Systèmes, Paradigmes et Langages pour le Big Data

Projet - M2 MIAGE ID

Farès AMIAR
Soufiane BOUSTIQUE
Johana LABOU

25 février 2023

Table des matières

1 Introduction

1.1 Rappel de la problématique de l'article

Dans ce projet, nous nous intéressons à la recherche de composantes connexes dans un graphe non orienté. Pour ce faire, nous nous sommes basés sur cet article de recherche, qui traite de cette problématique bien connue des informaticiens.

La quantité de données à traiter ayant considérablement augmenté ces dernières années, notamment en raison de nombreux réseaux sociaux tels que Facebook, LinkedIn, Twitter et autres, l'analyse de graphes massifs nécessite de nouveaux types d'algorithmes.

En effet, les centaines de millions d'utilisateurs et leurs connexions entre eux sont autant de données à traiter qui nécessitent de distribuer le travail sur plusieurs ordinateurs, d'où le besoin d'implémenter de nouveaux algorithmes parallélisés hautement évolutifs. L'article propose un algorithme pour trouver des composants connectés, qui est basé sur le modèle de programmation MapReduce et est mis en œuvre à l'aide de Hadoop.

1.2 But de notre projet

Le but de ce projet était donc de coder en PySpark l'algorithme MapReduce qui permet d'identifier les composantes connexes d'un graphe non orienté. En premier lieu, nous allons présenter deux versions de cet algorithme que nous avons codé en PySpark avec des explications illustrées par des schémas. Nous proposerons d'ailleurs une correction du schéma présenté dans le papier de recherche, qui comporte une erreur. Nous présenterons par ailleurs un module qui permet de tester nos algorithmes sur des graphes générés aléatoirement. Enfin, nous présenterons des analyses que nous avons menées pour comparer les différentes versions de notre algorithme ainsi que leurs performances.

Ici le lien vers notre notebook sur Google Colab.

2 Algorithme MapReduce

Dans cette partie, nous présentons les algorithmes que nous avons codé pour détecter les composantes connexes à partir du papier de recherche fourni. Nous avons codé deux versions de cet algorithme : ces deux versions sont équivalentes et produisent le même résultat. Cependant, leur implémentation diffère. La première version a été faite sans utiliser l'élément 'yield' de Python alors que la deuxième version l'utilise (et ressemble donc un peu plus au code affiché dans le papier de recherche).

2.1 Version 1

La première version de l'algorithme est implémentée par la fonction 'MapReduce_CFF_Iterate'. Cette fonction prend en entrée un RDD qui contient les arêtes de notre graphe (comme spécifié dans le papier de recherche) et un booléen pour choisir si la méthode avec 'sorting' doit être appliquée ou pas.

La première partie de cette fonction est un MAPPER, qui permet d'associer à chaque arête (X-Y) son opposé (Y-X). Ensuite, nous effectuons un shuffle avec la fonction 'groupByKey'. Et enfin dans la partie REDUCER, on filtre les éléments dont le min $<$ key et emit(value,min) tout en comptant le nombre de nouvelles paires dans la variable 'counter_new_pair'. Voici un affichage qui schématise l'enchaînement de notre fonction pour bien comprendre son fonctionnement.

Exemple avec RDD = [('A', 'B'), ('B', 'D'),('D', 'E'),('B', 'C')] et sorting = False.

2.1.1 Mapper

Ici on associe à chaque arête (X-Y) son opposé (Y-X).

Le RDD devient : [('A', 'B'), ('B', 'A'), ('B', 'D'), ('D', 'B'), ('D', 'E'), ('E', 'D'), ('B', 'C'), ('C', 'B')]

2.1.2 Shuffle

Ici, on regroupe tout simplement les éléments RDD avec leurs clés.

Le RDD devient : [('A', ['B']), ('B', ['D', 'A' ,'C']), ('C', ['B']), ('D', ['B', 'E']), ('E', ['D'])]

Ensuite, ce qui nous intéresse dans cette partie, c'est de détecter le minimum de chaque deuxième élément de la liste et placer ce minimum en début de liste. Par exemple, si nous prenons l'élément ('B', ['D', 'A' ,'C']), sa liste est ['D', 'A' ,'C'], son minimum est 'A' la liste devient donc ['A','D','C'], on effectue cette opération grâce à un Mapper et la fonction 'place_min_first' qui permet de placer le minimum d'un liste en tête de liste.

Finalement, le RDD devient : [('A', ['B']), ('B', ['A','D','C']), ('C', ['B']), ('D', ['B', 'E']), ('E', ['D'])]

2.1.3 Reducer

Dans cette partie, on commence par filtrer les éléments dont le min < key : l'élément ('A', ['B']) va être supprimé car A n'est pas supérieur à B.
Le RDD devient : [('B', ['A', 'D', 'C']), ('C', ['B']), ('D', ['B', 'E']), ('E', ['D'])]

Pour compter le nombre de nouvelles paires, pour chaque élément il faut compter le nombre d'éléments de sa liste sans le minimum ce qui revient à calculer len(L)-1. Par exemple, pour ('B', ['A', 'D', 'C']) le nombre de nouvelles paires est 2. En répétant ce processus avec le reste des éléments on obtient le nombre de nouvelles paires = 3.

Ensuite, il faut pour chaque élément emit(key, min) et emit(value, min) (pour ce deuxième emit comme on a placé le minimum en tête de liste il suffit juste de développer ce minimum avec le reste de la liste) ce qu'on fait grâce à un 'flatmap' dans la variable out_rdd.

Par exemple, pour l'élément ('B', ['A', 'D', 'C']) :

— pour emit(key,min) alors on va émettre : ('B','A)

— pour emit(value,min) alors on va émettre : ('D','A') , ('C','A')

On répète ce processus pour le reste des éléments et on enlève enfin les doublons grâce au distinct qui correspond au CFF-Dedup du papier de recherche.

Le RDD devient : [('B', 'A'), ('E', 'D'), ('D', 'A'), ('D', 'B'), ('E', 'B'), ('C', 'A'), ('C', 'B')]

Si on avait choisi sorting = True, au lieu de placer le minimum au début de la liste on aurait trié toute la liste ce qui peut prendre plus de temps dans certains cas, c'est pour cela que nous avons codé 'place_min_first'.

2.2 Version 2

Cette deuxième version 'MapReduce_CFF_Iterate_1' ressemble un peu plus à la manière de coder au papier de recherche car on utilise un yield comme emit. À noter que cette version a la même fonction que la première mais on a préféré coder deux versions pour comparer leur temps d'exécution et voir si une version est meilleure que l'autre. En effet, l'avantage du yield est qu'il permet de réduire le nombre de map utilisé, ce qui peut affecter le temps de calcul, bien qu'on ait essayé dans la version 1 de limiter au minimum le nombre de map.

Pour cette partie, on utilise des sous-fonctions map_func, reduce_func et reduce_func_sorting qui seront utilisées dans la fonction principale MapReduce_CFF_Iterate_1. Ces sous-fonctions correspondent exactement aux fonctions map et reduce de CCF-Iterate et CCF-Iterate (w. secondary sorting) du pseudo-code du papier de recherche, le CCF-Dedup correspont au .disctinct(). Dans la fonction MapReduce_CFF_Iterate_1, on applique les sous-fonctions citées, on rajoute juste une partie shuffle grâce à un groupByKey(). Enfin, pour compter le nombre de nouvelles paires dans les sous-fonctions, on émet dans le tuple 1 ou 0 puis on compte le nombre de 1 pour connaître le nombre de nouvelles paires.

3 Algorithme Principal

L'algorithme principal 'algo_main' va tout simplement itérer tant que le nombre de nouvelles paires n'est pas égal à 0. Le dernier RDD qui est émis avec counter_new_pair = 0, contient nos composantes connexes. Ces composantes sont contenues dans la variable *composants*. Par ailleurs, on stocke dans un dictionnaire toutes les informations utiles (temps d'exécution, nombre d'itérations, la somme totale des counter_new_pair, etc.) pour des analyses qu'on présentera plus tard.

Voici son schéma :

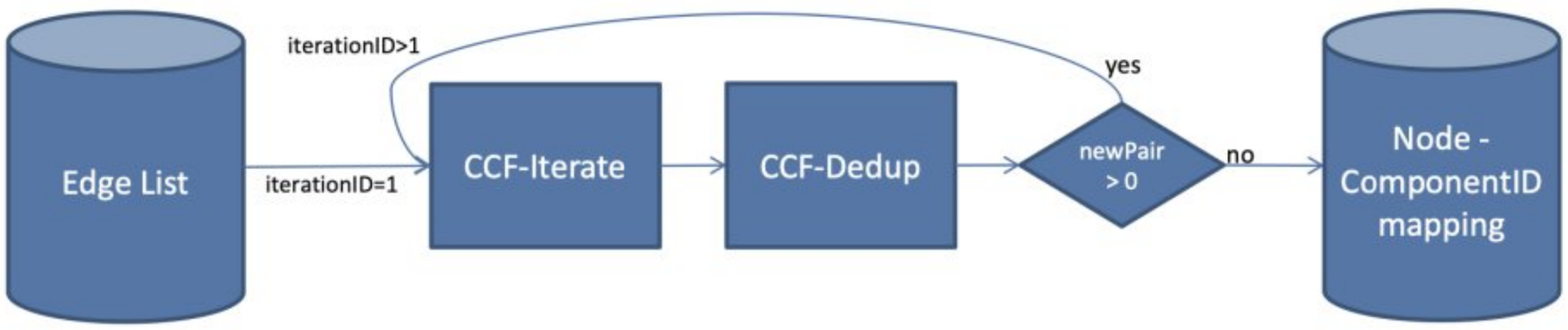
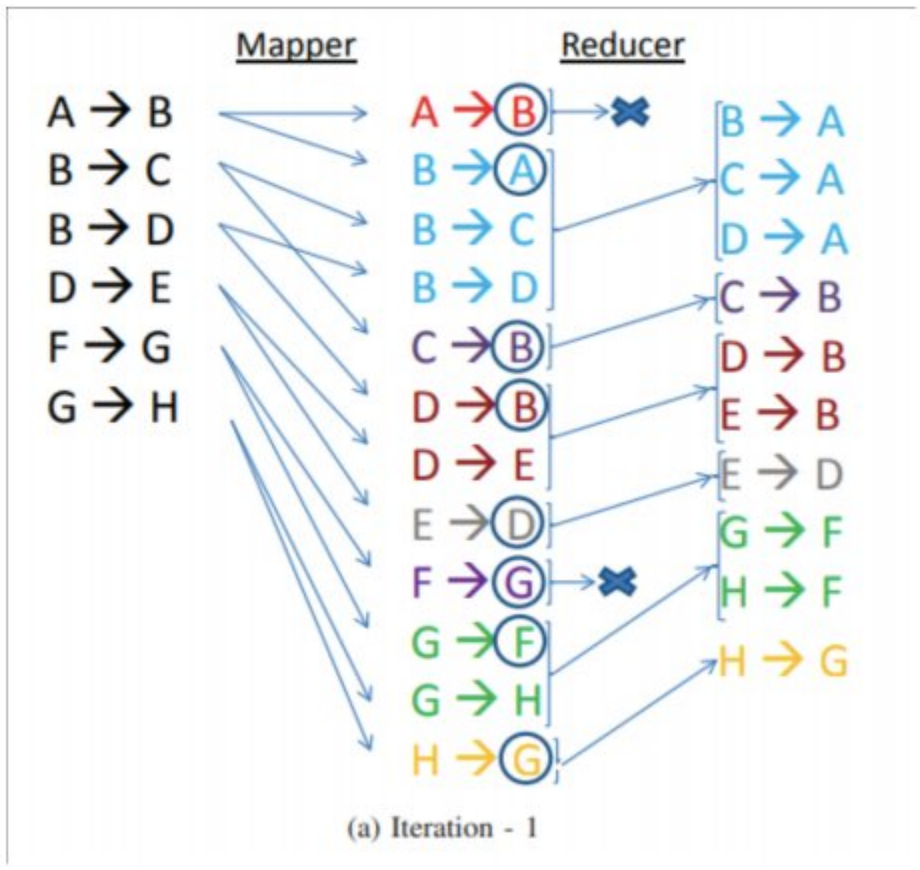


Figure 1. Connected Component Finder(CCF) Module

4 Exemple du papier de recherche

Nous avons détecté une erreur dans le papier de recherche. Au niveau de la première itération après le mapper, l'arête H->G est supprimé alors qu'elle devrait être conservée car H>G. Ceci affecte donc le reste des itérations et le nombre de nouvelles paires par itération.

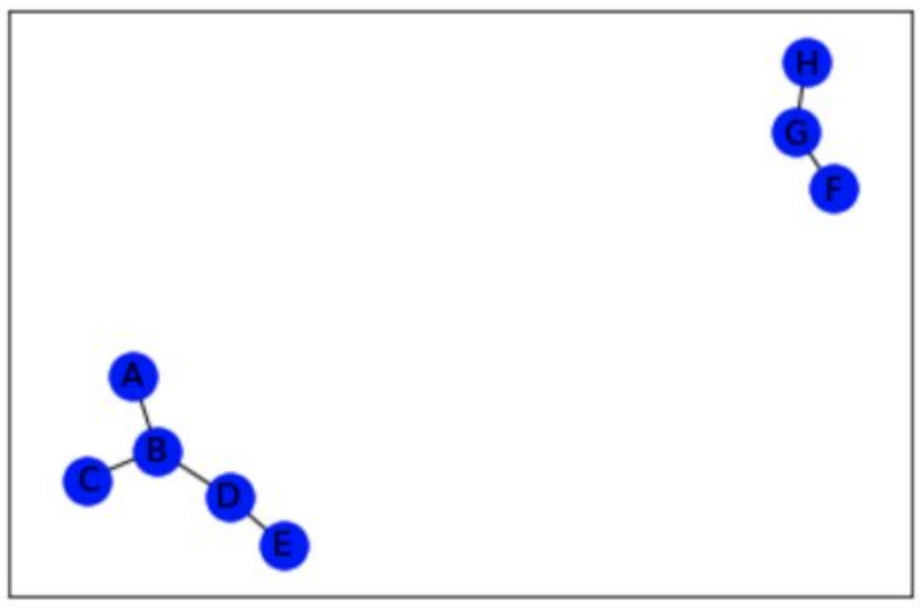
Voici la correction de la première itération du schéma :



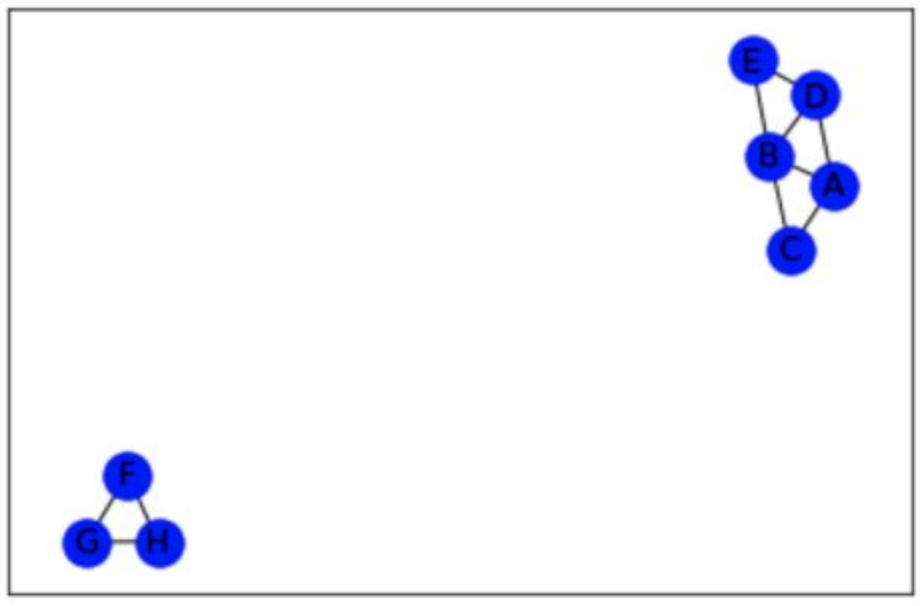
(a) Iteration - 1

Voici aussi des schémas qui explicitent l'enchaînement de l'algorithme sur le RDD de l'exemple du papier de recherche jusqu'à trouver les composantes connexes :

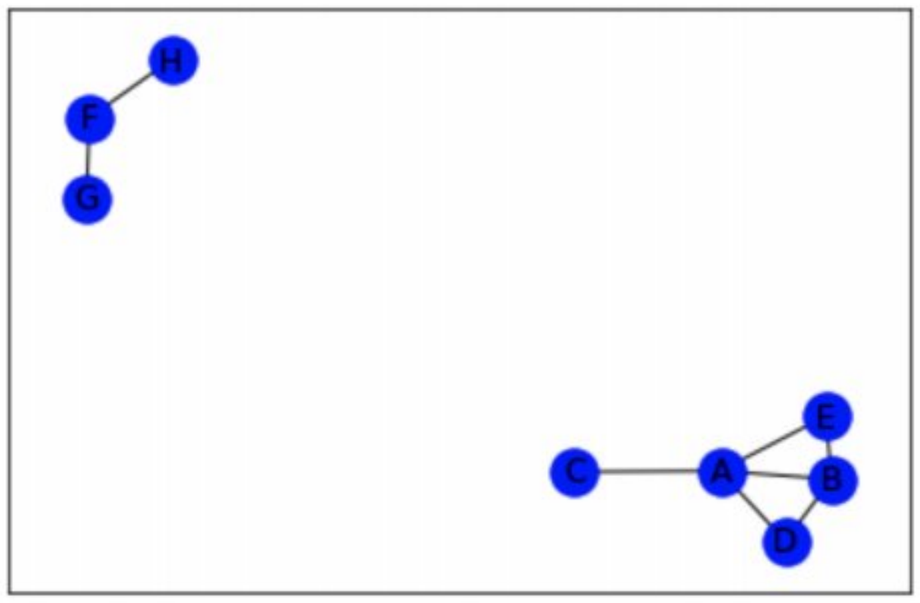
Voici le graphe de départ :
[('A', 'B'), ('B', 'C'), ('B', 'D'), ('D', 'E'), ('F', 'G'), ('G', 'H')]



Itération 1 :
[('B', 'A'), ('C', 'B'), ('C', 'A'), ('D', 'A'), ('D', 'B'), ('E', 'B'), ('E', 'D'), ('G', 'F'), ('H', 'F'), ('H', 'G')]
New Pair = 4

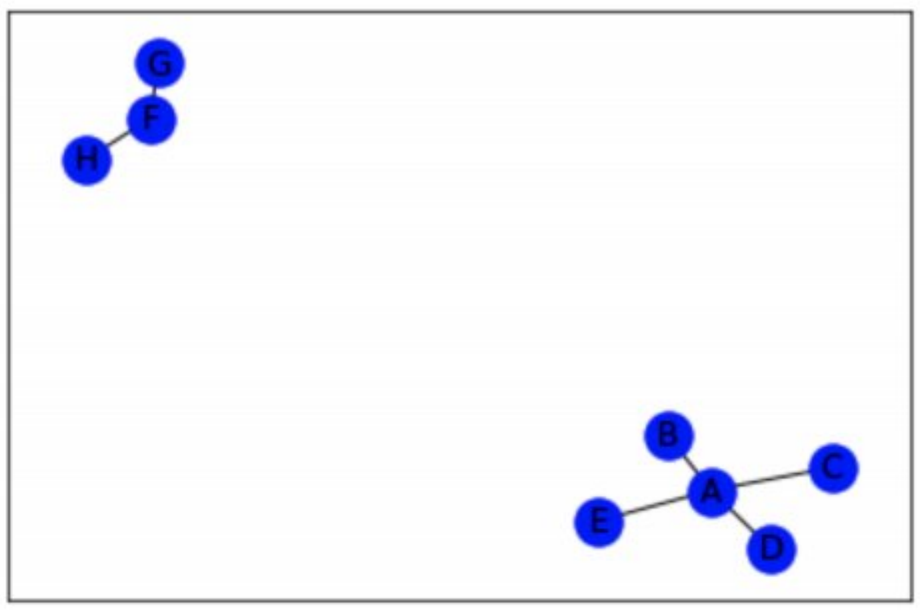


Itération 2 :
[('B', 'A'), ('C', 'A'), ('D', 'A'), ('D', 'B'), ('E', 'A'), ('E', 'B'), ('G', 'F'), ('H', 'F')]
New Pair = 9



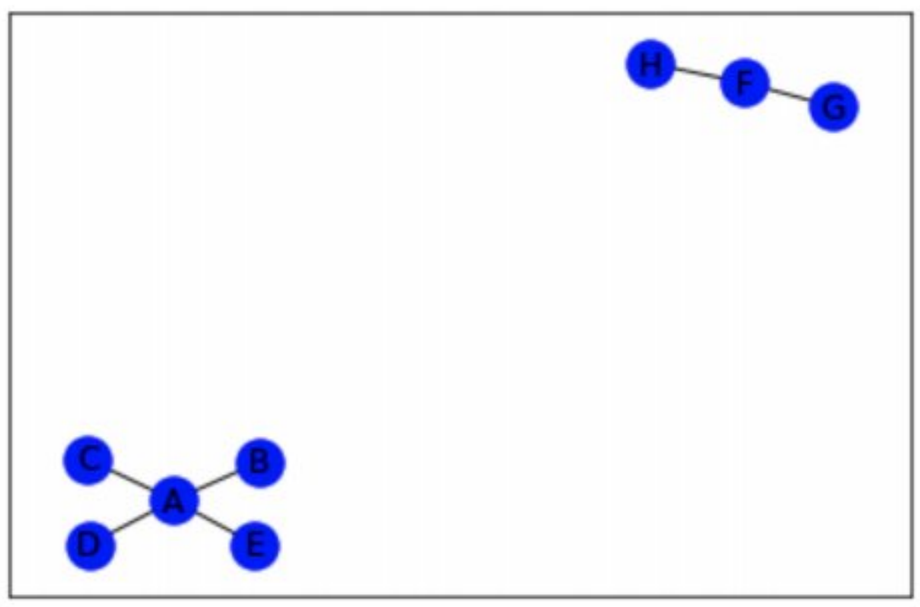
Itération 3 :

[('B', 'A'), ('C', 'A'), ('D', 'A'), ('E', 'A'), ('G', 'F'), ('H', 'F')]
New Pair = 4

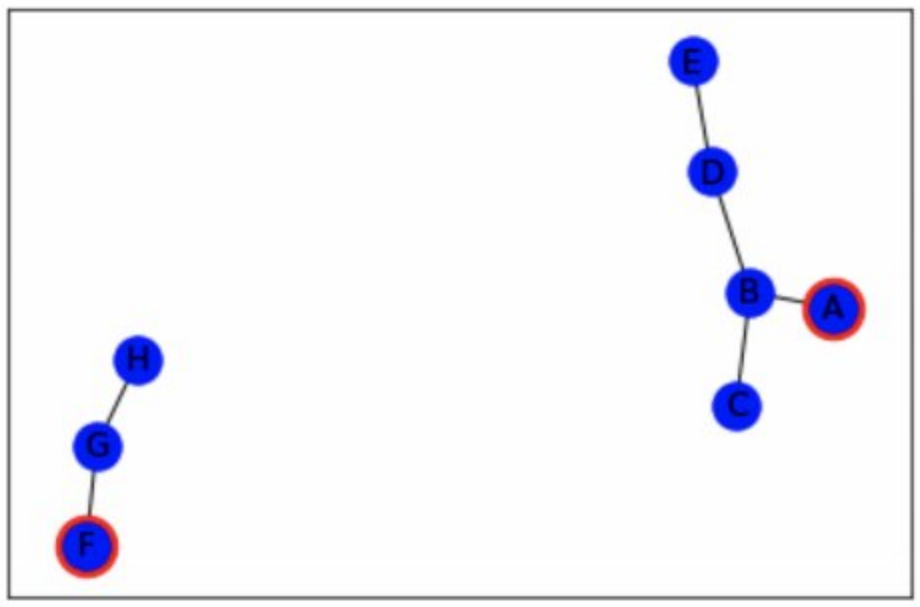


Itération 4 :

[('B', 'A'), ('C', 'A'), ('D', 'A'), ('E', 'A'), ('G', 'F'), ('H', 'F')]
New Pair = 0



Le nombre de composants connectés est : 2, et voici les composants :
[('A', ['B', 'D', 'E', 'C']), ('F', ['G', 'H'])]



À noter que dans notre Notebook Python nous avons utilisé trois fonctions intéressantes pour manipuler les graphes :

— La fonction 'draw' qui permet d'afficher un graphe à partir d'un RDD.

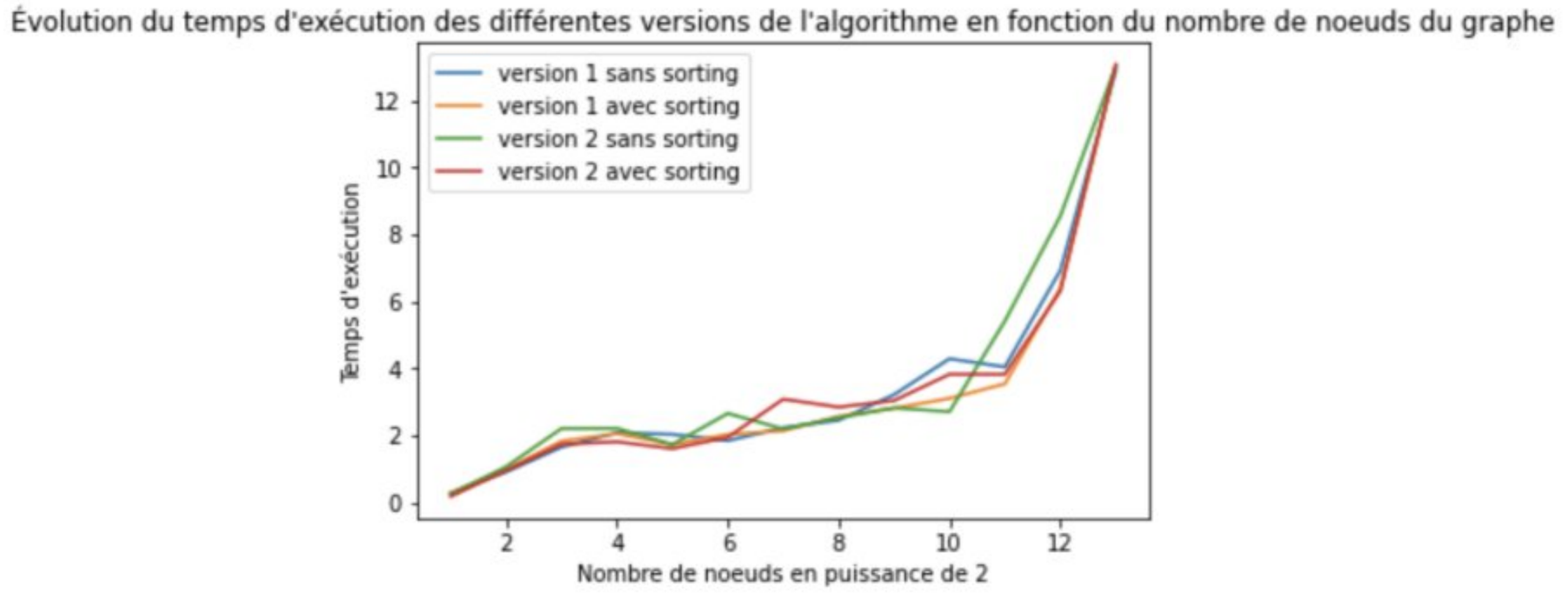
— La fonction 'generer_composantes_connexes' permet de générer des graphes aléatoires avec le nombre de composantes connexes souhaité, un nombre de nœuds par composantes connexes et enfin le nombre de voisins par nœud.

— La fonction 'afficher' affiche les informations retournées par la fonction 'algo_main' contenu dans le dictionnaire. Elle affiche le temps d'exécution total pour traiter le RDD et trouver les composantes connexes, elle affiche le nombre et les composantes connexes, si on choisit 'with_draw = True' alors le graphe du RDD est affiché en affichant les composantes connexes qui seront marqué par le noeud de valeur minimale marqué en rouge. Si on choisit 'with_draw_iteration = True', le graphe du RDD de chaque itération de la fonction 'algo_main' sera affiché.

Grâce à ceci, nous avons un module qui permet de générer des RDD de graphes aléatoirement et voir l'évolution des itérations pour trouver les composantes connexes et les afficher avec le temps d'exécution.

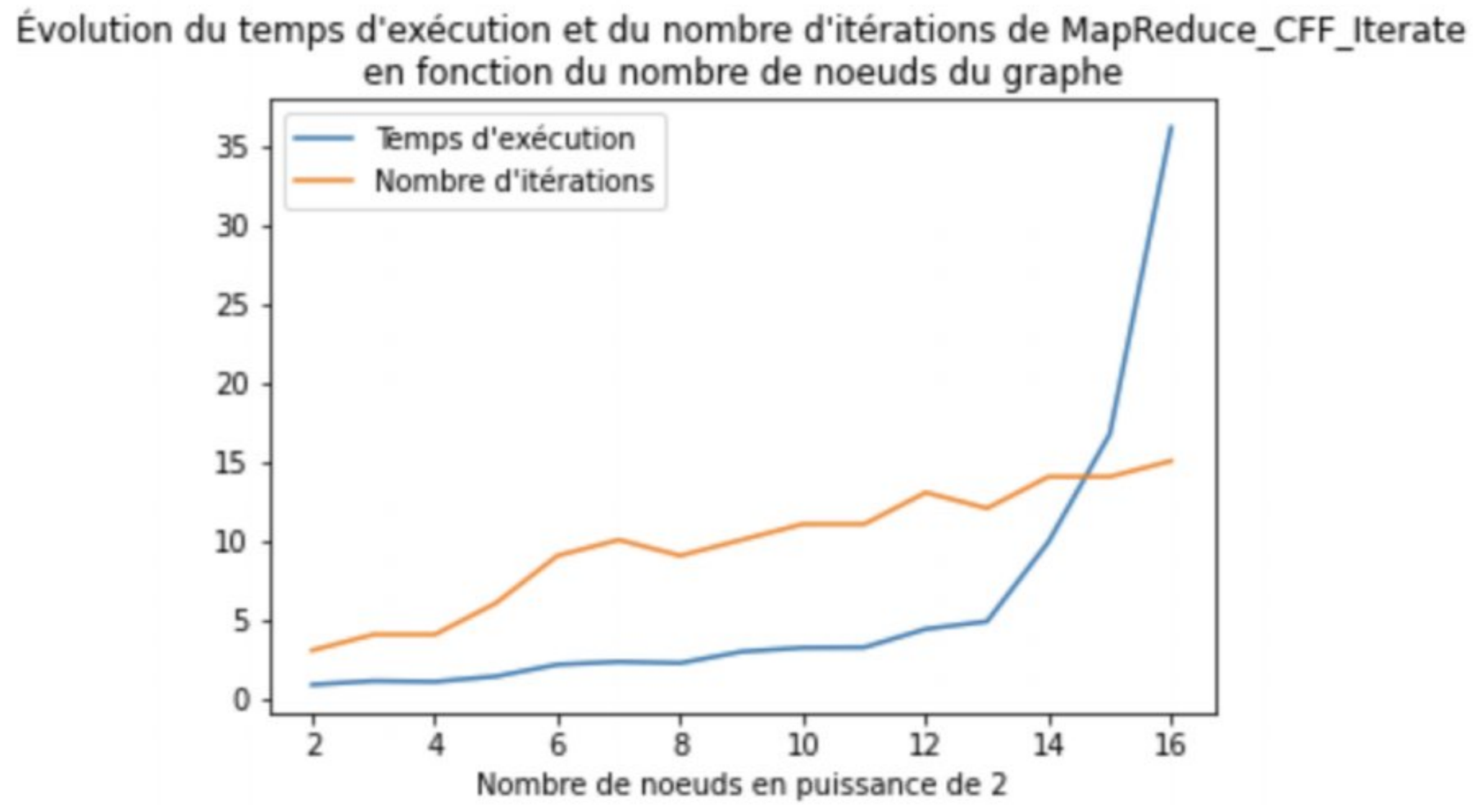
5 Analyses

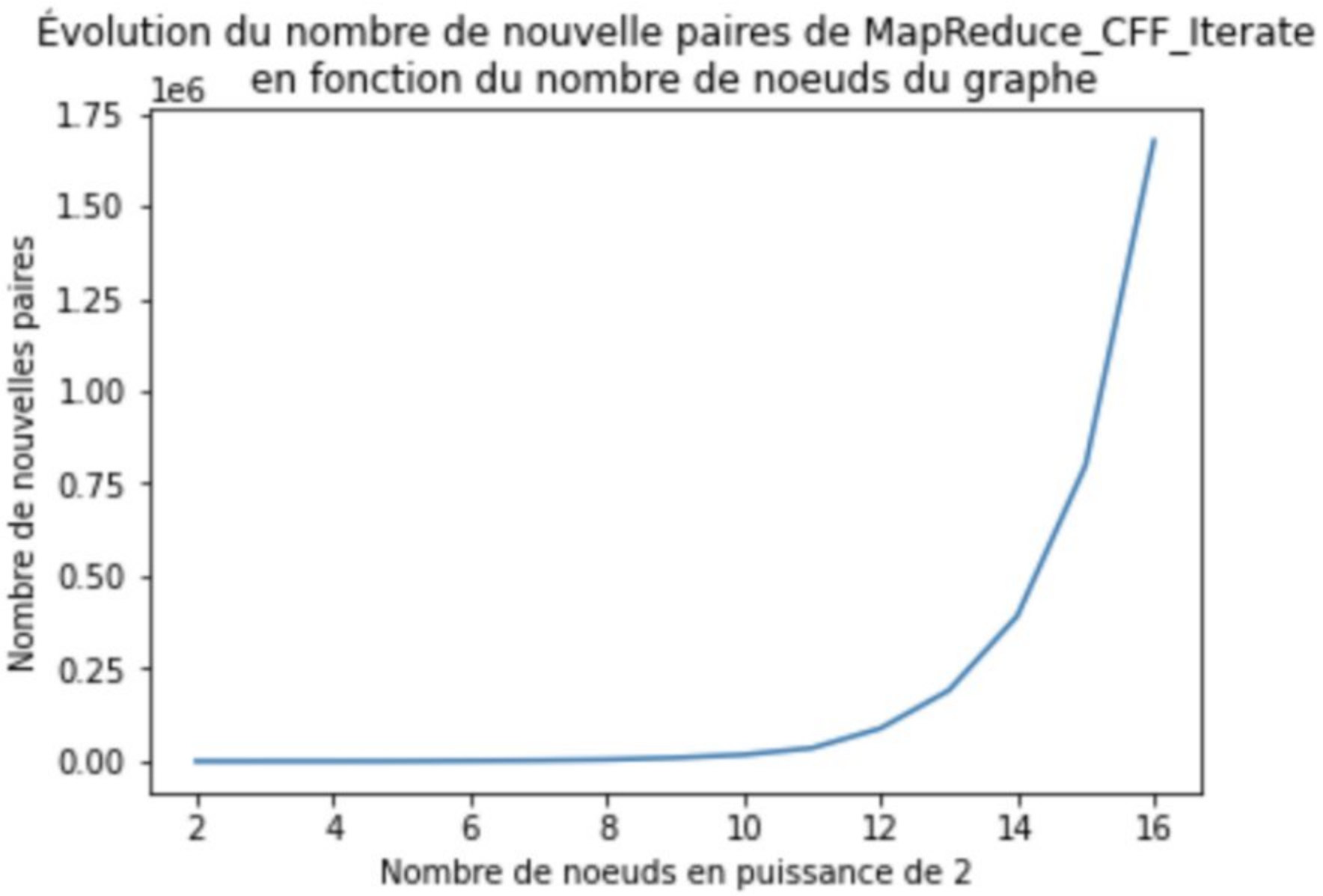
Dans cette partie, nous allons prendre des graphes avec un nombre de nœuds qui augmente, puis afficher l'évolution du temps d'exécution des différentes versions de l'algorithme en fonction du nombre de nœuds. Le nombre de nœuds augmente par 2 (x2).



On remarque que les différentes versions ont le même temps d'exécution, mais d'après le papier de recherche, pour un nombre de nœuds très grand (vers les 1 million), la méthode avec sorting a un temps d'exécution plus faible. Ici le nombre maximal de nœuds qu'on teste est 2 puissance 14 = 16 384. À noter que la version 1 est MapReduce_CFF_Iterate et la version 2 est MapReduce_CFF_Iterate_1.

Maintenant, nous allons prendre des graphes avec un nombre de nœuds qui augmente et choisir une version d'algo fixe, puis afficher l'évolution du temps d'exécution, le nombre d'itération ainsi que le nombre total de nouvelles paires trouvées en fonction du nombre de nœuds. Le nombre de nœuds augmente par 2 (x2).





6 Conclusion

En conclusion, ce projet nous a permis de renforcer nos connaissances intrinsèques sur MapReduce, ainsi que le principe des RDD. En effet, nous avons vu plusieurs méthodes pour implémenter différents algorithmes, ce qui nous a permis de comparer les performances et avantages/inconvénients de chacun. Grâce à cela, nous avons pu nous ouvrir aux différentes méthodes et approches pour implémenter du MapReduce.

L'exemple du papier de recherche nous a mis en situation concrète sur MapReduce. En effet, même si à la base les algorithmes sont très puissants et optimisés, lorsque l'on atteint des tailles de datasets très importantes (de l'ordre du TerraByte par exemple), les traitements sont beaucoup trop longs.

Implémenter des solutions MapReduce devient vital, et en répartissant les charges, avec du traitement parallélisé, on atteint des performances très intéressantes en réduisant considérablement le temps de traitement.