Détection de composantes connexes dans des graphes de grande taille avec Apache Spark

Projet de Systèmes et Paradigmes Big Data

Master 2 IASD

Adam Hannachi

Année universitaire 2025–2026

Université Paris Dauphine – PSL

1 Introduction

Le problème de la détection des composantes connexes dans les graphes est fondamental dans de nombreux domaines d'application tels que l'analyse des réseaux sociaux, le data mining ou encore le rapprochement de données. Lorsque les graphes deviennent très volumineux, les algorithmes séquentiels classiques ne sont plus adaptés et il devient nécessaire de recourir à des approches distribuées.

Dans ce projet, nous étudions et implémentons un algorithme scalable de calcul des composantes connexes basé sur l'approche *Connected Component Finder (CCF)*, initialement proposée dans un contexte MapReduce. L'algorithme est implémenté à l'aide d'Apache Spark et évalué expérimentalement sur des graphes de tailles croissantes.

2 Description de la solution adoptée

La solution adoptée pour résoudre le problème de la détection des composantes connexes dans un graphe repose sur l'algorithme *Connected Component Finder (CCF)*, tel que décrit par Kardes et al. Cet algorithme a été initialement conçu pour le modèle MapReduce et vise à traiter efficacement des graphes de très grande taille dans un environnement distribué.

Le problème consiste, à partir d'un graphe non orienté représenté sous forme de liste d'arêtes, à identifier l'ensemble des composantes connexes, c'est-à-dire les sous-graphes maximaux dans lesquels tout couple de sommets est relié par un chemin. La solution retenue repose sur une représentation simple et déterministe des composantes connexes : chaque composante est identifiée par le plus petit identifiant de nœud qu'elle contient.

L'algorithme fonctionne de manière itérative. À partir de la liste d'arêtes initiale, il maintient une correspondance entre chaque nœud du graphe et un identifiant de composante. À chaque itération, chaque nœud compare son identifiant courant avec ceux de ses voisins directs et adopte, le cas échéant, un identifiant plus petit. Ce mécanisme de propagation locale permet de diffuser progressivement l'identifiant minimal à l'ensemble des nœuds appartenant à une même composante connexe.

La solution est structurée autour de deux traitements principaux exécutés de manière répétée. Le premier traitement, appelé *CCF-Iterate*, est chargé de propager les identifiants de composantes à partir des relations de voisinage entre les nœuds. Le second traitement, *CCF-Dedup*, permet d'éliminer les associations redondantes générées lors de la phase de propagation afin de limiter le volume de données échangées entre les itérations et d'améliorer l'efficacité globale du calcul.

Le processus itératif se poursuit jusqu'à ce qu'aucune nouvelle association nœud–composante ne soit produite lors d'une itération complète. Cette condition d'arrêt garantit que tous les nœuds ont atteint l'identifiant minimal de leur composante connexe et que le calcul des composantes est terminé.

Dans le cadre de ce projet, l'algorithme CCF est implémenté en utilisant Apache Spark et l'API RDD. Deux implémentations sont réalisées : une implémentation de base fidèle à l'algorithme MapReduce original et une implémentation optimisée visant à améliorer l'efficacité mémoire et à réduire les coûts de communication. Ces deux versions sont ensuite comparées expérimentalement sur des graphes de tailles croissantes afin d'étudier leur comportement et leur passage à l'échelle.

3 Algorithmes conçus et description globale

3.1 Algorithme CCF basique

Le premier algorithme implémenté dans ce projet correspond à la version basique du *Connected Component Finder (CCF)* décrite dans l'article de référence. Le graphe d'entrée est représenté sous forme de liste d'arêtes non orientées. Chaque composante connexe est identifiée par le plus petit identifiant de nœud appartenant à cette composante.

L'algorithme repose sur une stratégie de propagation itérative. À chaque itération, chaque nœud examine les identifiants associés à ses voisins directs et met à jour son propre identifiant de composante lorsqu'un identifiant plus petit est détecté. Ce processus est répété jusqu'à convergence, c'est-à-dire lorsque plus aucune nouvelle association nœud–composante n'est générée.

L'algorithme peut être résumé comme suit :

- le graphe est d'abord symétrisé afin que chaque arête non orientée (u, v) soit représentée par les couples (u, v) et (v, u) ;
- pour chaque nœud, l'identifiant minimal parmi le nœud lui-même et ses voisins est calculé ;
- cet identifiant minimal est ensuite propagé au nœud et à ses voisins ;
- les paires dupliquées sont éliminées afin de limiter les informations redondantes ;
- le processus est répété jusqu'à ce qu'aucun changement ne soit observé.

Cette approche garantit la correction de l'algorithme tout en restant simple et bien adaptée à un modèle d'exécution distribué basé sur les RDD de Spark.

3.2 Algorithme CCF optimisé

Une seconde version optimisée de l'algorithme est également implémentée afin d'améliorer la scalabilité et l'efficacité d'exécution. Bien que la logique algorithmique reste identique à celle de la version basique, plusieurs optimisations sont introduites.

Tout d'abord, les RDD intermédiaires sont mis en cache afin d'éviter des recomputations inutiles entre les itérations successives. Cette optimisation est particulièrement importante puisque les mêmes ensembles de données sont réutilisés plusieurs fois au cours du processus itératif. Ensuite, les shuffles inutiles sont réduits en limitant le nombre de transformations appliquées à chaque itération et en contrôlant soigneusement les opérations de déduplication.

Ces optimisations deviennent particulièrement bénéfiques lors du traitement de graphes de grande taille, pour lesquels l'utilisation mémoire et les coûts de communication jouent un rôle déterminant. La version optimisée permet ainsi d'améliorer l'efficacité globale tout en préservant les propriétés de correction et de convergence de l'algorithme original.

3.3 Commentaires sur les principaux fragments de code

Cette sous-section décrit les principaux éléments de l'implémentation PySpark et leur lien avec la conception algorithmique.

La symétrisation du graphe consiste à transformer la liste d'arêtes initiale de manière à représenter chaque arête non orientée dans les deux sens. Cette étape est essentielle pour garantir une propagation correcte de l'information de voisinage au cours des itérations.

La propagation des identifiants de composantes constitue le cœur de l'algorithme. Elle est implémentée à l'aide d'opérations de regroupement qui collectent l'ensemble des voisins d'un nœud donné. Pour chaque groupe, l'identifiant minimal est calculé puis propagé au nœud et à ses voisins. Cette opération correspond directement à l'étape *CCF-Iterate* décrite dans l'algorithme original.

Après chaque itération, les paires nœud–composante dupliquées sont supprimées à l'aide d'une opération `distinct`. Cette phase de déduplication correspond à l'étape *CCF-Dedup* et permet de limiter la croissance des données ainsi que les coûts de communication lors des itérations suivantes.

L'algorithme est exécuté de manière itérative jusqu'à convergence. Après chaque itération, les nouvelles associations générées sont comparées à celles de l'itération précédente. En l'absence de différences, le calcul est considéré comme terminé.

Dans la version optimisée, la mise en cache des RDD fréquemment réutilisés permet de réduire les coûts de recomputation. Cette optimisation améliore les performances, en particulier lorsque la taille des graphes augmente, et met en évidence l'importance de la gestion de la mémoire dans les applications Spark itératives.

4 Analyse expérimentale et scalabilité

Les expériences ont été réalisées en mode local sur une seule machine en utilisant Apache Spark et une implémentation PySpark. Trois graphes synthétiques de tailles croissantes (`small`, `medium`, `large`) ont été utilisés afin d'observer l'évolution du temps d'exécution en fonction de la taille des données.

Pour chaque graphe, les deux versions de l'algorithme ont été exécutées, à savoir la version basique (`ccf_basic.py`) et la version optimisée (`ccf_optimized.py`) intégrant notamment la mise en cache des RDD. Le temps mesuré correspond au temps total d'exécution de l'algorithme jusqu'à convergence.

Les temps d'exécution observés sont présentés dans le Tableau 1.

Graphe	**CCF basique (s)**	**CCF optimisé (s)**
small	2.99	2.27
medium	1.70	2.21
large	2.22	3.01

Table 1: Comparaison des temps d'exécution entre l'implémentation basique et l'implémentation optimisée

Sur le graphe `small`, la version optimisée est plus rapide que la version basique. Ce comportement est cohérent avec l'objectif de l'optimisation, la mise en cache permettant de limiter les recalculs lors des itérations successives lorsque les données tiennent aisément en mémoire.

En revanche, sur les graphes `medium` et `large`, la version optimisée devient plus lente. Cette observation ne remet pas en cause la validité de l'algorithme et s'explique par le contexte d'exécution local. Pour des volumes de données encore modérés, le coût dominant peut être

l'overhead de Spark, incluant la planification des tâches, la sérialisation Python/JVM, la gestion mémoire et les shuffles induits par les opérations `distinct` et `subtract`. Dans ce contexte, l'ajout d'optimisations telles que le caching peut introduire un surcoût supérieur aux gains attendus.

Ces résultats montrent ainsi que la performance et la scalabilité d'une optimisation dépendent fortement du contexte d'exécution. Sur un environnement distribué et pour des graphes de taille significativement plus importante, on s'attend généralement à ce que les optimisations, notamment la réduction des recomputations grâce au caching, deviennent plus avantageuses. L'expérience met en évidence un point central en Big Data : les optimisations ne sont pas universelles et doivent être évaluées dans le contexte cible.

5 Analyse critique : points forts et limites des algorithmes

L'analyse expérimentale met en évidence plusieurs points forts et limites des deux implémentations du Connected Component Finder, en particulier dans le contexte d'une exécution locale avec Apache Spark.

Un premier point fort commun aux deux algorithmes est leur correction et leur stabilité. Les versions basique et optimisée convergent systématiquement vers une solution correcte en un nombre limité d'itérations, ce qui valide la pertinence de l'implémentation par propagation itérative des identifiants de composantes.

La version basique présente l'avantage d'une grande simplicité d'exécution. Son comportement est prévisible et son overhead est limité, ce qui la rend efficace sur des graphes de taille modérée dans un environnement local. Elle constitue également une référence utile pour comprendre le fonctionnement du CCF et pour servir de base de comparaison.

La version optimisée illustre l'intérêt des mécanismes d'optimisation proposés par Spark, en particulier la persistance des données intermédiaires. Elle permet d'améliorer les performances sur les graphes de petite taille et met en évidence l'importance de la gestion mémoire dans les applications distribuées.

Les expériences soulignent néanmoins plusieurs limites. La version optimisée introduit un surcoût non négligeable lié à la gestion de la mémoire et aux opérations de shuffle. Dans un contexte local et pour des volumes de données modestes, ce surcoût peut dépasser les bénéfices attendus. Plus généralement, la scalabilité effective des optimisations dépend fortement de l'environnement d'exécution. Les bénéfices de la version optimisée sont davantage attendus sur un cluster distribué que sur une seule machine.

Enfin, l'utilisation de Spark pour des graphes de taille relativement réduite met en évidence un overhead structurel important, lié à l'initialisation des jobs, à la communication Python/JVM et à la planification des tâches, ce qui limite la pertinence des comparaisons de performance sur de petits jeux de données.

En synthèse, la version basique apparaît plus adaptée aux contextes simples et aux volumes modérés, tandis que la version optimisée est conceptuellement mieux préparée pour passer à l'échelle sur des environnements distribués et des graphes de grande taille. Cette étude illustre ainsi un principe fondamental des systèmes Big Data : une optimisation doit toujours être évaluée au regard du contexte cible et ne garantit pas systématiquement un gain de performance.

6 Conclusion

Ce projet a permis d'implémenter et d'analyser un algorithme distribué de détection de composantes connexes à l'aide d'Apache Spark. L'étude comparative entre une version basique et une version optimisée met en évidence les compromis entre simplicité, overhead et scalabilité. Les résultats expérimentaux soulignent l'importance d'une évaluation empirique des optimisations et montrent que leur efficacité dépend fortement du contexte d'exécution.

A Annexe : Code source

L'intégralité du code source développé dans le cadre de ce projet est disponible dans un dépôt GitHub public.

Ce dépôt contient les implémentations Python des algorithmes CCF, les scripts de génération des graphes de test, le script d'expérimentation utilisé pour l'analyse des performances ainsi que les instructions nécessaires à la reproduction des expériences.

Le dépôt GitHub est accessible à l'adresse suivante :

`https://github.com/admhan/connected-components-spark`

References

[1] H. Kardes, S. Agrawal, X. Wang, and A. Sun, *Fast and Scalable Connected Component Computation in MapReduce*, 2013.