TP2

Very Large Graph

I. Exercice de recherche de cliques maximales dans un Graphe

Ressources :
https://fr.wikipedia.org/wiki/Clique_(th%C3%A9orie_des_graphes)
https://fr.wikipedia.org/wiki/Algorithme_de_Bron-Kerbosch
https://fr.wikipedia.org/wiki/D%C3%A9g%C3%A9n%C3%A9rescence_(th%C3%A9orie_des_graphes)

Implémentation en Python en suivant ces pseudo-codes :

Il s'agira cette fois de réaliser les implémentations de manière parallélisé.
Pour rappel, le multi-threading n'existe pas vraiment en Python à cause du GIL (possibilité de suppression dans les prochaines versions).

Librairies de multithreading en Python

En Python, il existe plusieurs bibliothèques pour gérer le multithreading. Voici quelques-unes des plus couramment utilisées :

1. **`threading`** : Il s'agit du module standard pour le multithreading en Python.
   - Il fournit des classes pour la création et la gestion de threads, ainsi que des primitives de synchronisation telles que les verrous et les sémaphores.
   - Exemple :
     ```python
     import threading

     def thread_function():
         # Code à exécuter dans le thread

     thread = threading.Thread(target=thread_function)
     thread.start()
     thread.join()
     ```

2. **`queue`** : Utilisé en combinaison avec `threading` pour fournir un moyen sûr de communiquer entre les threads.
   - Il est particulièrement utile pour implémenter des modèles de type producteur-consommateur.
   - Les queues thread-safe garantissent que les données sont ajoutées ou retirées d'une manière qui ne causera pas de conditions de concurrence.

3. **`concurrent.futures`** : Introduit dans Python 3.2, il offre une interface de haut niveau pour la création et la gestion de threads et de processus.
   - Le module fournit une classe `ThreadPoolExecutor` qui facilite la création et la gestion de pools de threads.
   - Exemple :
     ```python
     from concurrent.futures import ThreadPoolExecutor

     with ThreadPoolExecutor() as executor:
         future = executor.submit(function_name, arg1, arg2)
         return_value = future.result()
     ```

4. **`multiprocessing`** : Bien que ce module soit principalement conçu pour le multiprocessing (c'est-à-dire l'utilisation de plusieurs processus plutôt que de threads), il possède une interface similaire à celle du module `threading`, ce qui peut le rendre utile dans certains contextes où les threads ne sont pas appropriés en raison du Global Interpreter Lock (GIL) en Python.

5. **`asyncio`** : Introduit dans Python 3.3, il offre un cadre pour écrire des programmes concurrents à l'aide de coroutines. Bien qu'il ne s'agisse pas techniquement de multithreading, il permet une exécution concurrente et est souvent utilisé dans des situations où le multithreading pourrait également être considéré.

Il est essentiel de noter que, en raison du Global Interpreter Lock (GIL) en CPython (l'implémentation standard de Python), le multithreading peut ne pas offrir une amélioration significative des performances pour les tâches intensives en CPU. Dans de tels cas, le multiprocessing ou l'utilisation d'une autre implémentation de Python (comme Jython ou IronPython) pourrait être une meilleure option.

Les graphes seront représentés avec Networkx, l'affichage avec Matplotlib et les graphes peuvent être généré de manière aléatoire (par exemple Erdos Renyi https://networkx.org/documentation/stable/reference/generated/networkx.generators.random_graphs.erdos_renyi_graph.html)

Simple/naïf

```
fonction recherche_clique(graphe G):
    n = nombre de sommets de G
    taille_max_clique = 0
    clique_max = ensemble vide

    pour chaque sous-ensemble S de sommets de G:
        si la taille de S > taille_max_clique et est_clique(G, S):
            taille_max_clique = taille de S
            clique_max = S
```

```
    retourner clique_max

fonction est_clique(graphe G, ensemble S):
    pour chaque paire de sommets v, w dans S:
        si v n'est pas adjacent à w dans G:
            retourner faux
    retourner vrai
```

Bron Kerbosch

```
algorithme BronKerbosch1(R, P, X)
    si P et X sont vides alors
        déclarer que R est une clique maximale
    pour tout sommet v dans P faire
        BronKerbosch1(R ∪ {v}, P ∩ N(v), X ∩ N(v))
        P := P \ {v}
        X := X ∪ {v}

BronKerbosch1(∅, V, ∅) //appel initial
```

Bron Kerbosch pivot

```
algorithme BronKerbosch2(R, P, X)
    si P et X sont vides alors
        déclarer que R est une clique maximale
    choisir un sommet pivot u dans P ∪ X
    pour tout sommet v dans P \ N(u) faire
        BronKerbosch2(R ∪ {v}, P ∩ N(v), X ∩ N(v))
        P := P \ {v}
        X := X ∪ {v}
```

Bron Kerbosch pivot et dégénérescence

```
algorithme BronKerbosch3(G)
    P = V(G)
    R = ∅
    X = ∅
    pour tout sommet v visités dans un ordre de dégénérescence de G
faire
        BronKerbosch2({v}, P ∩ N(v), X ∩ N(v))
        P := P \ {v}
        X := X ∪ {v}
```

Dégénérescence

- Initialiser la liste de sortie *L* à la liste vide.
- Calculer une valeur d_v pour chaque sommet *v* de *G*, qui est le nombre de voisins de *v* qui n'est pas déjà dans *L* (initialement, il s'agit donc du degré des sommets dans *G*).

- Initialiser un tableau D tel que $D[i]$ contienne la liste des sommets v qui ne sont pas déjà dans L pour lesquels $d_v = i$.
- Initialiser la valeur k à 0.
- Répéter n fois:
    - Parcourir les cellules du tableau $D[0]$, $D[1]$, ... jusqu'à trouver un i pour lequel $D[i]$ est non-vide.
    - Mettre k à $\max(k,i)$.
    - Sélectionner un sommet v de $D[i]$, ajouter v en tête de L et le retirer de $D[i]$.
    - Pour chaque voisin w de v qui n'est pas déjà dans L, retirer une unité de d_w et déplacer w de la cellule de D correspondant à la nouvelle valeur de d_w.

PySpark

Même exercice que précédemment, en implémentant les différentes versions d'algorithmes de recherches de cliques maximales dans un graphe non-orienté et connexe en utilisant Spark pour cette fois paralléliser, l'avantage est que les fonctions d'agrégation (map, reduce, filter) pourront être distribuées sur plusieurs machines.
Attention, installation de Spark et Java 8.