Structures de données et algorithmes avancés
Projet 2019-2020

L2 MPCI
Célia Châtel et Valentin Emiya

Version du 9 mars 2021

Résumé

Ce projet va vous permettre de mettre en application certains algorithmes vus en cours sur des graphes réels et des graphes de grandes dimensions ainsi que d'en étudier les limites et les améliorations possibles. Vous allez en particulier :

— implémenter des classes de graphes pondérés ;
— générer aléatoirement des graphes en contrôlant leur taille pour faire ensuite des tests et des expériences ;
— implémenter des algorithmes de plus courts chemins avec des versions plus ou moins rapides ;
— comparer leurs performances sur des graphes générés mais aussi sur un graphe correspondant à des données réelles, en mesurant notamment les temps de calcul des plus courts chemins.

Conseil : lisez le sujet en entier régulièrement, au moins une fois au début de chaque séance de travail. Ceci vous aidera à ne pas perdre de vue les objectifs principaux, de prendre du recul, d'organiser vos priorités (il ne s'agit pas de traiter le sujet linéairement, certaines tâches sont à laisser de côté dans un premier temps) et de vous répartir le travail.

Table des matières

1 Nouvelles classes pour les graphes **2**

2 Générer des graphes artificiellement **3**

3 Implémentation rapide de l'algorithme de Dijkstra **4**

4 Expériences sur les graphes générés artificiellement **5**

5 Expériences sur les données Reddit Hyperlink Network **5**

6 Comparaison avec le module `networkx` **6**

Annexe A Outils méthodologiques **6**
A.1 Mesure des temps de calcul 6
A.2 Tracé de courbes 7
A.3 Profiling 8

Annexe B Diagramme de dépendances entre les tâches **8**

1 Nouvelles classes pour les graphes

On cherche à modéliser les graphes pondérés, orientés ou non-orientés.

Un graphe sera représenté par un unique attribut `edges` : un dictionnaire de dictionnaires qui, à un sommet u associe le dictionnaire qui, à un sommet v **voisin** (successeur) de u associe le poids de l'arête entre u et v. En pratique, vous pouvez utiliser des entiers pour représenter les sommets.

Nous travaillerons ensuite avec l'algorithme de Dijkstra qui ne fonctionne que pour des arêtes de poids positifs, vous veillerez donc à ce que vos graphes ne contiennent que des arêtes de poids positifs.

Tâche 1.1 (Graphe orienté)**.** Dans un fichier `graph.py`, implémentez une classe de graphes `DirectedGraph` qui supporte :

- l'accès à ses arêtes via `graph.edges` pour le graphe `graph` ;
- l'accès à ses sommets via `graph.vertices` pour le graphe `graph` ;
- la création d'un graphe vide (constructeur sans paramètre) ;
- l'ajout et la suppression d'un sommet (`add_vertex(vertex)`, `remove_vertex(vertex)`) ;
- l'ajout et la suppression d'une arête (`add_edge(vertex1, vertex2, weight)`, `remove_edge(vertex1, vertex2)`), qui ajoute le(s) sommet(s) si nécessaire ;
- la modification du poids d'une arête `change_weight(u, v, weight)` ;
- la remise à zéro (suppression de tous les sommets et toutes les arêtes) `reset()` ;
- le calcul d'un sous-graphe induit `induced_subgraph(subset)` ;
- l'accès à la taille du graphe (son nombre de sommets) via `len(graph)` pour le graphe `graph` ;
- l'accès aux voisins d'un sommet donné (`graph[2]` par exemple pour accéder aux voisins du sommet 2 dans le graphe `graph`) en définissant `__getitem__` ;
- le parcours des sommets du graphe par une boucle `for` sur le graphe `graph` (pour cela, on écrit la méthode `__iter__` comme un générateur [1]).
- l'affichage des sommets et arêtes sous une forme lisible avec la commande `print(graph)`.

Vous veillerez à respecter les noms de méthodes et attributs qui vous sont donnés afin de pouvoir utiliser le code que nous vous fournirons.

Tâche 1.2 (Test de 1.1)**.** Dans un fichier `test_graph.py`, vérifiez que votre classe fonctionne comme attendu en testant le code suivant :

```
graph = DirectedGraph()
graph.add_vertex(1)
graph.add_edge(1, 2, 1)
graph.add_edge(2, 1, 1)
graph.add_edge(2, 3, 1)

print(graph.vertices) # affiche dict_keys([1, 2, 3])
print(len(graph)) # affiche 3
print(graph[2]) # affiche {1: 1, 3: 1}
print(graph)

for vertex in graph:
    print(vertex)
```

1. https://anandology.com/python-practice-book/iterators.html#generators

```
graph.remove_edge(1, 2)
print(graph)
```

Tâche 1.3 (Graphe non-orienté)**.** Dans le fichier `graph.py`, implémentez une nouvelle classe de graphes `UndirectedGraph`, qui hérite de `DirectedGraph` et qui a les mêmes fonctionnalités que `DirectedGraph` mais représente un graphe non orienté. Attention à bien modifier tous les comportements qui doivent l'être.

Tâche 1.4 (Test de 1.3)**.** En complétant le fichier `test_graph.py`, vérifiez que votre classe fonctionne comme attendu en testant le même code que pour les graphes orientés.

2 Générer des graphes artificiellement

Pour tester et expérimenter, vous aurez besoin de graphes de tailles variées et dont les propriétés sont bien contrôlées. Dans cette partie, vous allez créer des fonctions qui génèrent ces graphes de façon artificielle. Les fonctions de cette partie seront écrites dans un fichier `graph_generation.py` et les tests dans `test_graph_generation.py`.

Tâche 2.1 (Générateur de graphes aléatoires non orientés de n nœuds et m arêtes)**.** Écrivez une fonction `generate_random_graph(n_nodes, n_edges)` qui renvoie un objet graphe de `n_nodes` nœuds et `n_edges` arêtes choisies au hasard. Le résultat sera une instance de `UndirectedGraph` et doit être un graphe connexe.

Tâche 2.2 (Test de `generate_random_graph`)**.** Écrivez un test qui génère plusieurs graphes avec `generate_random_graph` et qui vérifie que le résultat comporte le bon nombre de nœuds et d'arêtes. N'oubliez pas de tester les cas extrêmes (nombres minimal et maximal d'arêtes).

Tâche 2.3 (Optionnel – Générateur de graphes aléatoires orientés de n nœuds et m arêtes)**.** Ajouter un argument `directed` à la fonction `generate_random_graph` qui permet, lorsqu'il vaut `True`, de générer un graphe orienté (pas forcément fortement connexe).

Les graphes de communauté sont très utilisés, notamment pour modéliser des réseaux sociaux ou des comportements/profils d'utilisateurs. L'idée est que les nœuds se répartissent en sous-ensembles appelés communautés : il y a beaucoup d'arêtes à l'intérieur de chaque communauté, et peu d'arêtes entre des nœuds de communautés différentes.

Tâche 2.4 (Générateur de graphes aléatoires de communautés)**.** Écrivez une fonction

```
generate_random_community_graph(n_nodes_per_community, p_intra, p_inter)
```

qui prend en arguments :

- une liste d'entiers naturels `n_nodes_per_community`, dont la longueur k correspond au nombre de communautés souhaité et chaque élément au nombre de nœuds dans chaque communauté,
- une probabilité `p_intra` qu'il y ait une arête entre deux nœuds d'une même communauté,
- une probabilité `p_inter` qu'il y ait une arête entre deux nœuds de deux communautés différentes,

et qui renvoie un objet graphe non orienté contenant k communautés avec des nœuds tirés aléatoirement selon les probabilités données en arguments.

Remarque : il n'est pas évident de tester un code qui se comporte de manière aléatoire. Vous pouvez dans un premier temps faire des tests qui sont indépendants du caractère aléatoire du code (tester le nombre de nœuds, les cas extrêmes de probabilités 0 ou 1). Ensuite pour

vérifier que le nombre aléatoire d'arêtes générées est cohérent avec les statistiques utilisées, vous pouvez utiliser la notion d'intervalle de confiance. Après N tirages $X_1, \ldots, X_N$ indépendants selon une loi de Bernoulli de paramètre p ($\forall n, p(X_n = 1) = p = 1 - p(X_n = 0)$), le nombre $Y_N = \sum_{n=1}^{N} X_n$ de tirages égaux à 1 suit une loi binomiale. Lorsque N est grand, la probabilité que $Np - z\sqrt{Np \times (1-p)} \leq Y_N \leq Np + z\sqrt{Np \times (1-p)}$ est supérieure à 0.95 pour $z = 1.96$ (95% de confiance) et à 0.99 pour $z = 2.576$ (99% de confiance).

3 Implémentation rapide de l'algorithme de Dijkstra

Vous avez vu en cours l'algorithme de Dijkstra. Sur de gros graphes, l'algorithme tel quel peut vite devenir trop lent pour être utilisé en pratique.

Dans une implémentation classique, on utilise une liste (ou un dictionnaire) pour représenter les noeuds et leur `dist` à l'origine courant. Rechercher le noeud dont le `dist` est le plus petit coûte alors un parcours de la structure entière. Une amélioration possible du temps de calcul de l'algorithme de Dijkstra est d'utiliser un *tas binaire* à la place. Cette structure permet de récupérer en temps constant le plus petit élément.

Un *tas binaire* est un arbre binaire complet (un arbre binaire tel que tous les niveaux, sauf potentiellement le dernier, sont entièrement remplis ; si le dernier niveau n'est pas plein, il est rempli de gauche à droite). Un tas binaire respecte de plus la propriété de tas : pour chaque noeud, la clé qui lui est associée est supérieure ou égale (ou inférieure ou égale selon l'ordre choisi) aux clés de ses enfants. Le plus grand élément (ou le plus petit selon l'ordre choisi) est ainsi la racine de l'arbre.

Pour plus d'informations sur la manière de conserver la structure de tas binaire lors de l'insertion d'un nouvel élément : https://en.wikipedia.org/wiki/Binary_heap.

Dans un tas, on peut (entre autres) :

— créer un tas vide ;
— ajouter un élément au tas ($\mathcal{O}(1)$ en moyenne) ;
— récupérer et supprimer l'élément le plus petit ($\mathcal{O}(1)$).

En python, ces opérations se font de la manière suivante :

— création d'un tas vide : `queue = []` ;
— ajout de l'élément x au tas : `heapq.heappush(queue, x)` ;
— récupération du plus petit élément dans une variable x et suppression de cet élément du tas : `x = heapq.heappop(queue)`.

On peut également ajouter un tuple dans un tas binaire en python. Dans ce cas les opérations de tri s'opéreront sur le premier élément du tuple.

Tâche 3.0 (Algorithme de Dijkstra, version de base). Implémentez une nouvelle méthode dans votre classe de graphe qui permet de calculer le plus court chemin d'un sommet u donné à tous les autres sommets du graphe en utilisant l'algorithme de Dijkstra dans sa version de base vue en cours.

Tâche 3.1 (Algorithme de Dijkstra, version rapide avec tas)**.** Implémentez une nouvelle méthode dans votre classe de graphe qui permet de calculer le plus court chemin d'un sommet u donné à tous les autres sommets du graphe en utilisant un tas binaire.

Tâche 3.2 (Test de 3.1)**.** Testez l'algorithme rapide dans plusieurs situations en vérifiant qu'il donne un plus court chemin de même longueur que l'algorithme original.

Tâche 3.3 (Algorithme de Dijkstra, un seul chemin)**.** Implémentez une nouvelle méthode dans votre classe de graphe qui permet de calculer le plus court chemin d'un sommet u donné à un

sommet v donné du graphe en utilisant un tas binaire. L'algorithme s'arrêtera dès qu'il trouve le plus court chemin de u à v.

Tâche 3.4 (Optionnel – Algorithme de Bellman-Ford)**.** Implémentez une nouvelle méthode dans votre classe de graphe qui permet de calculer le plus court chemin d'un sommet u donné à tous les autres sommets du graphe en utilisant la méthode de Bellman-Ford (attention, nous avons vu en cours la version permettant de calculer les plus courts chemins vers un sommet donné).

4 Expériences sur les graphes générés artificiellement

Dans cette partie, on souhaite conduire plusieurs expériences pour mettre en évidence les différences de temps de calcul des algorithmes sur des graphes non orientés, en fonction de leurs mises en œuvre. Vous ferez un script python dans un fichier différent pour chaque expérience. Pour mesurer les temps de calcul et tracer les résultats, vous pourrez utiliser les outils présentés en Annexe A.

Tâche 4.1 (Comparaison des temps de calcul des algorithmes pour des problèmes de taille croissante)**.** On souhaite comparer les temps de calcul des algorithmes sur les graphes non orientés. En choisissant un générateur de graphe, faites varier le nombre de nœuds n en prenant $m = \alpha n^2$ arêtes (pour $0 < \alpha \leq 1$) et mesurez les temps de calcul de vos algorithmes. On fixera le nœud de départ et pour chaque algorithme, on calculera le plus court chemin vers chaque nœud destination possible. Tracez le temps de calcul en fonction de n (une courbe par algorithme).

Tâche 4.2 (Comparaison des temps de calcul des algorithmes pour un nombre de sommets fixé)**.** Faites une variante de l'expérience 4.1 dans laquelle le nombre de nœuds est maintenant fixé et le nombre d'arêtes varie. Tracez le temps de calcul en fonction du nombre m d'arêtes (une courbe par algorithme) et analysez vos résultats.

Tâche 4.3 (Statistiques sur l'algorithme le plus rapide)**.** Dans cette expérience, utilisez uniquement l'algorithme le plus rapide. Pour des graphes tels que générés dans les expériences 4.1 ou 4.2, répétez la procédure suivante plusieurs fois :

— tirer deux sommets au hasard
— mesurer le temps de calcul pour calculer le plus court chemin.

Calculez les statistiques suivantes par rapport à tous les essais effectués : temps minimum, temps maximum, temps moyen, temps médian. Sur un graphique avec des axes similaires aux expériences 4.1 ou 4.2, tracez les courbes des différentes statistiques.

5 Expériences sur les données Reddit Hyperlink Network

Nous allons à présent expérimenter les algorithmes de plus courts chemins sur un graphe qui représente des données réelles.

Téléchargez le fichier `soc-redditHyperlinks-title.tsv` sur le site https://snap.stanford.edu/data/soc-RedditHyperlinks.html. Il représente le graphe des liens entre des communautés de Reddit. Reddit est un réseau social fonctionnant comme un immense groupe de forums. Les utilisateurs peuvent accéder à des "subreddits" qui sont des communautés dédiées à un thème en particulier (défini par le nom du subreddit) sur lequel les utilisateurs postent des messages et commentent le message initial ou les autres commentaires.

Nous allons étudier les liens entre les subreddit. Un lien du subreddit A vers le subreddit B signifie que quelqu'un a posté sur le subreddit A un message dont le titre contenait un lien vers le subreddit B.

Dans le fichier de données, une ligne représente un lien entre deux subreddits. Les colonnes sont, dans l'ordre :

— le nom du subreddit source ;
— le nom du subreddit cible ;
— un identifiant du post ;
— la date et l'heure à laquelle il a été posté ;
— de nombreuses informations numériques que nous n'utiliserons pas (le nombre de caractères du poste, le nombre de chiffres, le nombre de mots, une note de positivité et de négativité du post...).

Par exemple, la ligne 4 du fichier signifie qu'un utilisateur a posté sur le subreddit "ps4" un lien vers un post du subreddit "battlefield_4" le 31/12/2013 à 17 :59.

Vous trouverez sur ametice le fichier `parser.py` qui vous permettra de créer en python le graphe correspondant à cet ensemble de données. Dans ce graphe, tous les arcs sont de poids 1.

Tâche 5.1 (Degré des noeuds)**.** Trouvez les 10 subreddits qui contiennent le plus de liens vers d'autres subreddits. Trouvez le nombre de communautés qui n'ont posté aucun lien vers une autre communauté.

Tâche 5.2 (Part d'activité des subreddits)**.** Trouvez la part du total des interactions qui est réalisée par les 2% de communautés les plus actives.

Tâche 5.3 (Plus courts chemins)**.** Trouvez la longueur du plus court chemin entre `disney` et `vegan`. La longueur du plus court chemin entre `greenbaypackers` et `missouripolitics`.

6 Comparaison avec le module `networkx`

Tâche 6.1 (Conversion d'un objet `DirectedGraph` vers un objet `networkx.DiGraph`)**.** Lisez la documentation de la classe `DiGraph` de networkx [2] qui représente des graphes orientés et écrivez une méthode `to_networkx(self)` qui transforme un graphe de votre classe graphe en un graphe de cette classe.

Tâche 6.2 (Temps de calcul des plus courts chemins)**.** En utilisant les fonctions de calcul des plus courts chemins de `networkx`, mesurer les temps de calcul, comparez-les à ceux obtenus dans la partie 4 et commentez vos résultats.

Annexe A Outils méthodologiques

A.1 Mesure des temps de calcul

Pour mesurer des temps de calcul, on utilisera le module `time` [3] qui fournit différentes fonctions pour mesurer le temps. On utilisera en particulier la fonction `process_time()` qui donne des mesures précises du temps écoulé dans le programme courant.

Principe : repérez le code que vous souhaitez chronométrer, il suffit ensuite d'encadrer ce code par deux appels à `process_time()` en stockant les résultats `t0` et `t1`. Chaque appel renvoyant l'instant courant, la différence entre les deux mesures donne ensuite le temps passé à exécuter le code (voir exemple ci-dessous).

2. https://networkx.github.io/documentation/networkx-2.2/reference/classes/digraph.html
3. `https://docs.python.org/3/library/time.html`

A.2 Tracé de courbes

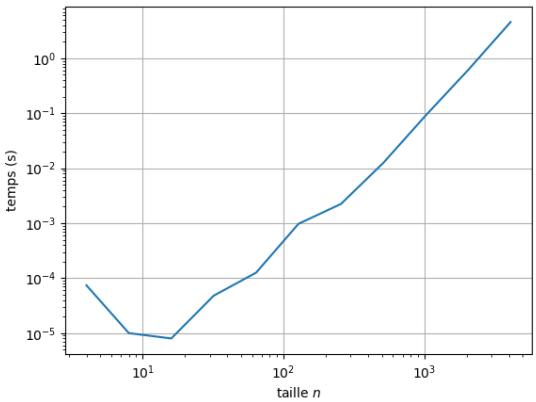
On utilisera `matplotlib`[4] et quelques unes de ses nombreuses fonctionnalités pour tracer des courbes (voir documentation de `matplotlib` pour plus de détails). L'exemple ci-dessous donne un exemple de certaines d'entre elles.

```
from time import process_time, perf_counter
import numpy as np
import matplotlib.pyplot as plt

n_range = 2**np.arange(2, 13)
t = np.empty(n_range.size)
for i, n in enumerate(n_range):
    a = np.random.randn(n, n)
    b = np.random.randn(n, n)
    t0 = process_time()  # début chronométrage
    c = np.dot(a, b)  # Code à chronométrer
    t1 = process_time()  # fin chronométrage
    t[i] = t1 - t0
    print('Temps pour multiplier 2 matrices {}x{}: {:.6f}s'.format(n, n, t[i]))

plt.plot(n_range, t)
plt.xlabel('taille $n$')
plt.ylabel('temps (s)')
plt.xscale('log')
plt.yscale('log')
plt.grid(True)
plt.savefig('exemple_process_time.png')
```

Résultat affiché :

```
Temps pour multiplier 2 matrices 4x4: 0.000074s
Temps pour multiplier 2 matrices 8x8: 0.000010s
Temps pour multiplier 2 matrices 16x16: 0.000008s
Temps pour multiplier 2 matrices 32x32: 0.000048s
Temps pour multiplier 2 matrices 64x64: 0.000126s
Temps pour multiplier 2 matrices 128x128: 0.000975s
Temps pour multiplier 2 matrices 256x256: 0.002238s
Temps pour multiplier 2 matrices 512x512: 0.012525s
Temps pour multiplier 2 matrices 1024x1024: 0.091418s
Temps pour multiplier 2 matrices 2048x2048: 0.614037s
Temps pour multiplier 2 matrices 4096x4096: 4.561713s
```

4. `https://matplotlib.org/`

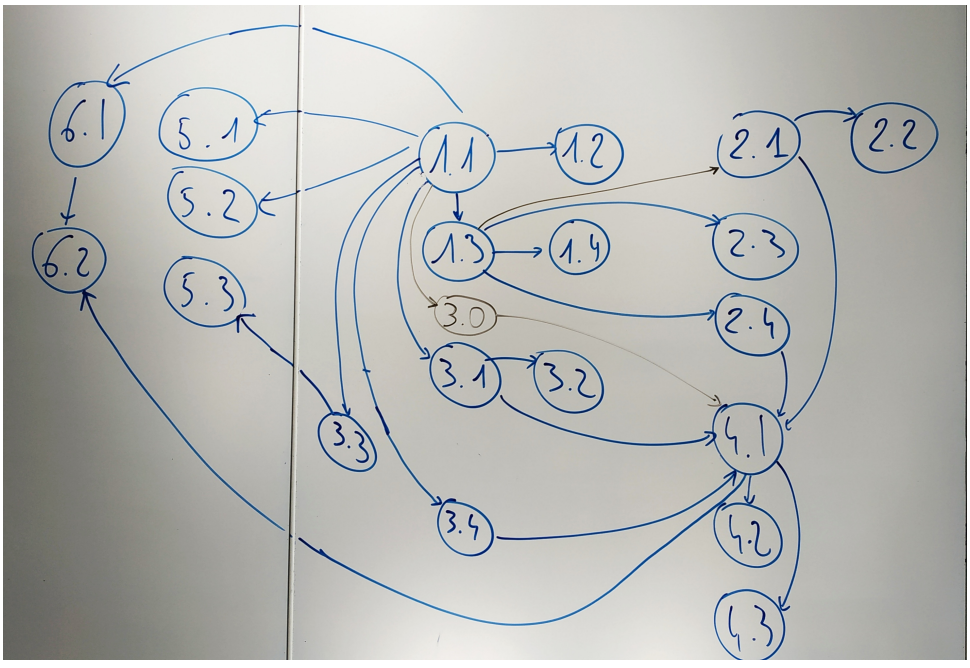


A.3 Profiling

Si l'on souhaite aller plus loin dans l'analyse des temps d'exécution, on peut utiliser des outils de profiling. Il existe un profiler [5] intégré à Python (et les environnements de développement comme PyCharm [6]).

Le module `line_profiler` [7] est un outil encore plus détaillé qui permet de mesurer le temps d'exécution de chaque ligne de code dans des fonctions ou méthodes préalablement sélectionnées. Cela permet notamment de repérer les lignes de code gourmandes en temps en vue de cibler les parties à optimiser.

Annexe B Diagramme de dépendances entre les tâches



5. https://docs.python.org/3/library/profile.html
6. https://www.jetbrains.com/help/pycharm/profiler.html
7. https://github.com/rkern/line_profiler