**2020/2021**



**RAPPORT PROJET SYSTEMES PARADIGMES BIG DATA**

**SPARK**

**Membres du Groupe:**

Rowann Talon

Julien Vu

Chenge Yan

Raoul Yamdjieu Kamaha

Sommaire :

* Introduction
* Données fournies en entrée
* Implémentation de l’algorithme CCF-Iterate
* Résultats de l’algorithme CCF-Iterate
* Implémentation de l’algorithme CCF-Iterate avec tri secondaire
* Résultats de l’algorithme CCF-Iterate avec tri secondaire
* Comparaison entre les deux implémentations
* Conclusion

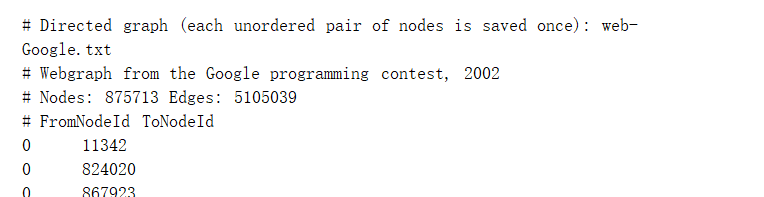
Introduction

L'énoncé du projet consistait à trouver les composants connectés de grands graphiques, et à implémenter les 2 variantes proposées dans PySpark au moyen de RDD.

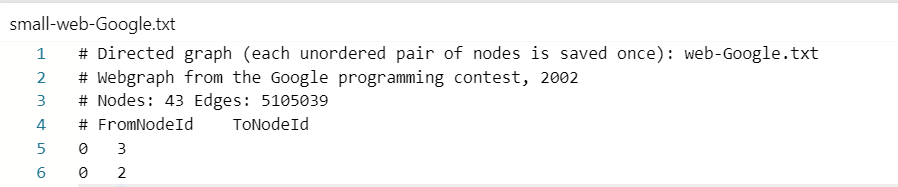
* Données fournies en entrée

Dans l’article CCF: Fast and Scalable Connected Component Computation in MapReduce, les auteurs affirment que cette méthode est réalisable sur un graphe de grande taille avec ~6B nœuds et ~92 arêtes. Dans le travail actuel, vu que nous ne pouvons pas accéder à de puissantes ressources de cluster, nous utilisons Databricks pour tester et revoir notre implémentation sur les graphiques de jouets.

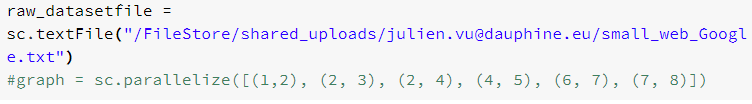
Nous avons choisi d’utiliser le graphique Web Google publie en 2002 par Google. Il y a 875 000 nœuds et 5,1 millions d’arêtes dans ce graphique. Les nœuds représentent des pages Web et les arêtes orientés représentent des liens hypertexte entre eux.



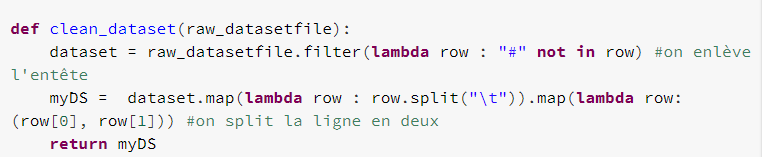
Etant donné que le graphique Web Google est de très grande taille, cela peut être entre 40 minutes et une heure pour obtenir les résultats du test. Du coup, pour gagner du temps et pour comparer si cette méthode a des effets de traitement similaires pour deux graphiques de réseau de tailles différentes. Donc dans le processus de test du code, nous avons utilisé un autre graphe de réseau, qui a 43 nœuds et environ 5.1 millions d’arêtes.



Les deux graphes mentionnés ci-dessus sont chargés dans le code sous la forme d’un fichier txt, chaque fichier contient deux colonnes : clé et valeur qui est représenté par leur ID de type Integer.



D’abord, on enlève les en-têtes, en conservant les lignes qui ne contiennent pas le symbole« # ». De plus, on sépare la ligne en deux afin d’avoir un premier élément correspondant à la clé et le deuxième correspondant à la valeur.



Méthode

La figure ci-dessous montre le module Connected Component Finder (CCF)

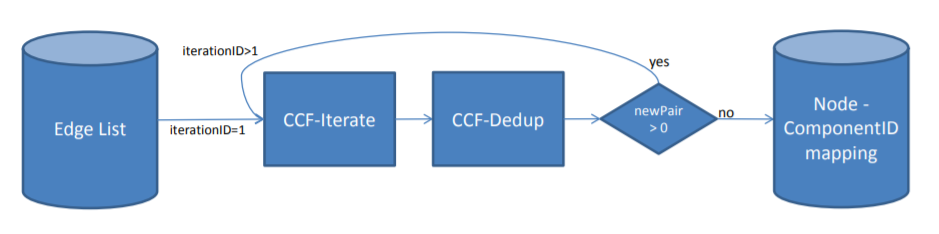


Figure 1. Connected Component Finder (CCF)

L'entrée du module est la liste de toutes les arêtes du graphique. On rappelle que les arêtes sont modélisées par une liste de (key, value) représentant les nœuds de départ et les nœuds finaux. En tant que sortie du module, ce que nous voulons obtenir est d’associer chaque nœud du graphique avec son ID du composant correspondant.

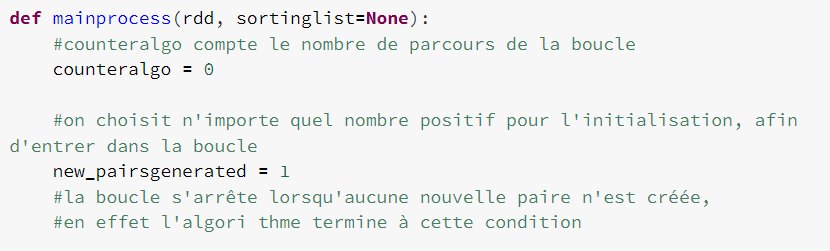
Pour plus de simplicité, nous utilisons le plus petit identifiant de nœud de chaque composant connecté comme l’identifiant de ce composant. Ainsi, le module doit sortir une table de mappage de chaque nœud du graphe vers le plus petit identifiant de nœud dans son composant connecté correspondant.

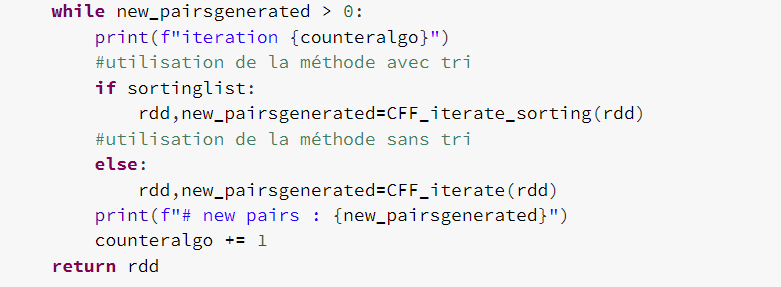
À la fin, nous avons conçu une chaîne de deux tâches MapReduce, à savoir CCF-Iterate et CCF-Dedup, qui s'exécuteront de manière itérative jusqu'à ce que nous trouvions les ID des composants correspondants à tous les nœuds du graphique.

1. CCF-iterate : lier les nœuds à l’ID de composant.
2. CCF-Dedup : copier de vidage des sorties de CCF-iterate.

Il existe 2 versions de CFF-iterate, la seconde version augmente la complexité de l’espace. Dans la première version, l’algorithme doit trouver la valeur minimale dans la liste, mais dans la deuxième version, les valeurs peuvent être transmises au Reducer de manière triée avec un partitionnement personnalisé. L’auteur mentionne dans l’article que les performances d’exécution de ces deux approches étaient très proches l’une de l’autre lorsque la taille du plus grand composant est relativement petite (c’est-à-dire jusqu’à 50 000 nœuds). Cependant, lorsqu’il existe des composants connectés avec des millions de nœuds, la seconde approche est beaucoup plus efficace.

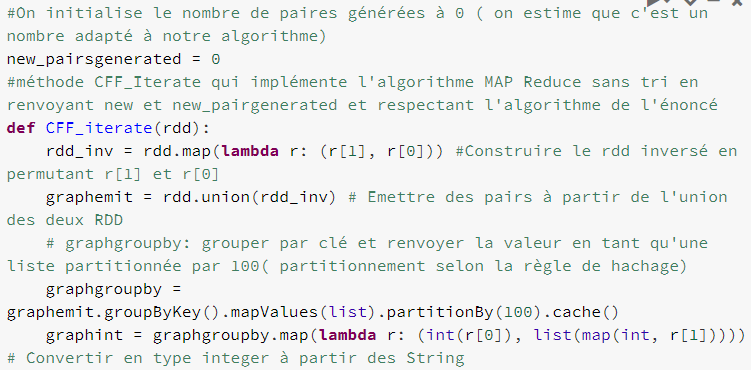
On crée d’abord une fonction mainprocess. Cette fonction contient la boucle principale. Elle est unique et peut s’adapter à chacune des deux méthodes ( sorted et non sorted). Par exemple, on peut appeler CCF\_iterate si la méthode avec tri, on peut aussi appeler CCF\_iterate\_sorting si la méthode sans tri.



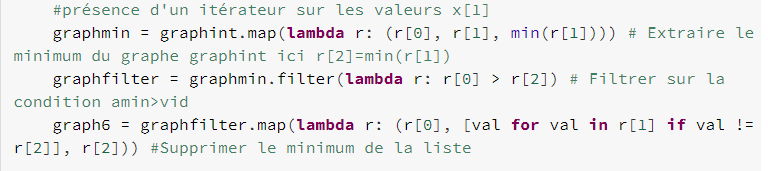


CCF-iterate naïve

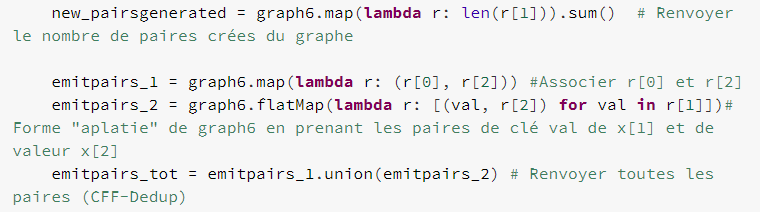
Dans cette partie il est question d’appeler la méthode Clean\_dataset, mainprocess , de comptabiliser le nombre de composants distincts avec les instructions distinct et count tout en augmentant 1 au composant du graphe le plus grand. Ainsi, on le nombre de paires générées à 0 se fait en implémentant l'algorithme Map\_Reduce sans tri en renvoyant new et new\_pairsgenerated tout en respectant l’algorithme de l’énoncé.



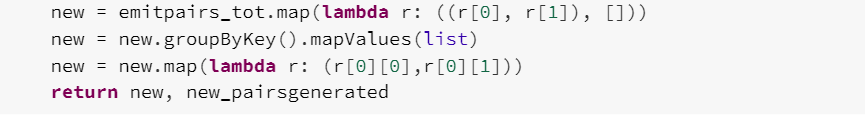
* Nous avons d’abord extrait le minimum du graphe et avons filtré sur la condition amin>vid comme l’auteur a mentionné dans l’article, ici c’est r[2]=min(r[1]), et puis on a supprimé le minimum de la liste.



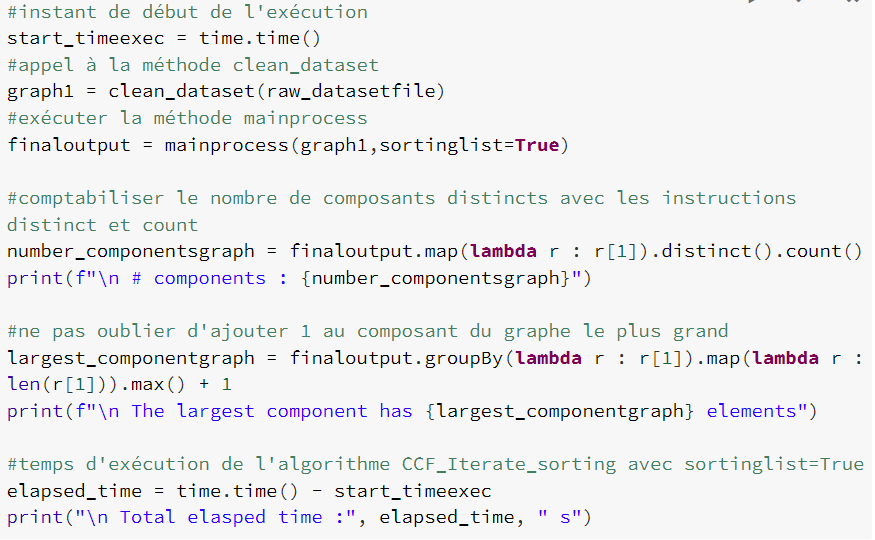
* Et ensuite nous avons renvoyé le nombre de paires crées du graphe et associe r[0] et r[2], et puis nous avons renvoyé toutes les paires, c’est l’algo CCF-Dedup.



* Nous avons traité les pairs avec les cas où cela mène à la croix comme dans le schéma de l'énoncé( ça ne renverra pas la paire suite au reducer).

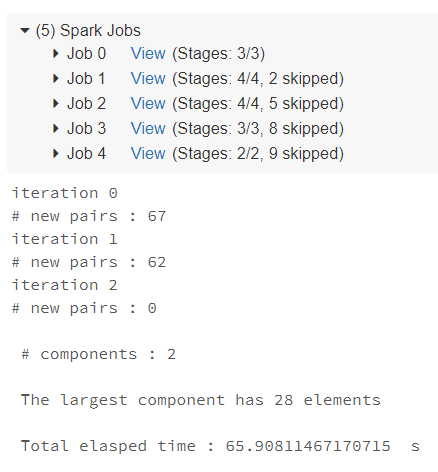


* Et à la fin, nous avons déclaré l’instant de début de l’exécution, et comptabilise le nombre de composants distincts avec les instructions distinct et count. Nous pouvons enfin obtenir le nombre de composant connexe et le temps pour obtenir tous les résultats.



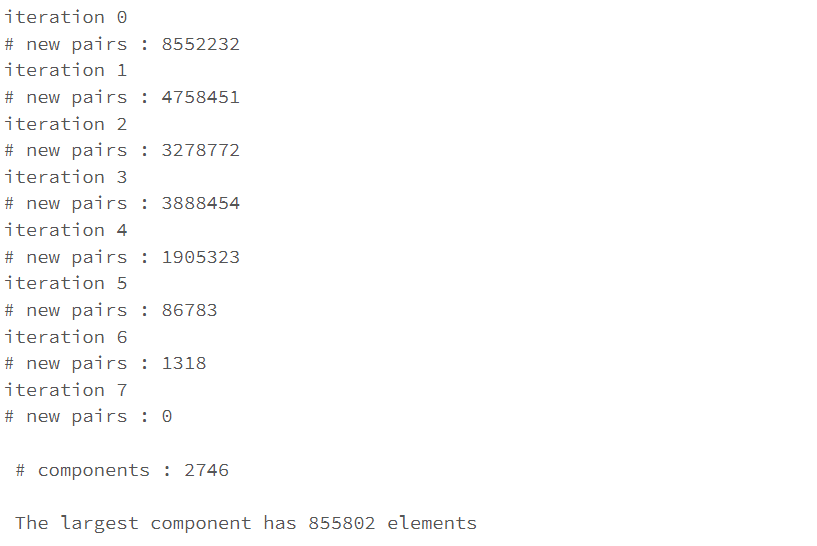
**Résultat avec l’algorithme CCF-iterate naïve**

Nous avons d’abord testé l’algorithme CCF-iterate naïve avec le graphe qui a moins de nœuds () et obtenus le résultat comme la figure ci-dessous.



Nous pouvons constater que dans ce graphe, il y a 2 composants connexes, le plus grand composant est constitué de 28 éléments et nombre d’itération est 2 fois. L’exécution de l’ensemble de l’algorithme prend : 65.90811467170715 s.

Nous avons ensuite testé cet algorithme avec le graphe avec le nombre de nœuds très important avec plus de graphe (web\_Google.txt), les résultats obtenus sont présentés dans la figure ci-dessous.

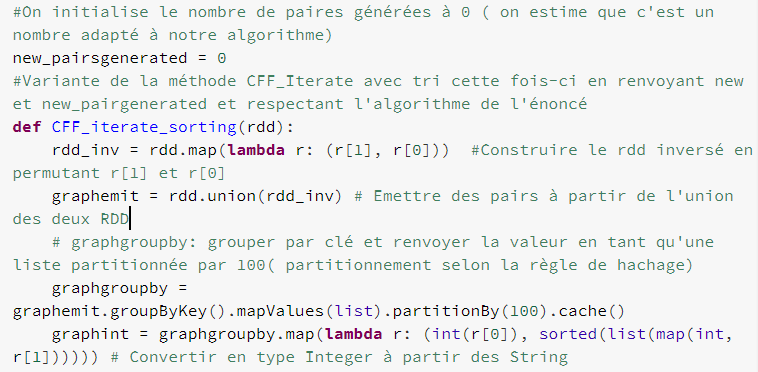




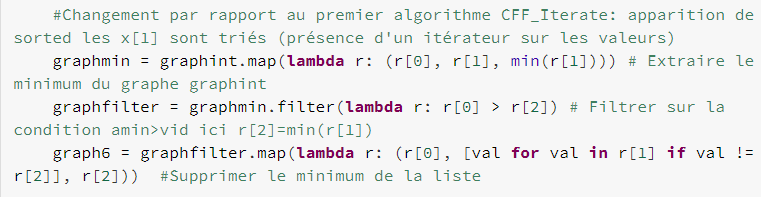
Nous pouvons constater que dans ce graphe plus grand, il y a un total de 2746 composants connexes, dont le plus grand composant contient 855802 éléments, le nombre d’itération est 7 et il faut 1331.9320995807648s pour exécuter l’algorithme complet.

CCF-iterate sorting

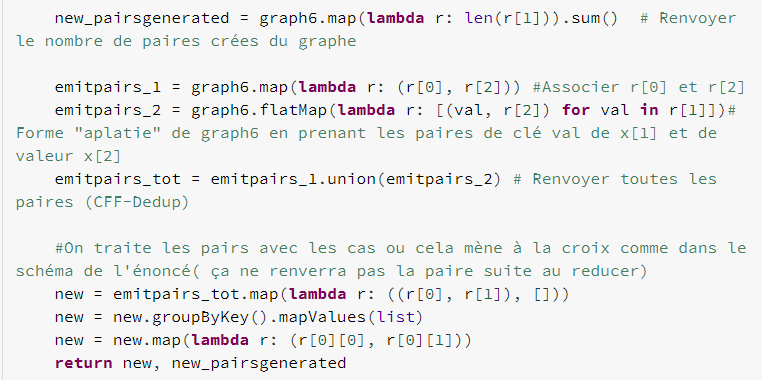
Dans l’ensemble, la deuxième version de l’algorithme CCF-iterate est similaire à la première version. Mais, il est avec le tri en renvoyant new et new\_pairsgenerated et respectant l'algorithme de l'énoncé.



* Changement par rapport au premier algorithme CFF\_Iterate: apparition de sorted les x[1] sont triés (présence d'un itérateur sur les valeurs).



* Et ensuite, comme CCF-iterate naïve, nous avons renvoyé le nombre de paires crées du graphe et associe r[0] et r[2], et puis nous avons renvoyé toutes les paires, c’est l’algo CCF-Dedup. Et nous avons traité les pairs avec les cas où cela mène à la croix comme dans le schéma de l'énoncé( ça ne renverra pas la paire suite au reducer).



Comme CCF-iterate naïve, nous avons calculé le nombre d’itérations, le nombre de composants connexes et le temps d’exécution.

**Résultat avec l’algorithme CCF-iterate sorting**

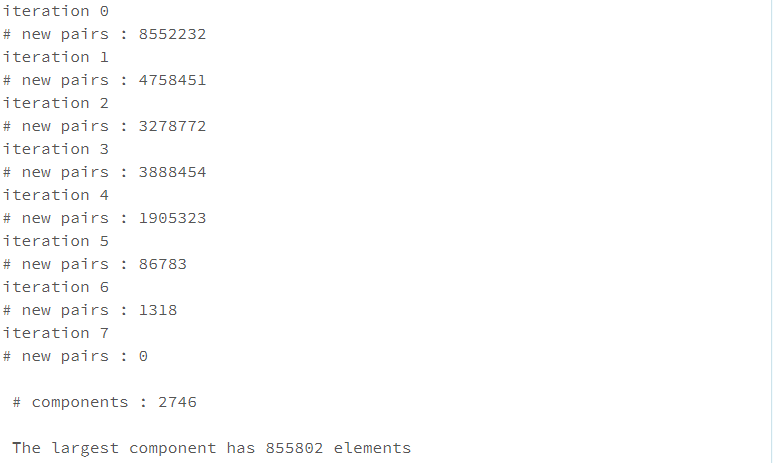
Nous avons également utilisé deux graphes de taille différentes pour tester l’algorithme CCF-iterate sorting. Tout d’abord, nous avons utilise le graphe avec moins de nœuds.

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

Nous pouvons constater que dans ce graphe, il y a 2 composants connexes, le plus grand composant est constitué de 28 éléments et nombre d’itération est 2 fois. L’exécution de l’ensemble de l’algorithme prend : 49.26186227798462 s.

Et pour le graphe avec plus nœuds, on a obtenu le résultat ci-dessous :





Nous pouvons constater que dans ce graphe, il y a 2746 composants connexes, le plus grand composant est constitué de 855802 éléments et nombre d’itération est 7 fois. L’exécution de l’ensemble de l’algorithme prend : 1287.3261504173279 s

**Comparaison des résultats :**

* Small\_web\_Google.txt

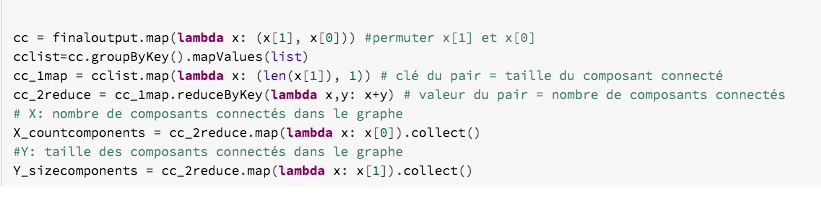
|  |  |  |
| --- | --- | --- |
|  | CCF-iterate naïve | CCF-iterate sorting |
| Nombre d’itération | **2** | **2** |
| Composant connexe | **2** | **2** |
| Temps d’exécution | 65.90811467170715 | 49.26186227798462 s |

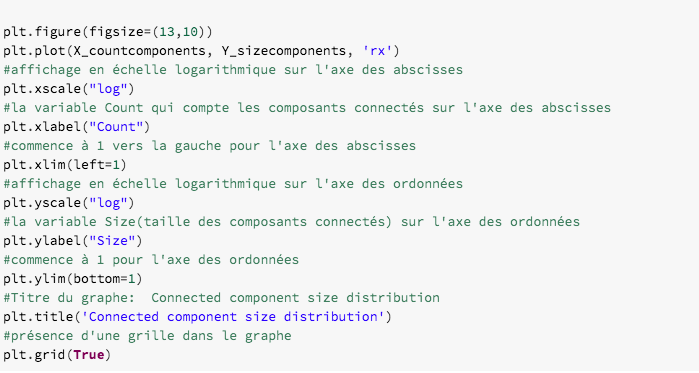
* web\_Google.txt

|  |  |  |
| --- | --- | --- |
|  | CCF-iterate naïve | CCF-iterate sorting |
| Nombre d’itération | **7** | **7** |
| Composant connexe | 855802 | 855802 |
| Temps d’exécution | 1331.9320995807648s | 1287.3261504173279 |

En comparant les résultats obtenus, on peut constater que pour des graphiques de tailles différentes, l’algorithme CCF-iterate Sorting est plus efficace que CCF-iterate Naïve et il a besoins moins de temps d’exécution. Et pour les graphes plus grands, la différence entre les deux algorithme est plus évident.

Dans cette partie du code, nous avons un nombre et taille de composants variables qui commence vers la gauche et qui est par la même occasion connectée dans le graphe en une échelle logarithmique sur l’axe des abscisses.





**Conclusion**

Avec l'implémentation de l’algorithme CCF-Iterate au fichier texte GoogleWeb, le Google toy graph parvient à identifier efficacement tous les composants connectés au sein du graphique. L'algorithme est simple à implémenter et s’inscrit pleinement dans le MapReduce. En accédant au cluster d’hadoop, les graphes utilisés en réalité comportent des milliards de nœuds et peuvent donc être traités. Dans ce projet, nous avons mis en œuvre les 2 versions de l'algorithme.