 13 (* Temps de calcul)**.** En générant des graphes aléatoires [2] et en ajustant leur taille pour que les calculs ne soient ni trop rapides, ni trop lents, mesurez les temps d'exécution de votre ou vos algorithme(s) de plus courts chemins. Un tuto « Mesurer les temps de calcul en Python » est disponible sur Ametice (la partie Profiling n'est pas nécessaire ici). Vous pouvez concevoir deux expériences : faire varier le nombre de sommets avec un paramètre de Bernoulli fixé ; ou fixer le nombre de sommets et faire varier le paramètre de Bernoulli. Vous tracerez ensuite les temps de calcul en fonction du paramètre qui varie, avec une courbe par algorithme (avec et sans file de priorité), sur une même figure, et une figure par expérience.

Exercice 14 (* Manipulation d'un graphe de réseau social)**.** Le fichier `facebook_combined.txt` contient un sous-graphe du graphe Facebook [3], avec environ 4000 sommets correspondant à des utilisateurs et des arêtes reliant les « amis Facebook ».

1. Chargez le graphe avec `networkx` en utilisant la fonction `read_edgelist`.
2. Affichez le graphe en calculant les positions des sommets avec la fonction `spring_layout` (ce calcul peut prendre environ une minute) ; l'affichage se fait avec la fonction `draw_networkx`, en fournissant les positions des sommets via l'argument `pos`. Commentez ce que vous obtenez. Vous pouvez exporter la figure dans un fichier image avec la fonction `savefig` de `matplotlib` pour ne pas avoir à relancer les calculs coûteux plusieurs fois.
3. Convertissez le graphe pour obtenir un objet de votre classe (via la méthode `from_nx`).
4. Affichez le nombre de sommets et le nombre d'arêtes du graphe.
5. Calculez et affichez un histogramme des degrés des sommets, c'est-à-dire le nombre de sommets de degré d pour d variant de 0 au degré maximum. Calculez également le degré moyen et le degré médian. Commentez ces résultats dans le cadre d'un réseau social.
6. Étudiez la longueur des plus courts chemins entre les paires de sommets, par exemple en affichant l'histogramme de ces longueurs.

2. Si vous n'avez pas traité l'exercice 11, vous pouvez utiliser la fonction `networkx.gnp_random_graph`
3. Source : `https://snap.stanford.edu/data/egonets-Facebook.html`