Rapport Big Data :
Finding connected components in graph

Min HUANG
Mounia ZRIGUI
Mohamed KOUHOU

April 2023

1 Description de la solution adoptée

Les graphes sont largement utilisés pour représenter des systèmes complexes tels que les réseaux sociaux, les réseaux de transport et les systèmes biologiques. L'un des problèmes fondamentaux de l'analyse des graphes est le calcul des composantes connexes. Une composante connexe est un groupe de sommets qui peuvent être reliés deux à deux par un chemin dans le graphe. Elles ont de nombreuses applications, notamment la détection de communautés, la segmentation de réseaux et la détection d'anomalies.

1.1 Modélisation du problème

Notre projet est basé sur l'article CCF: Fast and scalable connected component computation in MapReduce (2014) de H. Kardes, S. Agrawal, X. Wang et A. Sun.

Les auteurs affirment que les algorithmes existants pour le calcul des composantes connexes sont soit trop lents, soit trop gourmands en mémoire pour être utilisés sur de grands graphes. L'algorithme qu'ils proposent, CCF (Connected Component Finder), est conçu pour être efficace et évolutif pour les grands graphes.

Avant d'expliquer l'algorithme, essayons d'abord de comprendre la modélisation du problème à travers l'exemple de la figure 1.

Soit $G = (V, E)$ un graphe non orienté où
$V = \{V1, V2, V3, V4, V5, V6, V7, V8, V9, V10, V11, V12\}$ est l'ensemble des sommets et $E = \{E1, E2, E3, E4, E5, E6, E7, E8, E9, E10, E11\}$ est l'ensemble des arêtes. D'après le schéma, le graphe G a comme composantes connexes les 3 sous-graphes $C1 = \{V1, V2, V3, V4, V5, V6\}$, $C2 = \{V7, V8, V9\}$ et $C3 = \{V10, V11, V12\}$ puisqu'ils vérifient les conditions suivantes $C1 \cup C2 \cup C3 = G$ et $C1 \cap C2 \cap C3 = \emptyset$.

Le CCF prend en entrée la liste des couples (k, v) représentant les arêtes du graphe (arête sortant du $k^{\text{ème}}$ sommet et rentrant au $v^{\text{ème}}$ sommet) et on

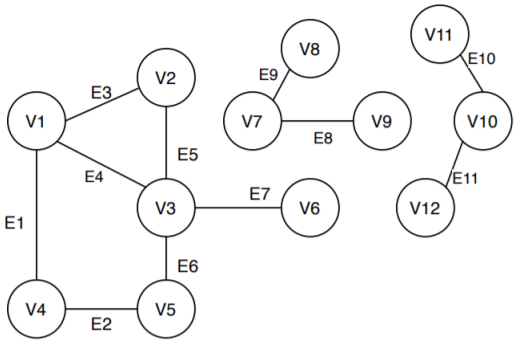

Figure 1: Exemple de graphe

souhaite obtenir en sortie pour chaque nœud son componentID. Ce dernier sera représenté par le numéro du plus petit nœud de la composante. Ainsi, pour l'exemple précédent, voici la sortie du module CCF
$(V1, V1)$,$(V2, V1)$,$(V3, V1)$,$(V4, V1)$,$(V5, V1)$,$(V6, V1)$,$(V7, V2)$,$(V8, V2)$,
$(V9, V2)$,$(V10, V3)$,$(V11, V3)$,$(V12, V3)$

1.2 Méthodologie et technologies adoptées

Les graphes devenant de plus en plus grands, un stockage distribué (comme HDFS) est nécessaire ainsi qu'un traitement distribué (des jobs MapReduce) pour la recherche des composantes connexes. Le paradigme MapReduce permet d'écrire des algorithmes parallélisables qui peuvent être exécutés sur plusieurs machines.

L'algorithme de CCF est un algorithme itératif conçu de manière à être facilement implémenté à travers une succession de tasks MapReduce. Dans ce projet, nous proposons une implémentation de ce module avec PySpark sur Databricks en deux variantes (CCF-Iterate et CCF-Iterate with secondary sorting). Chaque variante sera expliquée en détail et commentée par la suite.

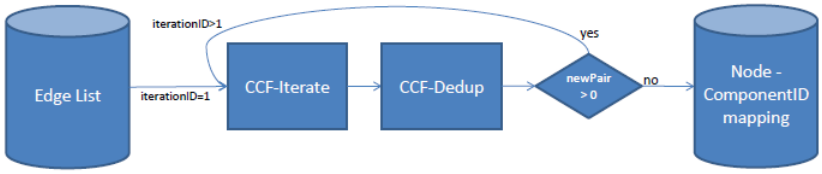

Figure 2: Module CCF

2 Implementation

2.1 Algorithmes conçus et commentaires globaux associés

2.1.1 Le job CFF-Iterate

Ce job a pour objectif final de créer un mapping entre chaque nœud et le componentID qui lui correspond. Il est exécuté en une succession d'itérations jusqu'à ce que tous les composants soient trouvés. Il est composé de deux phases :

- La phase **map** qui extrait les liens entre les nœuds en retournant tous les pairs clé-valeur et valeur-clé (pour garantir d'avoir tous les liens entre les nœuds dans les deux sens).
- La phase **reduce** traite les pairs clé-valeur retournés par le map en les regroupant par clé et en puis en déterminant la nouvelle clé à partir du minimum des valeurs associées à chaque clé.

```
CCF-Iterate
map(key, value)
    emit(key, value)
    emit(value, key)

reduce(key, < iterable > values)
    min ← key
    for each (value ∈ values)
        if(value < min)
            min ← value
        valueList.add(value)
    if(min < key)
        emit(key, min)
        for each (value ∈ valueList)
            if(min ≠ value)
                Counter.NewPair.increment(1)
                emit(value, min)
```

Figure 3: Pseudo-code du job CCF-Iterate

Plus concrètement, un `reduceByKey()` est effectué pour regrouper les arêtes par clé et pour obtenir la liste de toutes les arêtes liées à cette clé. Ensuite, on cherche la valeur minimale de la liste des valeurs, et on retourne le pair (clé, minValeur) où la clé est supérieure à cette valeur minimale. Finalement, pour chaque élément de la liste des valeurs, on retourne le couple (valeur, minValeur).

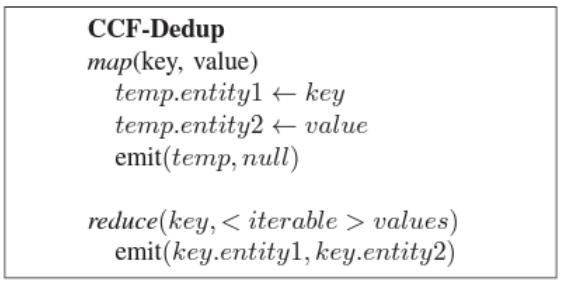

Figure 4: Pseudo-code de CCF-Dedup

Le job CFF-Iterate calcule les composantes connectées à partir d'un graphe données en entrée à travers des itération MapReduce, mais la sortie peut contenir des doublons, ce qui peut entraîner des erreurs ou des incohérences dans le traitement en aval. Le job CCF-Dedup résout ce problème en supprimant les éléments dupliqués de la sortie du job CCF-Iterate.

2.1.2 L'algorithme CCF-Iterate avec tri secondaire

C'est un algorithme amélioré. En utilisant cette méthode, nous n'avons plus besoin de deux itérations. Ce qui peut simplifier l' algorithme et améliorer la performance.

D'abord, pour l'étape MAP, c'est le même avec le premier CCF-Iterate. On fusionne le clé-valeur et valeur-clé pour propager l'information entre les différentes partitions de la RDD.

Ensuite, en Reduce, nous avons qu'une fois d'itération, grâce au tri secondaire, ce qui consiste à trier les valeurs associées à chaque clé par ordre croissant. Le tri permet de réduire le nombre de comparaisons nécessaires pour créer les nouveaux tuples, ce qui améliore les performances de l'algorithme.

Après le tri, nous filtrons sur les (clé, valeur) avec la valeur de clé supérieur à la valeur minimale et puis retour des nouveaux tuples en associant la plus petite valeur avec toutes les autres valeurs de cette même clé directement, il n'y a pas d'autre comparaison supplémentaire.

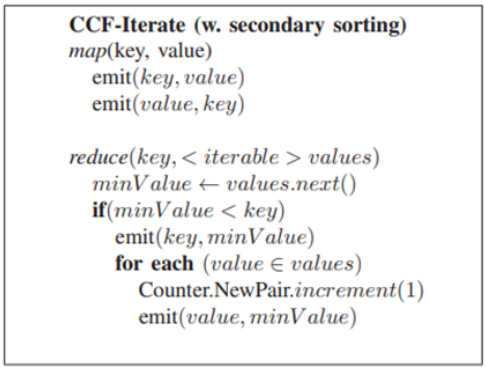

Figure 5: L'algorithme CCF-Iterate avec tri secondaire

En résumé, l'algorithme CCF-Iterate secondary sorting fonctionne en itérant sur les paires clé-valeurs d'une RDD, en choisissant les paires avec le clé est plus que la valeur minimale. Puis, retourner les tuples des clé-valeurs minimale et de nouveaux tuples en associant les valeurs les plus petites de chaque partition avec toutes les autres valeurs de cette même partition, en utilisant une technique de tri secondaire pour améliorer les performances. Ce processus est répété jusqu'à le `NewPair` (nombre de nouveaux tuples retourné) égale à 0.

2.2 Commentaires aux principaux fragments de code

2.2.1 CCF-Iterate

■ **Étape du Map :**

La fonction `map_ccf(r)` est utilisée pour mapper chaque pair (clé, valeur) dr RDD `r` en un pair (valeur, clé). Puis, la deuxième étape consiste à fusionner les RDD `r` et `r1` en un seul RDD en utilisant l'opération `union`. Cette opération est réalisée pour garantir d'avoir les liens entre chaque deux sommets dans les deux sens. Ainsi nous aurons un mapping exhaustif pour tous les noeuds du graphe que l'on aura en entrée.

```
def map_ccf(r):
    r1 = r.map(lambda x:(int(x[1]),int(x[0])))
    return r.union(r1)
```

Figure 6: Fonction map

■ **Étape du Reduce :**

Nous avons défini la fonction `reduce_ccf_iterate(r_map)` qui est l'implémentation de la tâche Reduce du job CFF-Iterate . Elle prend comme entrée la sortie de la fonction `map_ccf(r)`, à savoir les pairs (k,v) représentant les liens entre les sommets du graphe. A l'aide de la fonction `reduceByKey()`, ces pairs sont regroupés par clé et pour chaque clé seule la valeur minimale est retenue parmi toutes les valeurs associées. A ce stade, on obtient un RDD `v_min_1` de pairs (k,v) avec k la clé et v le minimum des valeurs. De plus, la valeur minimale de `v_min_1` est la min de la liste de valeurs, mais il existe le cas où la clé est plus petite que la min de la liste de valeurs. Donc on crée un le RDD `v_min`, quand la clé est plus petite que la valeur min, on retourne ¡clé,clé¿, sinon on garde l'ancien résultat.

```
def reduce_ccf_iterate(r_map):
    # Chercher la valeur minimale
    v_min_1 = r_map.reduceByKey(lambda x,y:min(x,y))
    v_min = v_min_1 \
        .map(lambda x: (x[0],x[0]) if x[0]<x[1] else (x[0],x[1]))
    # Premier emit : if(min < key) => emit(key, min)
    emit_1 = v_min.filter(lambda x: x[0]>x[1])

    # Deuxième emit : if(min != value) => emit(value, min)
    min_value = r_map.join(v_min) \
        .filter(lambda x:x[0]>x[1][1]) \
        .filter(lambda x: x[1][0] != x[1][1])
    NewPair = min_value.count()
    emit_2 = min_value.map(lambda x:(x[1][0],x[1][1]))
    return NewPair,emit_1.union(emit_2)
```

Figure 7: Code CCF-Iterate

Ensuite, à partir du RDD précédent on crée un autre RDD `emit_1` dans lequel on ne garde que le couple ¡clé, minValeur¿ dont la clé est plus grande que la valeur, cela étant fait en passant par un RDD intermédiaire `v_min`.

`emit_2` contient les autres couples ¡valeur, minValeur¿ qui respectent la même contrainte que `emit_1`. Cela est possible grâce à la jointure des deux RDD `r_map` (le RDD de départ pour récupérer les valeurs) et `v_min` (pour récupérer la valeur minimale).

La variable `NewPair` est un compteur qui détermine si l'on a trouvé toutes les composantes ou non. S'il est à 0, cela veut dire que toute les composantes ont été trouvées et la condition d'arrêt de la boucle de recherche est alors remplie.

- **Boucle principale :**

```
import time
def combine_iterate_1(r):
    newPair = 1
    newPairs = []
    i = 0
    iters = []
    T_start = time.time()
    iters_times = []
    while newPair > 0:
        T_start_iter = time.time()
        r_map = map_ccf(r)
        newPair, r_reduce = reduce_ccf_iterate(r_map)
        r = CCF_Dedup(r_reduce)
        T_end_iter = time.time()
        iter_time = T_end_iter - T_start_iter
        iters_times.append(iter_time)
        newPairs.append(newPair)
        iters.append(i)
        i = i+1
        print('iterate:%d, newPair:%d, Iteration time:%s s' \
            %((i,newPair,iter_time)))
    else :
        T_end = time.time()
        print('Total execute time:%s s' % ((T_end - T_start)))
        return iters_times, iters, newPairs, r
```

Figure 8: Boucle principale

Dans la boucle principale, on fait une itération des fonctions map, reduce et dédoublonnage. Ce processus est répété jusqu'à ce que la variable `NewPair` (nombre de nouveaux tuples retourné) soit égale à 0. De plus, on sauvegarde des listes de nombre d'itération, le nombre des nouveaux tuples dans chaque itération, ainsi que la liste des durées d'exécution des itérations.

2.2.2 CCF-Iterate avec tri secondaire

Ce job utilise la même fonction map que le job CCF-Iterate.

```
def cff_second_iterate(r_map):

  # List de valeurs triée et emit minV alue < key
  min_key = r_map.groupByKey() \
    .mapValues(list) \
    .map(lambda x: (x[0],sorted(x[1]))) \
    .filter(lambda x:x[0]>x[1][0]) \
    .partitionBy(100).cache()
  min_key_emit = min_key.map(lambda x:(x[0],x[1][0]))
  # Emit les tuple (value, minV alue)
  value_min_emit = min_key.flatMap(lambda x: \
    [(value, x[1][0]) for value in x[1][1:]])

  # Count the new pair
  newPair = value_min_emit.count()
  return newPair, min_key_emit.union(value_min_emit)
```

Figure 9: CCF-Iterate avec tri secondaire

Dans l'étape Reduce, on récupère la liste de valeur de chaque clé et trie chaque liste de valeurs associées par ordre croissant et renvoie des tuples (clé, liste triée), puis garde seulement les tuples avec la valeur du clé supérieur à la valeur minimale de la liste des valeurs, donc le premier élément de la liste triée, et retourne les tuple clé-valeur minimale. Après, on retourne des nouveaux tuples, la plus petite valeur avec toutes les autres valeurs de la même clé. Ensuite on compte le nombre des nouveaux tuples comme `NewPair`.

Pour la boucle principale de cette variante, elle est similaire à la boucle principale de CCF-Iterate précédent, sauf qu'elle utilise la fonction `cff_second_iterate(r_map)` au lieu de la fonction `reduce_ccf_iterate(r_map)` pour effectuer les tasks Reduce sur les RDD et calculer les `newPairs`.

3 Analyse expérimentale

Pour évaluer la performance des algorithmes développés, nous avons utilisé les données **Google web graph**. Il s'agit d'un gros fichier txt qui contient les données d'un graphe dont nœuds représentent les pages web et les arêtes représentent les liens hypertextes entre ces pages. Ces données ont été publiées en 2002 par Google dans le cadre de la compétition Google Programming Contest.

```
rdd_txt = sc.textFile('web-Google.txt') \
    .map(lambda x: x.split('\t')) \
    .filter(lambda x: not x[0].startswith('#')) \
    .map(lambda x: (int(x[0]),int(x[1])))
```

Figure 10: Importation et nettoyage des données

Dans le code ci-dessus, on lit le fichier texte `web-Google.txt` à l'aide de la méthode `textFile()` de `SparkContext`. Le fichier contient des lignes avec des valeurs séparées par des tabulations. La méthode `map()` est alors appelée sur le RDD résultant pour transformer chaque ligne en une liste de chaînes de caractères divisées par "`\t`". Ensuite, on élimine les lignes de commentaire (qui commencent par un "#") avec la fonction `filter()`. Enfin, la méthode `map()` est à nouveau utilisée pour transformer chaque ligne restante en un tuple de deux entiers (pairs clé-valeur).

Initialement, nous avons essayé de faire tourner le code sur Google Colab, mais les ressources très limtées de ce dernier ont fait que le job CCF-Iterate tournait pendant très longtemps pour enfin s'arrêter brusquement car la limite des ressources a été atteinte. Nous avons alors basculé sur Databricks sur lequel nous avons créé un cluster Google Cloud ayant la configuration suivante :

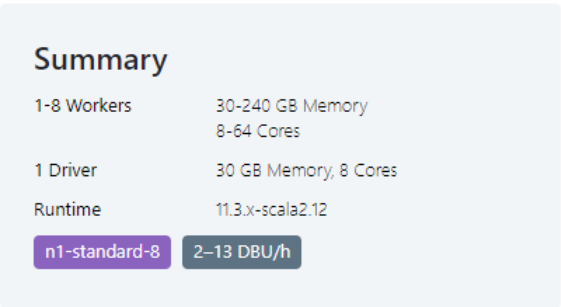

Figure 11: Configuration du cluster Google Cloud utilisé

Ce qui a permit d'accélérer considérablement le calcul grâce à l'augmentation du nombre des coeurs (meilleur parallélisme).

3.1 Résultats du CCF-Iterate

La complexité temporelle de CCF-Iterate sans tri secondaire est de $\mathcal{O}(A)$, où A est le nombre d'arêtes dans le graphe. En effet, à chaque itération de CCF-Iterate, l'algorithme traite chaque arête du graphe exactement une fois. Pour chaque arête, il vérifie si les sommets ont la même étiquette, et si ce n'est pas le cas, il met à jour les étiquettes et émet un nouvel enregistrement pour propager les étiquettes mises à jour. Par conséquent, la complexité temporelle de CCF-Iterate est proportionnelle au nombre d'arêtes dans le graphe d'entrée.

En utilisant le graphe web-Google, le CCF-Iterate a mis environ **5690 s** en total pour s'exécuter en **8** itération. Voici un graphe qui montre en détail le temps d'exécution pour chaque itération :

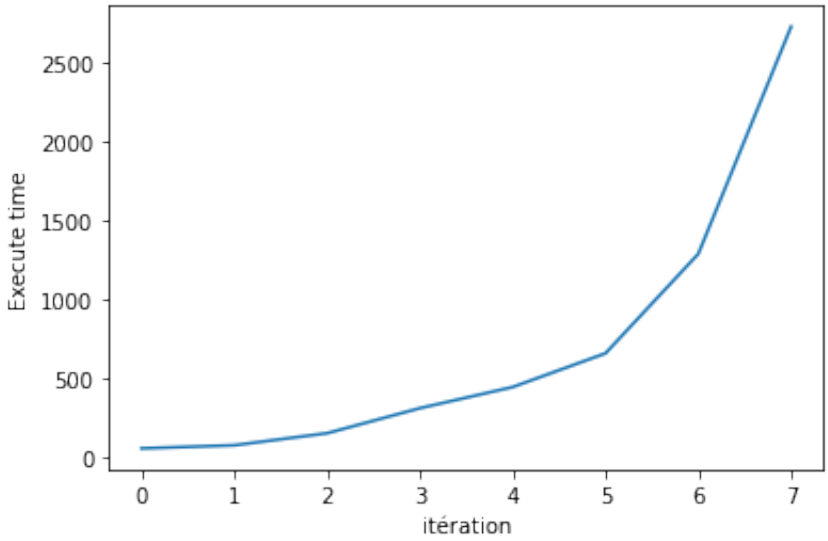

Figure 12: Temps d'exécution par itération pour CCF-Iterate

On constate que la pente de la courbe dans chaque itération est plus grande que dans celle qui précède. Cela veut dire que chaque itération prend plus de temps que la précédente. Cela est causé par l'accumulation d'enregistrements intermédiaires inutiles générés au cours du traitement.

Le graphe suivant suivant représente la variation du nombre de nouveaux pairs générés en fonction des itérations. La tendance globale de la courbe montre que le nombre de nouveaux pairs diminue pour converger vers 0, ce qui est naturel puisque cela indique que les composants connectés sont identifiés et étiquetés avec précision.

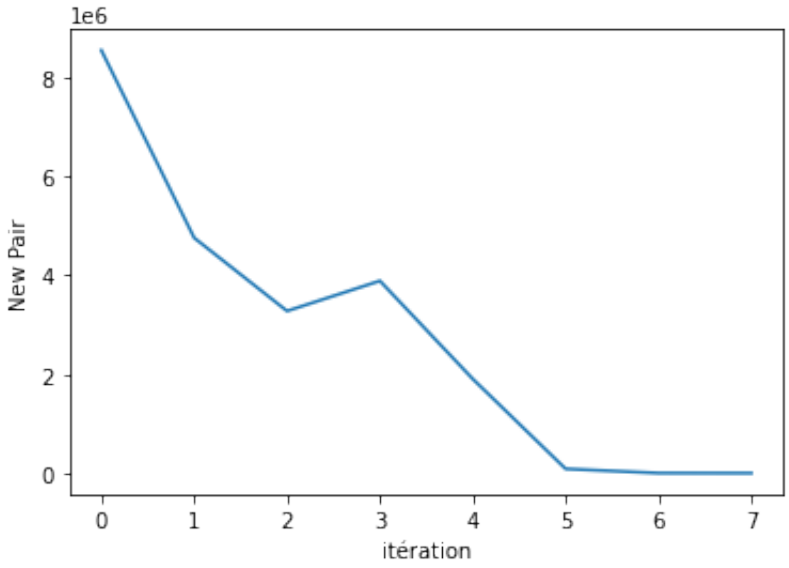

Figure 13: Nombre de nouveaux pairs par itération pour CCF-Iterate

La petite augmentation du nombre de nouveaux pairs entre l'itération 3 et 4 peut être expliquée par le fait qu'il y a des petits sous-graphes déconnectés ou des nœuds isolés dans le graphe d'entrée qui prennent plus de temps à converger vers leurs étiquettes finales.

3.2 Résultats du CCF-Iterate avec tri secondaire

CCF-Iterate avec tri secondaire est conçu pour fonctionner efficacement dans un environnement distribué. Cette méthode a une bonne évolutivité, qui sont des structures de données distribuées et tolérantes aux pannes. Puisque :

1. La fonction prend en entrée un RDD `r_map` qui représente un graphe sous la forme d'un ensemble de paires clé-valeur (nœud, voisins) où nœud est un sommet du graphe et voisins est une liste de ses voisins.

2. On a utilisé `groupByKey()` et `mapValues()` pour regrouper les valeurs par leurs clés correspondantes.

3. La fonction `map()` est utilisée pour trier les valeurs de chaque clé et émettre les tuples avec la valeur minimale inférieure à la clé.

4. La fonction `flatMap()`, qui émet les valeurs restantes pour chaque clé.

5. A la fin, la fonction `union()` combine le RDD au retourne.

Donc, dans les fonctions ci-dessus, la fonction `groupByKey()` et la fonction `mapValues()` peuvent être exécutées en parallèle sur plusieurs nœuds d'un cluster, ce qui permet un traitement efficace de grandes quantités de données. De même, la fonction `flatMap()` et la fonction `union()` peuvent être exécutées en parallèle sur plusieurs nœuds, améliorant encore l'évolutivité de l'algorithme.

De plus, l'algorithme CCF-Iterate avec tri secondaire a une faible complexité de calcul, ce qui peut adapter au traitement de graphes à grande échelle. L'algorithme a une complexité de $\mathcal{O}(A \log S)$, où A est le nombre d'arêtes dans le graphe et S est le nombre de sommets.

En utilisant la source de données web-Google dans le papier, le CCF-Iterate avec tri secondaire a mis 200.58132338523865 s en total pour s'exécuter. De plus, On a la relation entre le nombre de composants et le nombre d' itération comme suivant :

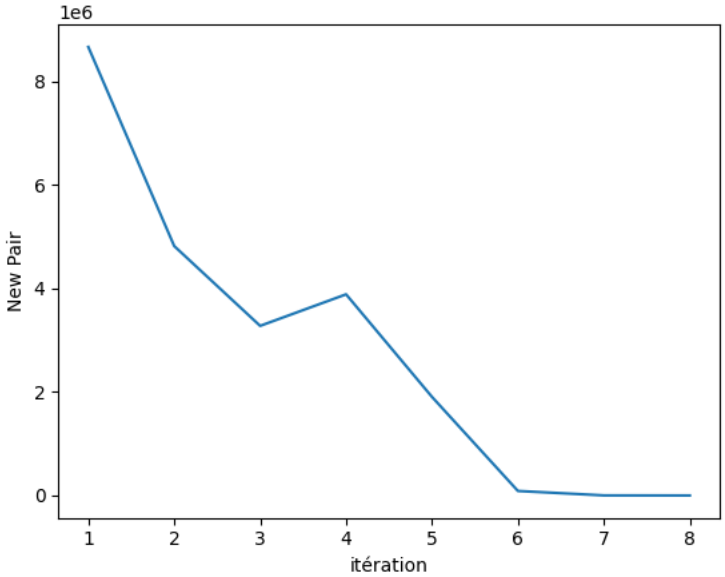

Figure 14: Nombre de nouveaux pairs par itération pour CCF-Iterate avec tri secondaire

De plus, le changement du temps de l'exécution de chaque itération est comme l'image au-dessous. Nous pouvons voir qu'à mesure que le nombre de boucles augmente, le taux de croissance du temps d'exécution devient de plus en plus petit. Car à mesure que le nombre de boucles augmente, nous traitons de moins en moins de nouvelles paires, comme le figure ci-dessus.

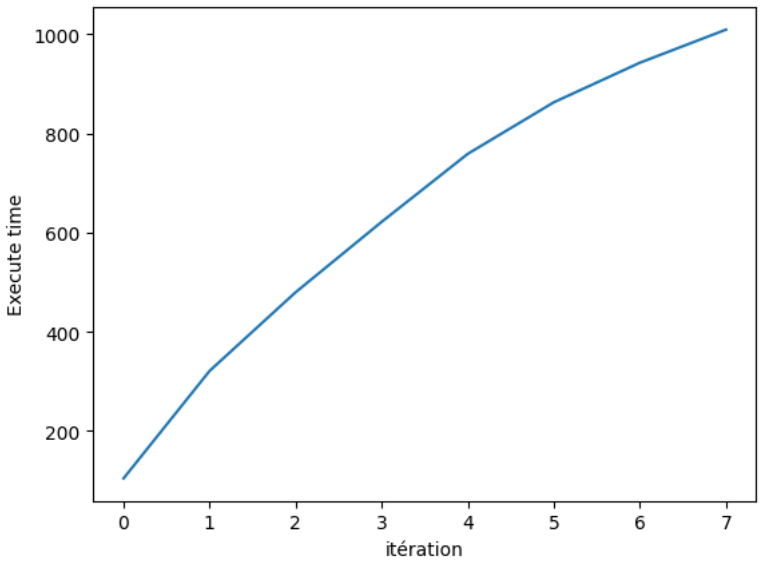

Figure 15: Temps d'exécution par itération pour CCF-Iterate avec tri secondaire

4 Avantages et inconvénients

4.1 CCF-Iterate

L'algorithme CCF-Iterate présente plusieurs avantages. En effet, il est simple et facile à mettre en œuvre. Il est aussi efficace en termes de nombre d'itérations nécessaires pour converger vers les composantes connectées finales (8 itérations pour Google Web Graph qui présente 875713 nœuds et 5105039 arêtes, ce qui est énorme!). Enfin, CCF-Iterate est un algorithme robuste qui permet de converger vers les composantes connectées à la fin des itérations.

Cependant, CCF-Iterate sans tri secondaire présente plusieurs inconvénients qui le rendent désagréable, notamment sa grande complexité spatiale ($\mathcal{O}(S)$ où S est le nombre de sommets) et temporelle ($\mathcal{O}(A)$ où A est le nombre d'arêtes), ce qui le rend peu efficace dans le cas de très grands graphes et peu évolutif (cf figure 12 où le taux d'augmentation du temps d'exécution est d'autant plus grand que l'on a de pairs en entrée).

4.2 CCF-Iterate avec tri secondaire

Cet algorithme présente de nombreux avantages. Tout d'abord, il est efficace. L'algorithme CCF-Iterate avec tri secondaire a une complexité de calcul relativement faible de $\mathcal{O}(E \log V)$ (comme indiqué précédemment), ce qui le rend efficace pour le traitement de grands graphes.

De plus, il est évolutif, il utilise des RDD distribués et tolérants aux pannes, ce qui permet une mise à l'échelle facile et une meilleure utilisation des ressources de calcul. De plus, comme il y a moins d'itérations, le code de la fonction est relativement simple et facile à comprendre. Au final, c'est une façon exacte, le tri secondaire garantit que toutes les paires de nœuds appartenant à la même communauté seront fusionnées en un seul composant lié.

Cependant, il y a encore des inconvénients à la méthode. La fonction nécessite un grand tri qui peut prendre beaucoup de temps et de mémoire pour les grands graphiques. De plus, l'algorithme nécessite des communications coûteuses entre les nœuds pour échanger les paires de nœuds nouvellement fusionnées. Ensuite, l'algorithme peut être sensible aux données déséquilibrées, car certaines clés peuvent avoir beaucoup plus de valeurs que d'autres, ce qui peut ralentir le traitement.

5 Code

```
!pip install pyspark
from pyspark import SparkConf
from pyspark.context import SparkContext
sc = SparkContext.getOrCreate(SparkConf().setMaster("local[*]"))


# Data Load, Data clean
rdd_txt = sc.textFile('web-Google.txt').map(lambda x: x.split('\t')).\
 filter(lambda x: not x[0].startswith('#')).\
 map(lambda x: (int(x[0]),int(x[1])))


# MAP
def map_ccf(r):
 r1 = r.map(lambda x:(int(x[1]),int(x[0])))
 return r.union(r1)


# CCF-Iterate-Reduce
def reduce_ccf_iterate(r_map):
   # Chercher la valeur minimale
   v_min_1 = r_map.reduceByKey(lambda x,y:min(x,y)).\
   partitionBy(100).cache()
   v_min = v_min_1.map(lambda x: (x[0],x[0]) if x[0]<x[1] else (x[0],x[1]))
   # Premier emit : if(min < key) => emit(key, min)
   emit_1 = v_min.filter(lambda x: x[0]>x[1])

   # Deuxième emit : if(min != value) => emit(value, min)
   min_value = r_map.join(v_min).filter(lambda x:x[0]>x[1][1]).\
```

```
   filter(lambda x: x[1][0] != x[1][1])
   NewPair = min_value.count()
   emit_2 = min_value.map(lambda x:(x[1][0],x[1][1]))
   return NewPair,emit_1.union(emit_2)


# CCF-Iterate (w. secondary sorting) (old)
def cff_second_iterate(r_map):

 # List de valeurs triée et emit minV alue < key
 min_key = r_map.groupByKey().mapValues(list).\
 map(lambda x: (x[0],sorted(x[1]))).filter(lambda x:x[0]>x[1][0])
 min_key_emit = min_key.map(lambda x:(x[0],x[1][0]))
 # Emit les tuple (value, minV alue)
 value_min_emit = min_key.\
 flatMap(lambda x: [(value, x[1][0]) for value in x[1][1:]])
 # Count the new pair
 newPair = value_min_emit.count()
 return newPair, min_key_emit.union(value_min_emit)


# Etape CCF_Dedup
def CCF_Dedup(cff_r):
 map_r_cff = cff_r.map(lambda x : ((x[0],x[1]),0))
 dedup = map_r_cff.reduceByKey(lambda x,y:x+y).\
 map(lambda x: (x[0][0],x[0][1]))
 return dedup


# Main loop CCF-Iterate
import time
def combine_iterate_1(r):
   newPair = 1
   newPairs = []
   i = 0
   iters = []
   T_start = time.time()
   iters_times = []
   while newPair > 0:
       T_start_iter = time.time()
       r_map = map_ccf(r)
       newPair, r_reduce = reduce_ccf_iterate(r_map)
       r = CCF_Dedup(r_reduce)
       T_end_iter = time.time()
       iter_time = T_end_iter - T_start_iter
       iters_times.append(iter_time)
```

```
        newPairs.append(newPair)
        iters.append(i)
        i = i+1
        print('iterate: %d, newPair: %d, Iteration time: %s s'
        %((i,newPair,iter_time)))
    else :
        T_end = time.time()
        print('Total execute time:%s s' % ((T_end - T_start)))
        return iters_times, iters, newPairs, r



# Main loop CCF-Iterate (w. secondary sorting)
def combine_iterate_2(r):
    NewPair = 1
    i = 0
    NewPair_list = []
    iterat_list = []
    T = []
    T1 = time.time()
    while NewPair > 0:
        r_map = map_ccf(r)
        NewPair, r_reduce = cff_second_iterate(r_map)
        r = CCF_Dedup(r_reduce)
        T2 = time.time()
        T.append(T2 - T1)
        iterat_list.append(i)
        NewPair_list.append(NewPair)
        i = i+1
        print('Execute time:%s s' % ((T2 - T1)))
        print('iterate : %d,NewPair : %d' %((i,NewPair)))
    else :
        T2 = time.time()
        print('Total execute time:%s s' % ((T2 - T1)))
        return T,iterat_list,NewPair_list,r


# Visualisation
import matplotlib.pyplot as plt
plt.plot(iters,newPairs)
plt.ylabel("New Pair")
plt.xlabel("itération")

plt.plot(iters,iters_times)
plt.ylabel("Execute time")
```

```
plt.xlabel("itération")
```