Projet PLDAC

Rapport

Analyse de Très Grands Graphes de Proteines

Étudiants :
Anyes TAFOUGHALT
Racha Nadine DJEGHALI

Encadré par :
Hubert NAACKE

26 mai 2024

Table des matières

1 Introduction

Les graphes sont des structures de données fondamentales utilisées dans une multitude de domaines, allant des réseaux sociaux à la sécurité informatique, en passant par la géographie et la bioinformatique. Leur polyvalence et leur capacité à modéliser des données complexes en font un outil précieux pour la résolution de nombreux problèmes. Cependant, l'utilisation des graphes présente des défis, notamment la gestion de grandes quantités de données, ce qui complique et alourdit les analyses en termes de puissance de calcul, de mémoire et de stockage nécessaires.

Dans ce projet, nous sommes confrontés à l'analyse d'un graphe à grande échelle qui représente les similarités entre des protéines. Chaque nœud du graphe correspond à un identifiant de protéine, tandis que les arêtes connectent deux protéines en fonction de leur degré de similarité, mesuré par un poids variant généralement de 80% à 100%, comme illustré dans la figure 1. Ces protéines peuvent également être associées à des annotations fonctionnelles, agissant comme des labels pour les nœuds respectifs. Cependant, ces annotations présentent souvent des lacunes, car de nombreuses protéines ne sont pas annotées, et même une seule protéine peut avoir des annotations différentes, ce qui ajoute une complexité supplémentaire à l'analyse.

Un défi majeur réside dans la taille volumineuse de ce jeu de données, avec un graphe contenant plus de 20 milliards d'arcs. Cette échelle pose des difficultés pour l'analyse car les outils existants sont incapables de traiter des graphes de cette taille en raison des limitations de la mémoire. Les plateformes distribuées sur plusieurs machines ne sont pas non plus efficaces pour les analyses envisagées, car la quantité de données échangée entre les machines ralentit considérablement le processus.

Dans ce contexte, nos objectifs principaux sont doubles :

— D'une part, trouver une solution pour gérer cette quantité massive de données en développant une méthode de calcul parallèle pour identifier les composantes connexes du graphe de manière optimale et rapide.

— D'autre part, prédire les annotations fonctionnelles des séquences non annotées au sein de ces composantes connexes.

Ainsi, ce projet vise à résoudre les défis liés à l'analyse des composantes tout en améliorant la prédiction des annotations fonctionnelles dans le cadre d'un graphe de protéines à grande échelle.

FIGURE 1 – Graphe de protéines

2 Analyse sur les composantes du graphe

Avant d'entreprendre le développement d'une méthode de calcul parallèle optimale et rapide pour déterminer les composantes connexes du graphe, nous avons réalisé une analyse statistique approfondie des composantes de graphes. Ce graphe en particulier contenait un nombre considérable d'arêtes, soit 24 milliards (24,344,690,305 précisément), et un total de 606,250,510 nœuds. L'objectif de cette analyse était de comprendre les propriétés structurelles et fonctionnelles de ces composantes, que ce soit sur le graphe complet ou sur un graphe filtré, en ne conservant que les arêtes présentant une similarité supérieure ou égale à 99%.

2.1 Présentation de la Base de Données

Pour débuter notre analyse, nous présentons dans la figure 2 les deux tables extraites de notre base de données, utilisées pour les évaluations et les analyses de ce projet.

query_id	query_length	target_id	target_length	match_length	percent_identity	e_value	relative_sim
10057083:0	183	77301963:3	663	183	99.5	1.5e-101	0.995
10057083:0	183	TARA_RED_19_MAG_00047_000000005302.1.2	296	183	99.5	2.2e-101	0.995
10057085:3	420	TARA_ION_45_MAG_00178_000000007249.3.3	254	254	99.6	6.5e-133	0.996
10057090:0	676	93099071:3	450	450	100	7.3e-250	1
10057093:2	876	81196815:0	100	100	99	1.1e-48	0.99
10057094:2	649	81196815:0	100	100	99	7.8e-49	0.99
10057096:4	440	TARA_ION_45_MAG_00159_000000007790.3.1	277	277	99.6	1.7e-163	0.996
10057096:4	440	TARA_PSW_86_MAG_00304_000000018971.1.5	277	277	99.3	3.2e-162	0.993
10057096:4	440	TARA_ION_45_MAG_00159_000000006200.1.1	133	133	100	1.2e-71	1
10057098:3	610	36028816:2	156	156	99.4	1.1e-83	0.994

(a) Base de données repésantant le graph de protéines

seq_id	component_id	pfamAcc
100006141:2	2069752	None
100009131:3	4951345	PF00448
10001054:1	271	PF02786
10001082:0	1849121	PF07714
100013718:0	0	PF00125
10001645:0	6858176	None
100018302:1	2042	None
100018755:4	301942	PF00096
100019211:5	11908	None
100021938:1	0	PF00022

(b) Bases de données représentant les annotations pfam et les composantes des noeuds

FIGURE 2 – Base de données

2.2 Graphe complet

2.2.1 Taille des composantes :

Nous avons étudié la distribution des tailles des composantes, c'est-à-dire le nombre de composantes pour chaque valeur de taille.

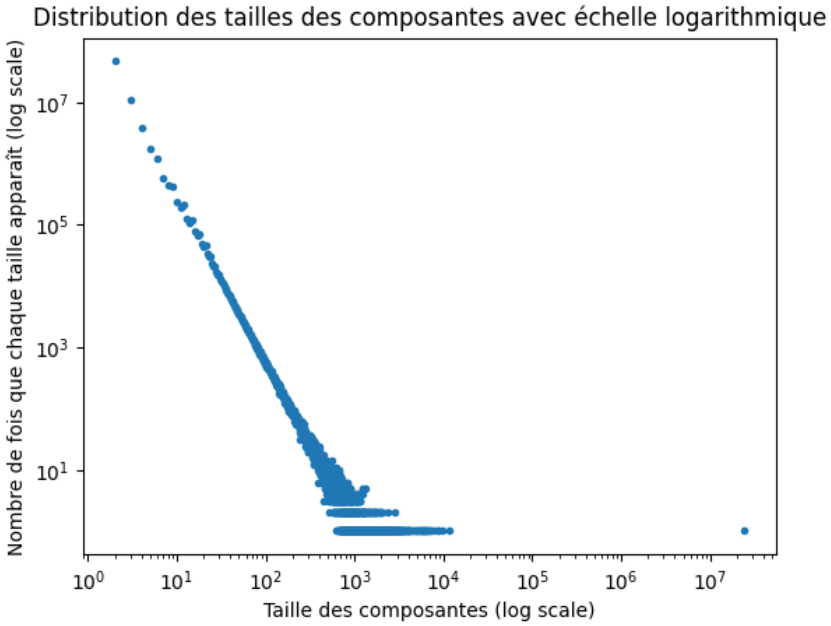


FIGURE 3 – Fréquence des composantes selon leur taille (échelle logarithmique)

Observation : Nous observons sur ce graphe une décroissance des fréquences avec l'augmentation des tailles, avec une tendance nette à la décroissance sur l'échelle logarithmique. Cela indique qu'on a 46 millions (plus exactement 46 848 540) des composantes connexes ont des tailles très petites (égale à 2), tandis que le nombre de grandes composantes est considérablement réduit. En effet, on observe une seule occurence pour la plus grande composante de taille d'environ 24 million (plus exactement 24 479 543).

2.3 Graphe G99

2.3.1 Définition du graphe G99 :

Le "G99" fait référence à une partie spécifique du graphe obtenu en appliquant un seuillage sur les arêtes. Plus précisément, cela signifie que seules les arêtes ayant une similarité égale ou supérieure à 99% sont conservées dans le graphe.

2.3.2 Calcul des homogénéités fonctionnelles :

$$\text{Indice d'homogénéité} = \frac{\text{Nombre d'annotations différentes}}{\text{Nombre de nœuds annotés}}$$

Nous avons entrepris le calcul des homogénéités fonctionnelles pour chaque composante afin de déterminer leur degré d'annotation et le nombre d'annotations distinctes qu'elles contenaient. Ce processus nous a permis de mieux comprendre la diversité des informations fonctionnelles associées à chaque composante. Une fois ces homogénéités fonctionnelles calculées, nous avons généré un histogramme pour regrouper les composantes

en fonction de leur indice d'homogénéité. Cette représentation graphique nous a permis d'observer la répartition des composantes en fonction de leur niveau d'annotation et de mettre en évidence les différences de fréquence entre les différentes catégories d'homogénéité fonctionnelle. Ces résultats ont fourni des informations précieuses sur la distribution des annotations au sein des composantes et ont éclairé notre compréhension de la diversité fonctionnelle des éléments étudiés.

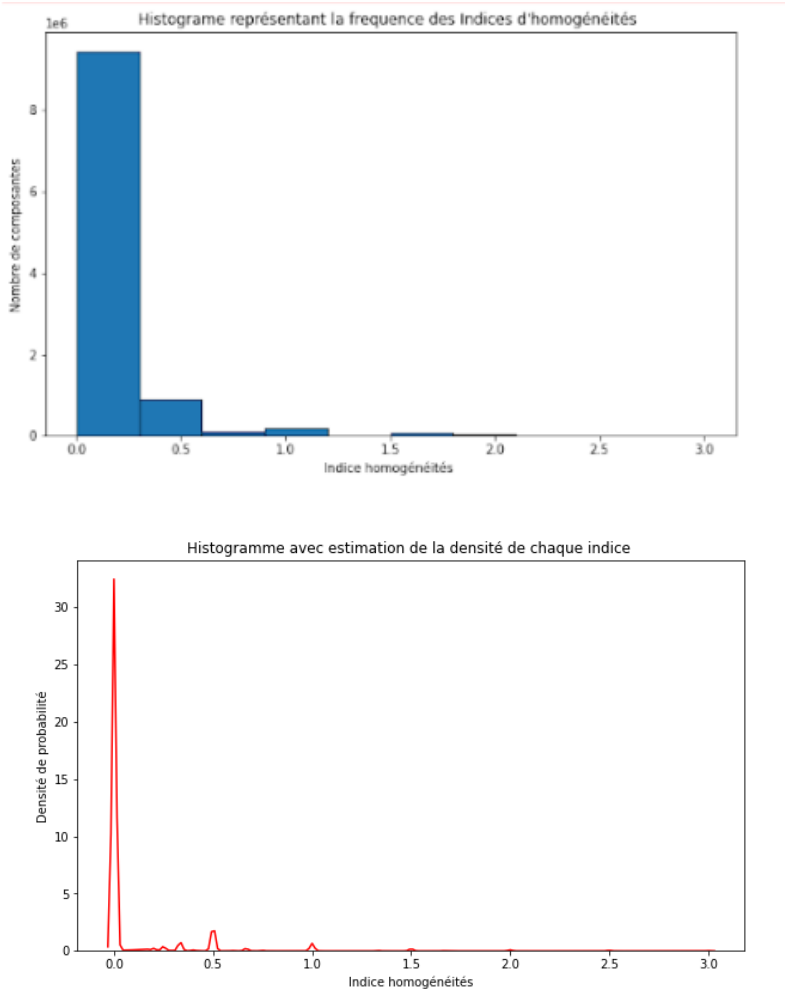


FIGURE 4 – Regroupement par indice d'homogénéités fonctionnelles

L'histogramme ainsi que l'estimation de densité mettent en évidence que la majorité des composantes ont un indice d'homogénéité égale à zéro. Cela suggère que la plupart des séquences dans ces composantes partagent la même annotation. Cependant, on observe également l'existence d'indices assez élevés, ce qui indique la présence de différentes annotations pour une même séquence dans certaines composantes.

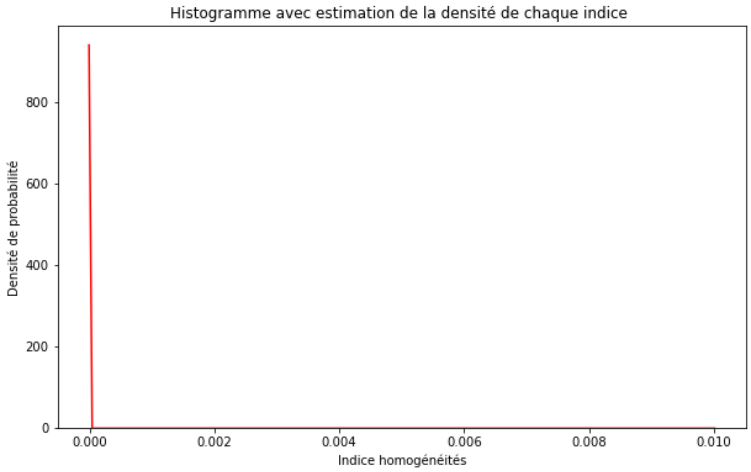


FIGURE 5 – Zoom sur le graphe de la Figure 6

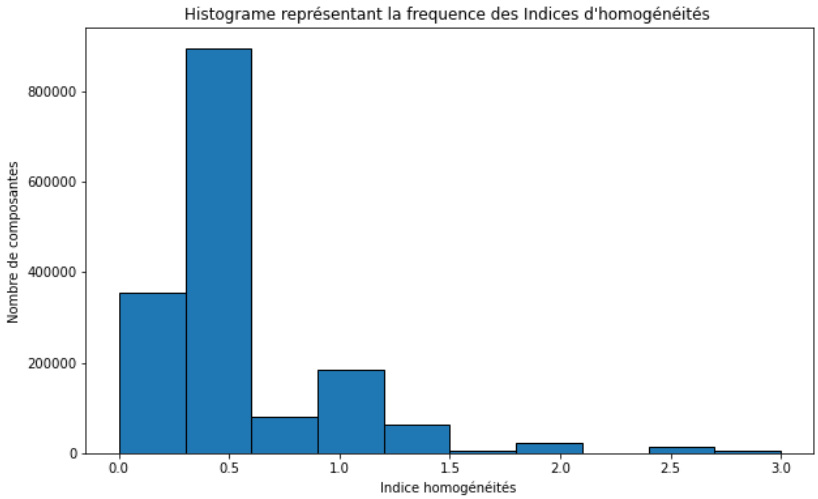


FIGURE 6 – Zoom sur le graphe sans tenir compte de l'indice d'homogénéité nul

2.3.3 Le nombre de composantes avec n annotations :

2.3.4 Annotations distinctes

En complément de l'analyse des homogénéités fonctionnelles, nous avons également dessiné un histogramme pour visualiser la fréquence des nombres d'annotations distinctes présentes dans les composantes. Cette représentation graphique nous a permis d'observer la distribution des composantes en fonction du nombre d'annotations distinctes qu'elles contenaient, offrant ainsi un aperçu plus détaillé de la diversité et de la richesse des informations fonctionnelles associées à chaque composante.

Il est remarquable que plus le nombre d'annotations différentes est élevé, moins on a de composantes. On observe particulièrement un point isolé qui représente le plus grand nombre d'annotations différentes, avec une occurrence de 1, ce qui correspond à la plus grande composante de taille 24 millions.

Plus précisément, nous avons 9 095 506 composantes comportant 0 annotations distinct, ainsi qu'une seule composante avec 2 115 annotations différentes.

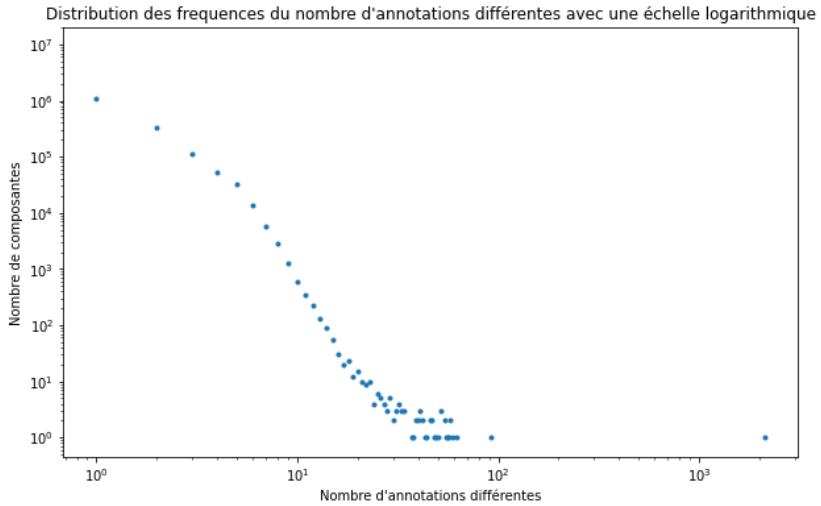


FIGURE 7 – Fréquence des nombres d'annotations distinctes

2.3.5 Annotations totales

Nous avons identifié le nombre de composantes contenant exactement n annotations, pour différentes valeurs de n allant de 0 à p. Cette analyse a été réalisée sur le graphe G99 :

1. Sur le graphe G99 sim $>$ 0.99 :

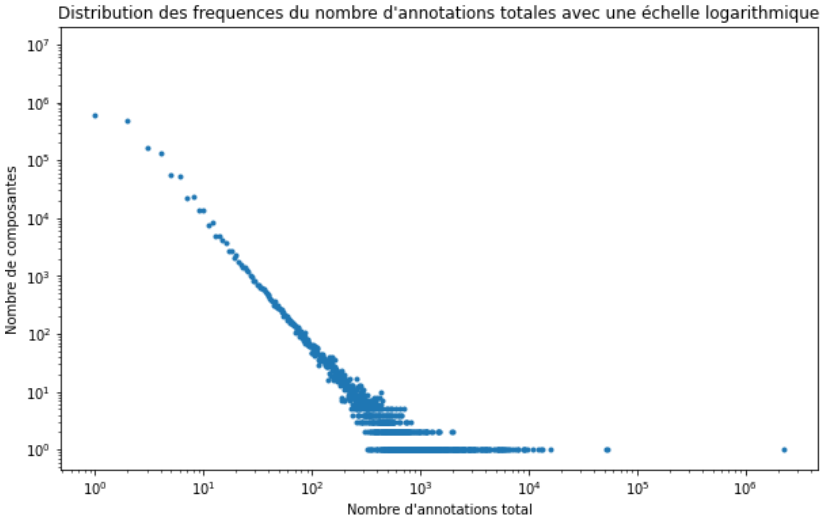


FIGURE 8 – Représentation du nombre de composantes en fonction du nombre d'annotation qu'elle contiennent sur un graphe avec seuil = 99%

D'aprés ce graphe on remarque que nous avons beaucoup de composantes (9 095 506) comportant 0 annotations au total , ainsi qu'une seule composante avec 2 231 651 annotations au total.

L'idée étant ici de réaliser cette visualisation sur différents graphes (G99, G98, ...) sur lesquels des seuils de similarité variés ont été appliqués (sim $>$ 0.99, sim $>$ 0.89, ...). Afin d'en identifier le meilleur .

2.3.6 Le nombre de composantes avec n séquences annotées :

Nous avons créeer un histograme afin de visualiser le nombre de composantes contenant un certain nombre de noeuds annotées afin de savoir en génaral cobien de sequence ont une annotations fonctionnelles sur les composantes du graphe avec un seuil de 99%

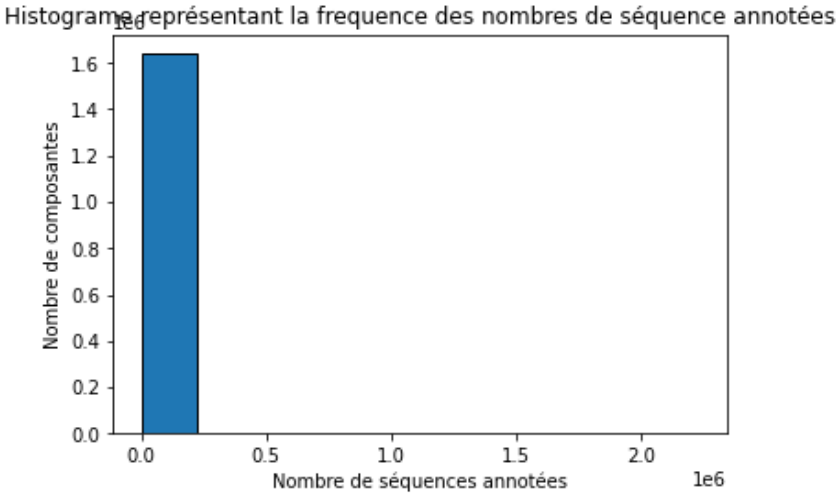


FIGURE 9 – Représentation du nombre de composantes en fonction du nombre de séquences annotées

De plus nous avons généré un graphique pour visualiser la distribution du nombre de séquences annotées dans les composantes du graphe, en utilisant une échelle logarithmique pour mieux représenter les données sur une large plage de valeurs.

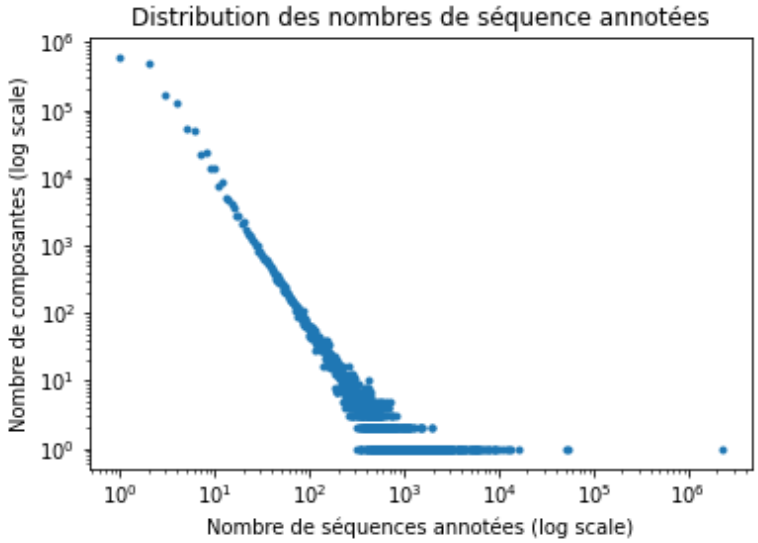


FIGURE 10 – Distribution des nombre de séquences annotées

En observant les deux graphiques, on constate que la plupart des composantes ont un faible nombre de séquences annotées, tandis qu'un nombre restreint de composantes contiennent un grand nombre de séquences annotées.

Plus précisément, nous avons 2 231 651 composantes comportant une seule séquence annotée, ainsi qu'une seule composante avec 603 760 séquences annotées.

2.3.7 Variations du nombre de noeuds annotés en fonction des tailles des composantes :

La tendance observée dans les données révèle une croissance du nombre de nœuds annotés. De plus, il est clair que pour des tailles plus petites, le nombre d'annotations varie considérablement, tandis que cette variation diminue à mesure que la taille des données devient plus grande.

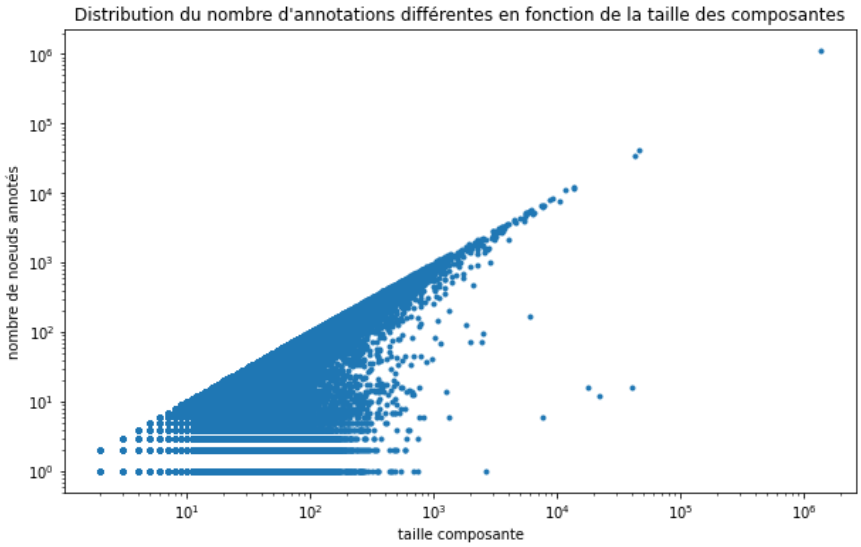


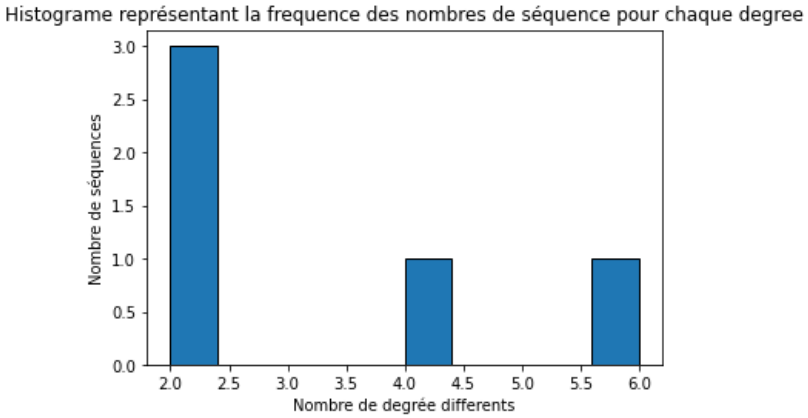
FIGURE 11 – Variation du nombre de sequences annotées en fonction des tailles des composantes avec une echelle logaritmique

2.4 Autres fonctionnlités :

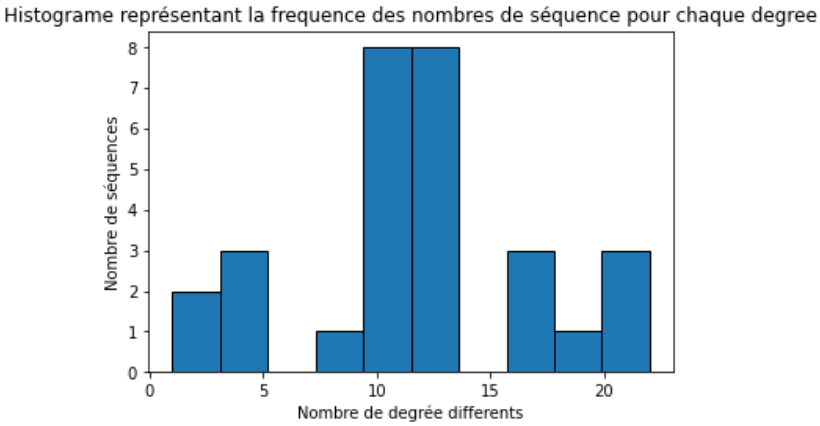
2.4.1 Analyse pour une composante donnée

Nous avons implémenté une fonction qui prend en entrée le numéro d'une composante. Cette fonction vise à offrir une représentation graphique de la distribution des degrés des nœuds au sein de la composante sélectionnée. Son utilisation permet d'approfondir la compréhension de la connectivité et de la complexité des relations entre les nœuds dans chaque composante.

Les deux graphiques suivants illustrent la distribution des degrés des nœuds pour deux composantes spécifiques, respectivement la composante 3281489 et la composante 43534.



(a) La composante 3281489



(b) La composante 43534

FIGURE 12 – La distribution des degrées des noeuds d'une certaine composante

2.4.2 Proprités de chaque composante :

Nous avons organisé les informations spécifiques pour chaque composante, y compris le nombre de séquences annotées, le nombre total d'annotations, et le nombre d'annotations uniques. Voici un extrait de cette structure de données qui comprend de telles informations :

component_id	seq_id	pfamAcc	Nb_noeuds_annotes	Nb_noeuds_annotes_tot	Nb_annotation_diff	Nb_annotation_total
650	[48387767:4]	[PF02469]	1	1	1	1
852	[154749929:5, 136...]	[PF10152, PF10152]	2	2	1	2
906	[134435551:3]	[PF00574]	1	1	1	1
1188	[142801656:5]	[PF03006]	1	1	1	1
1697	[108186008:4]	[PF07837]	1	1	1	1
2326	[41940699:1]	[PF01712]	1	1	1	1
4959	[27940831:3, 2794...]	[PF16906, PF00467...]	196	288	2	288
5794	[147176375:4, 108...]	[PF03462, PF00472...]	2	3	2	3
7117	[77098053:3, 6218...]	[PF03849, PF03849...]	4	4	1	4
7348	[5097845:0, 11039...]	[PF11913, PF11913]	2	2	1	2

FIGURE 13 – Visualisation de quelques informations proprores à chaque composantes

2.4.3 Visualisation de composantes :

Nous avons aussi intégré une fonctionnalité permettant d'afficher une composante spécifique en mettant en évidence ses nœuds annotés. Cela affiche l'annotation pfam de chaque nœud annoté.

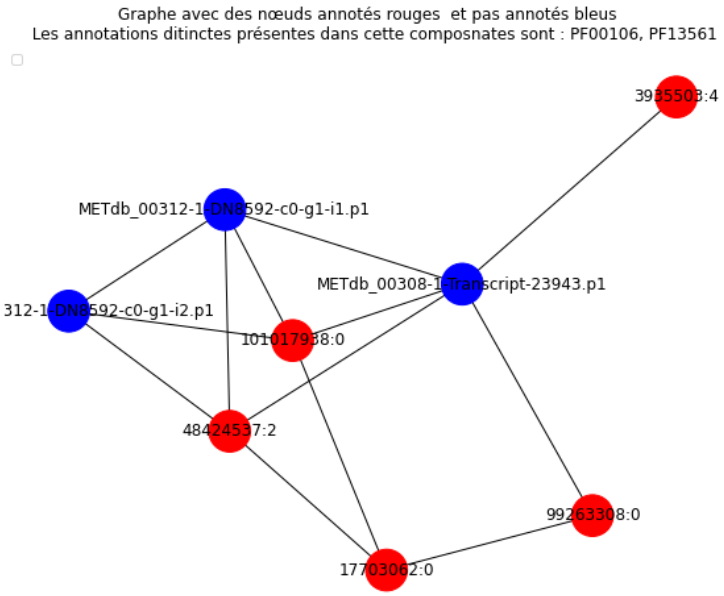


FIGURE 14 – Visualisation d'une composante avec des noeuds annotés rouges

SEQid	pfam
3935503 :4	PF00106 PF13561
48424537 :2	PF00106 PF13561
101017938 :0	PF13561 PF00106 PF00106
99263308 :0	PF13561 PF00106
17703062 :0	PF00106 PF13561

FIGURE 15 – Les annotations pfam pour chaque nœud annoté

3 Calcul des composantes connexes

3.1 Solutions déjà existantes

3.1.1 À petite échelle

La détection de composantes en mémoire est une approche qui implique le traitement du graphe dans un environnement centralisé, où toutes les opérations sont exécutées sur une seule machine. Cette méthode est généralement utilisée lorsque le graphe est de petite taille et peut être entièrement chargé en mémoire sans rencontrer de contraintes de capacité. Voici quelques caractéristiques importantes de cette approche :

- **Solution centralisée :** Toutes les opérations de chargement du graphe et de calcul des composantes sont effectuées sur une seule machine. Cela signifie que toutes les ressources matérielles de cette machine sont utilisées pour exécuter le processus.
- **Applicable seulement si le graphe est de petite taille :** Cette approche est adaptée uniquement aux graphes de petite taille, car le chargement complet du graphe en mémoire peut devenir impraticable pour les graphes de grande taille en raison des contraintes de capacité mémoire.
- **Contrainte : nb d'arcs $<$ plafond :** Il est important de respecter une limite sur le nombre d'arcs dans le graphe pour que la méthode de détection en mémoire reste réalisable. Au-delà de cette limite, la taille du graphe peut dépasser la capacité mémoire de la machine, rendant le traitement impossible.
- **Dépend des capacités mémoire de la machine :** Le succès de cette approche dépend directement des capacités de mémoire de la machine utilisée pour le traitement. Si la machine dispose de suffisamment de mémoire pour charger le graphe en entier, le calcul des composantes peut être réalisé de manière efficace.
- **Solution séquentielle :** Dans le cas où le graphe peut être chargé en mémoire, le calcul des composantes est généralement effectué de manière séquentielle, utilisant un seul cœur de processeur pour exécuter l'algorithme de détection des composantes connexes.

En résumé, la détection de composantes en mémoire est une approche pratique pour traiter des graphes de petite taille, où le graphe peut être entièrement chargé en mémoire sans rencontrer de contraintes de capacité. Cependant, cette approche n'est pas scalable pour les graphes massifs et dépend fortement des capacités de mémoire de la machine utilisée.

3.1.2 À grande échelle

Parmi les algorithmes connus pour résoudre ce problème, la *propagation de label* est largement utilisée.

- **Proposition de la propagation de label :** L'algorithme de propagation de label est une méthode itérative pour attribuer des labels à des nœuds dans un graphe. À chaque itération, le label de chaque nœud est mis à jour en prenant le minimum entre son label actuel et les labels de ses voisins, selon la formule suivante :

$$label(n) \leftarrow \min(label(n), labels\ des\ voisins\ de\ n)$$

— **Parallélisme intra-opération :** L'une des caractéristiques clés de la propagation de label est son parallélisme intra-opération. Cela signifie que chaque opération élémentaire de la propagation peut être exécutée de manière concurrente, exploitant ainsi pleinement les capacités de calcul des plateformes Big Data parallèles.

Cependant, un défi majeur dans la mise en œuvre de ce parallélisme réside dans la gestion des échanges de données entre les cœurs de calcul à la fin de chaque itération. Étant donné que la taille des données échangées peut être importante, une stratégie efficace d'échange de données est nécessaire pour minimiser les coûts de communication et maximiser les performances.

— **Gestion des résultats intermédiaires :** De plus, la gestion des résultats intermédiaires est cruciale pour assurer l'efficacité de l'algorithme. Étant donné que la taille des résultats intermédiaires peut être du même ordre de grandeur que le nombre de nœuds du graphe, une stratégie efficace de gestion de la mémoire est nécessaire pour éviter les problèmes de saturation de la mémoire vive.

Dans les cas où la taille des résultats intermédiaires dépasse la capacité mémoire disponible, le stockage temporaire sur disque peut être nécessaire. Cependant, cette opération est coûteuse en termes de temps d'accès aux données, ce qui peut entraîner des ralentissements significatifs dans le traitement des données.

3.1.3 Détail de la méthode existante à large échelle

Nous disposons de deux relations :

— **Arcs du Graphe :**

$$G(\text{src}, \text{dst})$$

— **Composantes :**

$$C(n, \text{comp})$$

1. **Initialisation :** Numéro de composante $\leftarrow$ numéro de nœud : $C_0(n, n)$
2. **Propagation par itérations :**
   (a) Jointure entre G et C tel que dst $= n$:
      — Permet d'obtenir les composantes des nœuds voisins.
   (b) Regrouper par src pour déterminer le nouveau numéro de composante :
      — Qui sera le min des composantes voisines.
      — On obtient C_1.
   (c) Recommencer pour obtenir C_2, C_3, ..., tant que C_i diffère de C_{i-1}.

La première étape ici consiste à diviser le graphe en deux partitions en utilisant un module 2 (pair et impair) sur l'identifiant des nœuds.

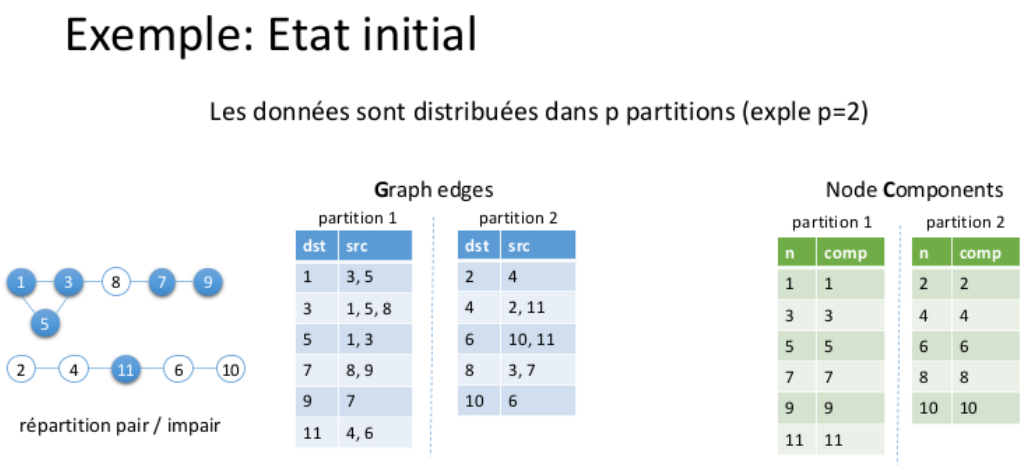


FIGURE 16 – État initial

Pour cette première itération, nous récupérons les voisins de chaque nœud et attribuons à chaque nœud la plus petite composante parmi ses voisins.

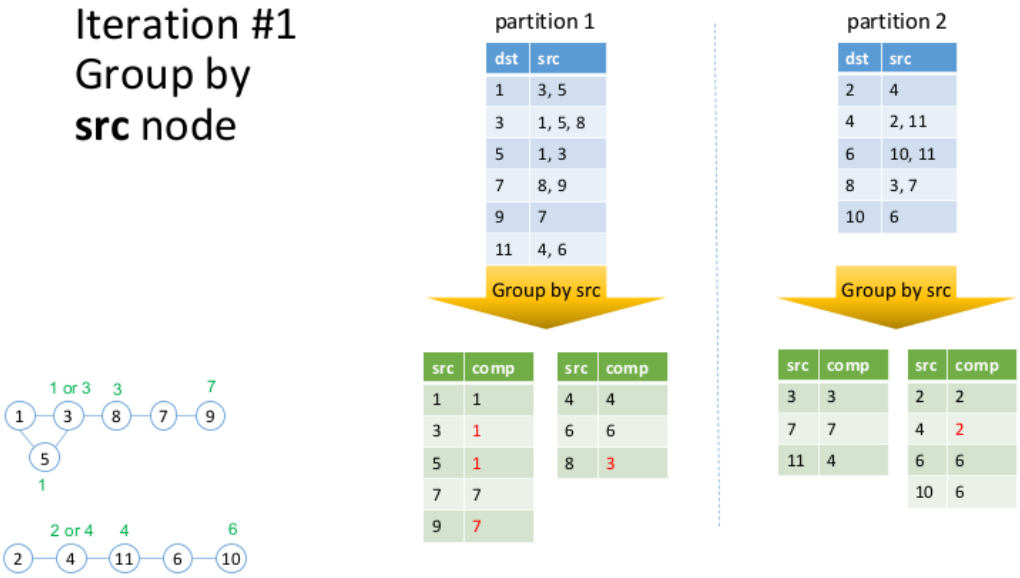


FIGURE 17 – Première itération

Nous effectuons un échange de données en attribuant à chaque nœud présent dans deux partitions différentes le numéro de composante le plus petit qui lui a été attribué.

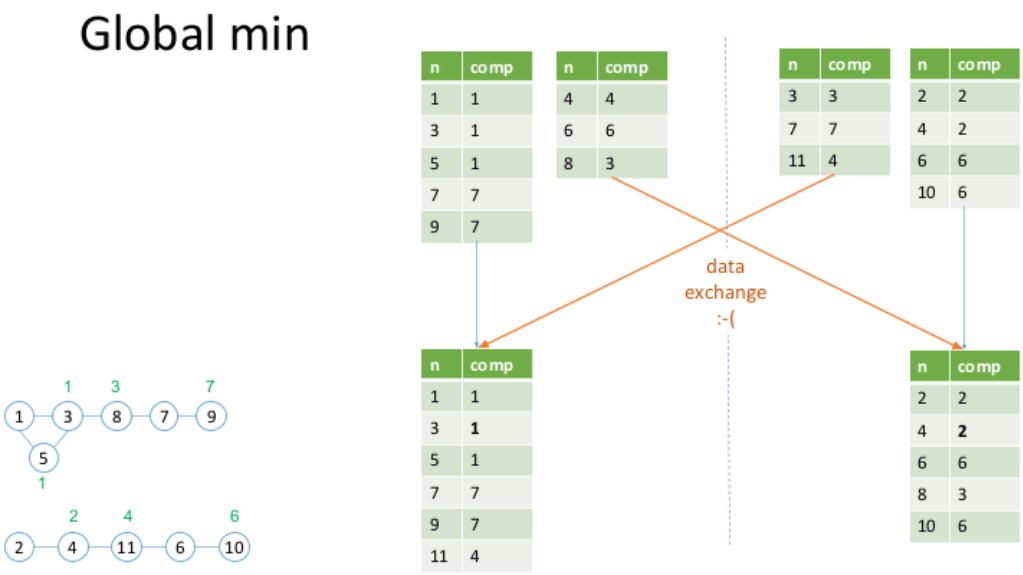


FIGURE 18 – Premier échange de données afin de récupérer le plus petit label

Nous propageons les nouveaux numéros de composantes petit à petit entre les voisins.

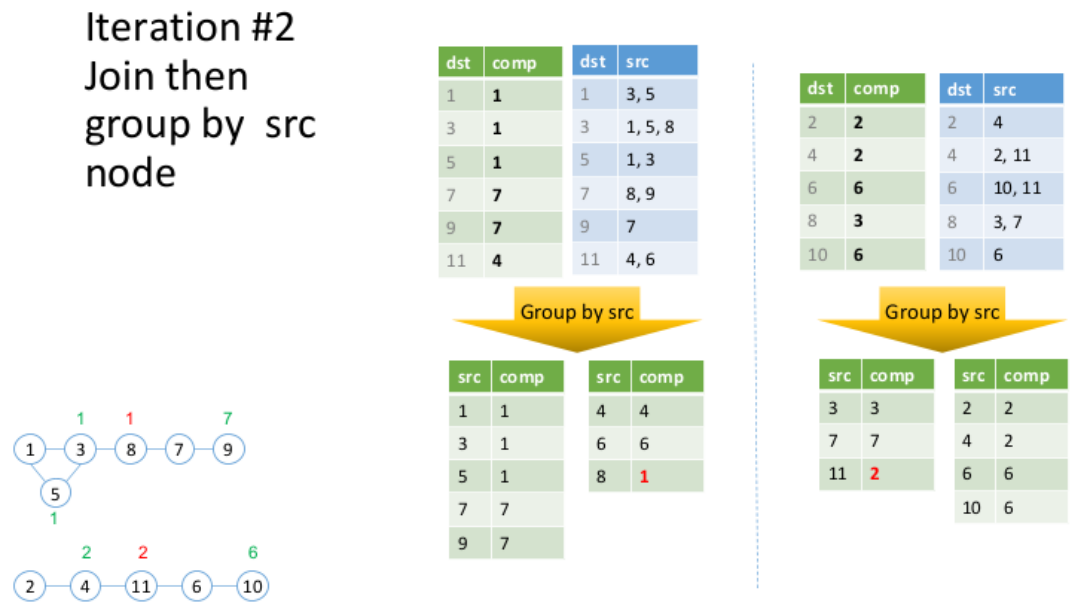


FIGURE 19 – Deuxième itération

Nous effectuons à nouveau des échanges entre les informations récupérées dans les partitions pour un même nœud commun.

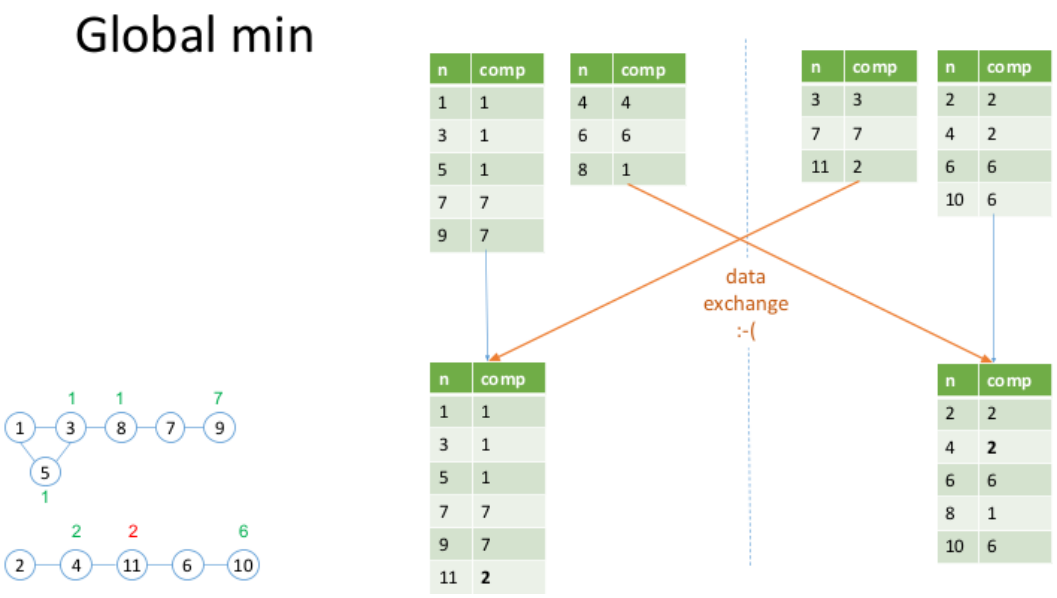


FIGURE 20 – Deuxième échange de données

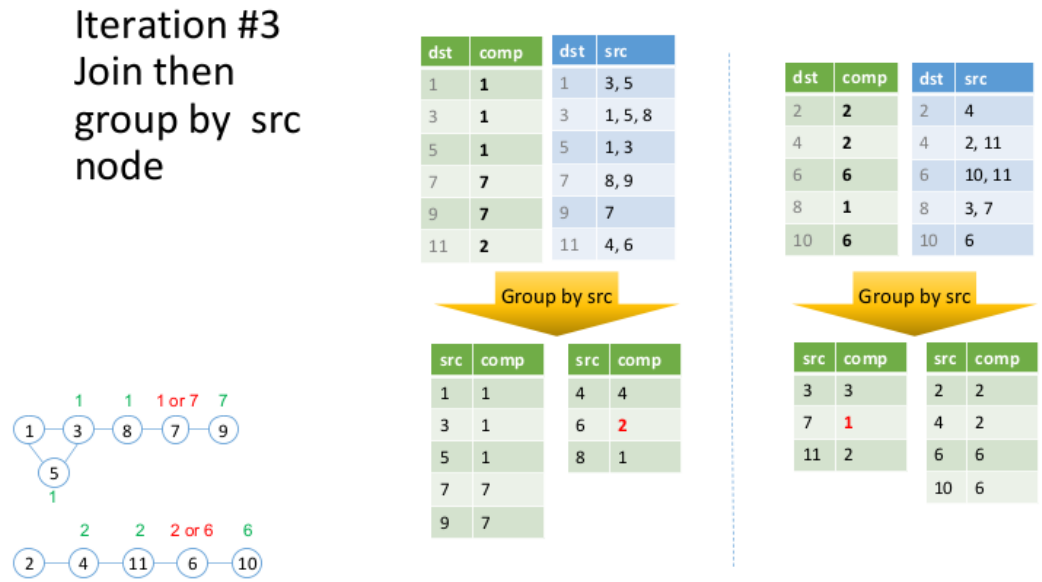


FIGURE 21 – Troisième itération

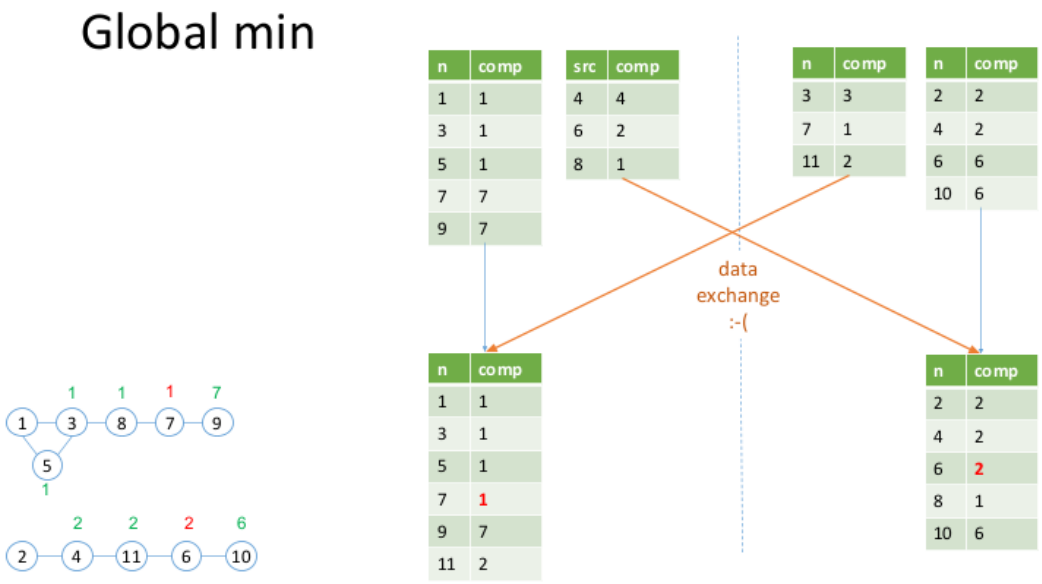


FIGURE 22 – Troisième échange de données

Nous répétons la même manipulation qu'à l'itération précédente en suivant le même principe de propagation.

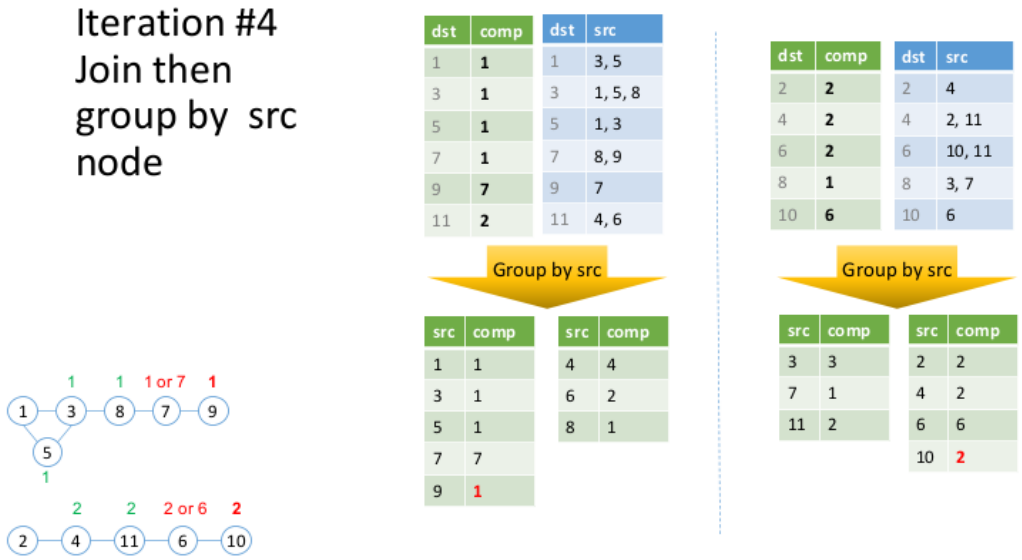


FIGURE 23 – Quatième itération

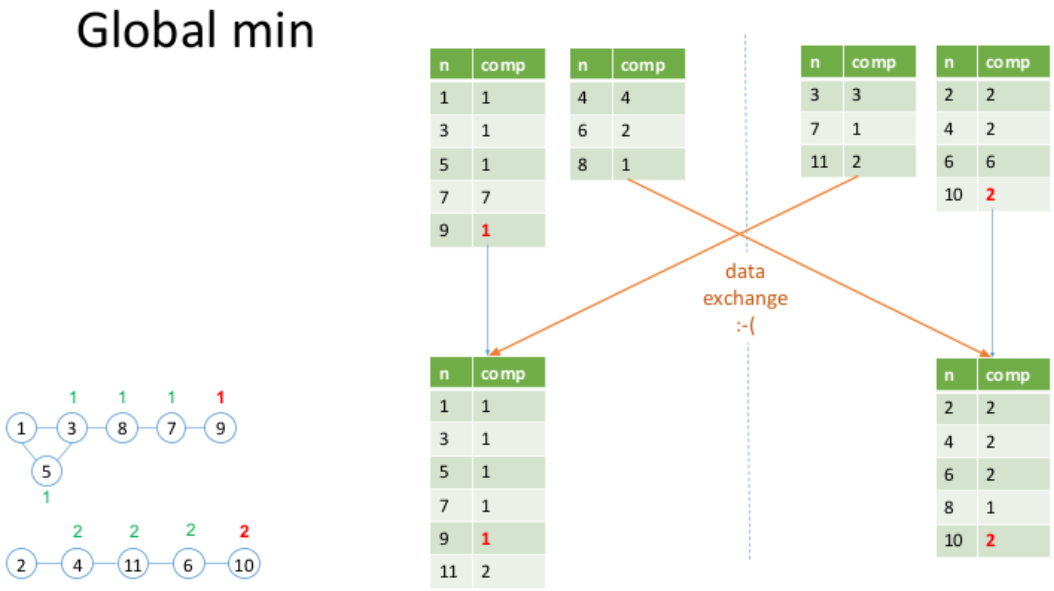


FIGURE 24 – Quatième échange de données

De même pour cette avant dernière itération, on remarque qu'après cette dernière propagation, tous les nœuds formant une même composante comportent le même numéro de composante, qui est le plus petit.

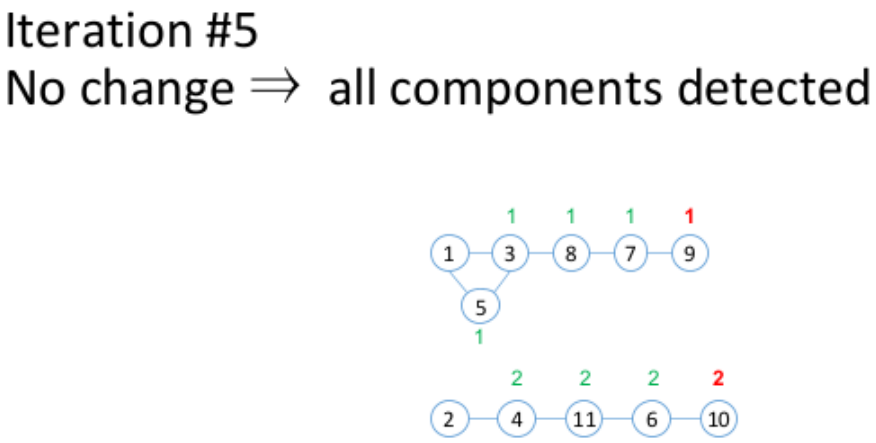


FIGURE 25 – Résultat final

Pour cette dernière itération, nous n'avons constaté aucun changement, ce qui sera notre condition d'arrêt. Cela permet de s'assurer que nous avons nos composantes finales avec les bons numéros de composantes pour chaque nœud.

3.1.4 Conclusion :

Les méthodes existantes pour le calcul des composantes dans les graphes utilisent généralement peu de ressources et limitent le nombre de calculs effectués localement. Elles dépendent fortement des échanges de données à chaque fin d'itération, ce qui n'est pas optimal et ralentit considérablement le traitement des graphes de grande taille.

Notre objectif avec la méthode proposée est différent. Nous visons à maximiser l'utilisation des ressources disponibles en effectuant davantage de calculs localement et de manière parallèle, tout en minimisant les échanges de données. Cela permet non seulement de rendre la méthode plus efficace en termes de temps, mais aussi d'optimiser l'utilisation des ressources pour le traitement de graphes massifs.

3.2 Comparaison des performances d'iGraph et GraphFrames pour le calcul des composantes

Pour entreprendre cette analyse comparative, nous avons débuté par le chargement des données, stockées au format parquet, en utilisant Apache Spark (1). Par la suite, nous avons évalué les performances de ces deux bibliothèques sur deux graphes distincts : le premier contenant 10K arêtes et le second 1M d'arêtes. Cette évaluation a été réalisée en mesurant le temps d'exécution de chaque méthode offerte par les deux bibliothèques.

À l'issue de cette comparaison directe, nous avons remarqué que iGraph (2) surpassait GraphFrames (3) en termes de rapidité lors du calcul des composantes connexes, comme indiqué dans le tableau 1. Toutefois, étant donné que le graphique que nous souhaitons analyser est de taille très importante, la méthode d'iGraph ne peut pas être appliquée directement en raison de sa limitation à des graphes de taille raisonnable.

GraphFrames est spécifiquement élaboré pour le traitement de graphes à grande échelle, bénéficiant d'une architecture distribuée et de fonctionnalités optimisées à cet effet. Basé sur Apache Spark, un framework de traitement distribué, GraphFrames offre la possibilité de distribuer les calculs sur un ensemble de coeurs, facilitant ainsi la gestion efficace de vastes volumes de données. En revanche, iGraph est conçu pour opérer sur un seule coeur, ce qui restreint sa capacité à traiter des quantités considérables de données.

Face à la contrainte de la taille de notre graphe de protéines, il est donc logique d'envisager l'utilisation de GraphFrames, spécialisé dans le traitement de graphes à grande échelle. Cependant, il est important de noter que l'utilisation exclusive de GraphFrames peut entraîner des durées de calcul prolongées, pouvant s'étendre sur plusieurs jours. C'est pourquoi, dans la suite de notre projet, nous avons envisagé une approche hybride, combinant la rapidité de iGraph sur des graphes de taille modérée avec le parallélisme offert par Apache Spark. Cette combinaison devrait nous permettre de bénéficier à la fois de la rapidité de iGraph et de la capacité de traitement des graphes à grande échelle, offrant ainsi une solution efficace pour l'analyse de graphes massifs.

	GraphFrames	**iGraph**
Graph 10K	36 secondes	0.007 secondes
Graph 1M	219 secondes	0.69 secondes

TABLE 1 – Comparaison des temps d'exécution pour le calcul des composantes connexes entre GraphFrames et iGraph

3.3 Méthode proposée

Pour mettre en place notre méthode de calcul des composantes connexes d'un graphe, nous avons suivi les étapes détaillées ci-dessous :

3.3.1 Partitionner le graphe G

Nous avons initié le processus en partitionnant le graphe en plusieurs sous-ensembles. Chaque arête a été assignée à la partition à laquelle appartient son nœud source (un exemple de partitionnement en trois partitions est illustré dans la figure 26).

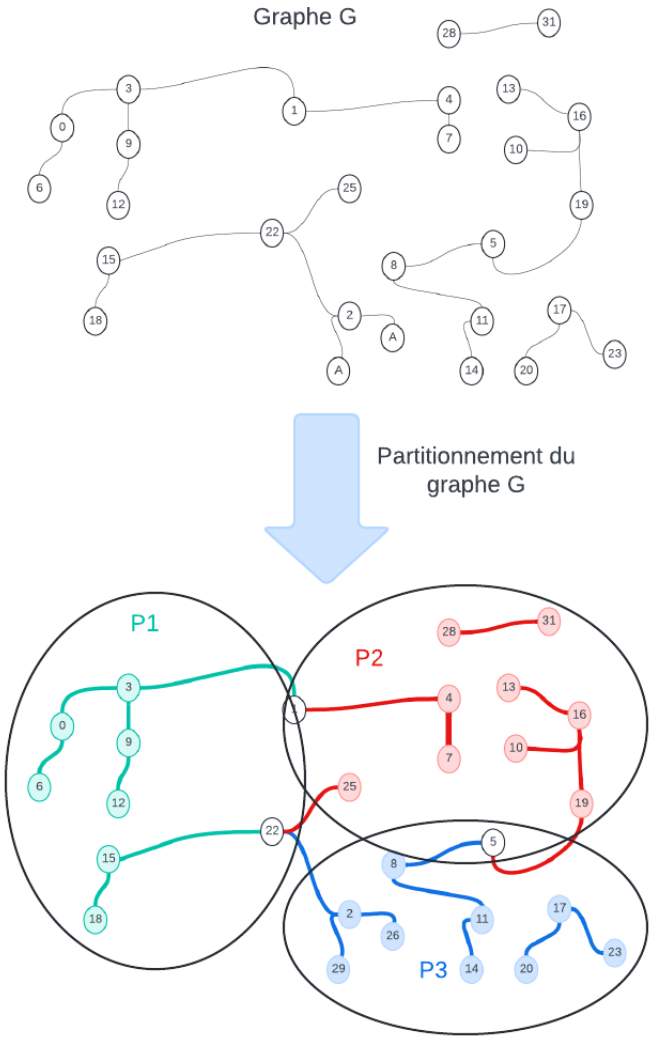


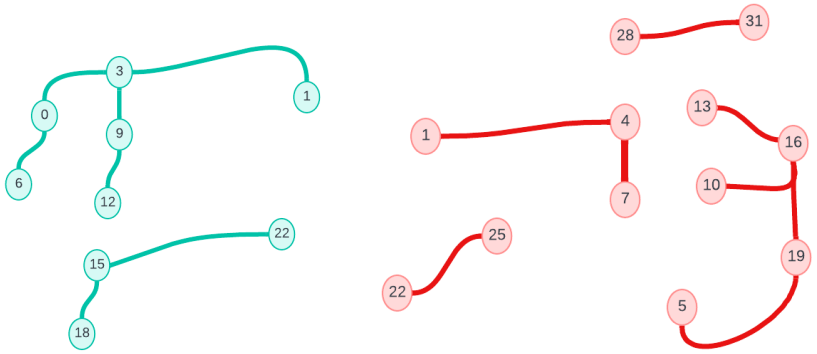
FIGURE 26 – Exemple de partitionnement d'un graphe en 3 partitions

Par la suite, nous avons sauvegardé chaque partition dans un fichier distinct au format parquet. Cette étape de partitionnement vise à distribuer efficacement les données du graphe pour un traitement parallèle ultérieur.

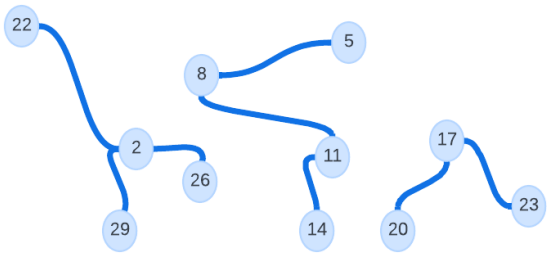
3.3.2 Calculer les composantes partielles (CP)

Pour calculer les composantes partielles de chaque partition, nous avons mis en place un DataFrame Spark dans lequel nous avons stocké les noms de fichiers de chaque partition.

Par la suite, nous avons développé une fonction qui utilise iGraph pour calculer les composantes connexes. Cette approche nous permet de bénéficier de la rapidité d'iGraph dans le calcul des composantes connexes.



(a) Partition 1 (b) Partition 2



(c) Partition 3

FIGURE 27 – Calcul des composantes partielles dans chaque partition

Ensuite, en exploitant les fonctionnalités de parallélisme offertes par Spark, nous avons parallélisé le calcul des composantes partielles pour chaque partition.

Cette stratégie permet d'accélérer significativement le processus de calcul en répartissant la charge de travail sur plusieurs nœuds du cluster Spark.

3.3.3 Création du nouveau graphe G'

Pour créer un nouveau graphe G' où chaque nœud représente une composante partielle (CP -> N), nous avons suivi une méthodologie structurée. Notre première étape a été de déterminer les composantes à fusionner.

Lorsqu'un nœud appartient à deux partitions distinctes, nous avons mis en place un mécanisme pour créer une arête entre le nœud représentant la composante partielle à laquelle il appartient dans la première partition et le nœud représentant la composante partielle correspondante dans la deuxième partition, et ainsi de suite pour toutes les partitions concernées.

Ce processus de fusion nous a permis de construire le graphe G' de manière incrémentale. Le nouveau graphe ainsi construit aura une taille nettement inférieure à celle du graphe initial G, car chaque nœud de G' représente désormais une composante partielle plutôt qu'un nœud individuel du graphe complet.

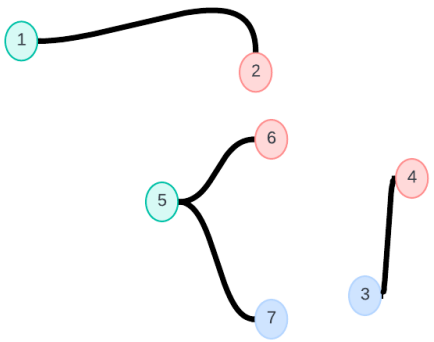


FIGURE 28 – Création du graphe G'

Cette approche de création du graphe G' nous permet de réduire efficacement la taille du graphe tout en préservant les informations essentielles sur les composantes connexes du graphe initial G.

3.3.4 Calculer les composantes (C) dans le graphe G'

En utilisant la bibliothèque iGraph, nous avons initié le processus de calcul des composantes connexes sur le nouveau graphe G', qui présente une taille significativement réduite par rapport au graphe initial G. Cette réduction de taille est le résultat du regroupement des nœuds en composantes partielles lors de la création de G'.

3.3.5 Associer les nœuds N du graphe G avec leur vrai composante C

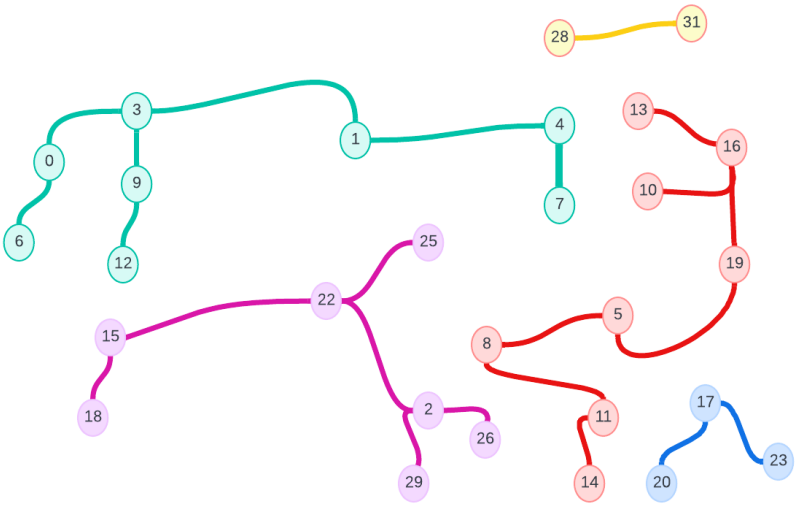


FIGURE 29 – Résultat final du calcul de composantes sur le graphe G

Après avoir calculé les composantes du graphe G', notre prochaine étape consistait à associer chaque nœud N à sa véritable composante C. Pour ce faire, nous avons effectué une jointure entre les composantes partielles obtenues précédemment et les composantes du graphe G'.

Cette opération nous a permis d'établir une correspondance entre les nœuds du graphe G et les composantes connexes auxquelles ils appartiennent réellement. Ainsi, chaque nœud N du graphe G est désormais associé à sa composante C correspondante.

3.3.6 Évaluation et validation de l'efficacité de notre méthode

Pour garantir l'efficacité de notre méthode dans le calcul des composantes connexes d'un graphe de manière plus rapide que GraphFrames, nous avons entrepris une série de tests rigoureux.

Nous avons tout d'abord testé notre méthode sur un graphe de petite taille, comprenant 10 000 arêtes, afin de comparer les résultats obtenus avec ceux de GraphFrames. Nous avons vérifié la concordance entre les composantes retournées par notre méthode et celles retournées par GraphFrames. Cette comparaison a révélé une identité parfaite entre les deux ensembles de composantes, confirmant ainsi la précision de notre méthode.

En outre, nous avons évalué les performances en termes de temps d'exécution de notre méthode par rapport à celle de GraphFrames. Nos tests ont démontré que le temps nécessaire à l'exécution de notre méthode était inférieur à celui de la méthode de calcul de composantes de GraphFrames. Cette constatation confirme l'efficacité de notre approche en termes de rapidité et d'exactitude dans le calcul des composantes connexes des graphes.

En résumé, nos résultats de validation confirment que notre méthode offre une alternative plus rapide et tout aussi précise que GraphFrames dans le calcul des composantes connexes des graphes, ouvrant ainsi la voie à des analyses plus efficaces et efficientes sur des ensembles de données de grande taille.

3.4 Analyse des performances de la solution proposée

Pour effectuer une analyse exhaustive des performances de la méthode que nous proposons, nous avons suivi une approche étape par étape.

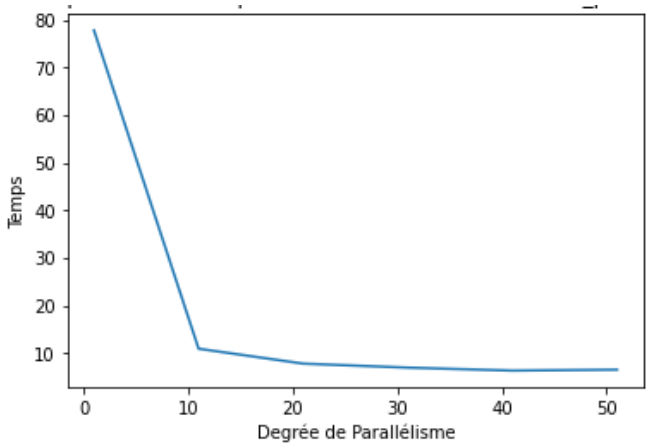
3.4.1 Lecture simple de fichiers

Dans un premier temps, nous nous sommes exclusivement concentrés sur le concept de parallélisme mis en place par l'outil Spark. Pour ce faire, nous avons évalué le temps d'une lecture simple d'un fichier en utilisant différents degrés de parallélisme. En pratique, nous avons varié le nombre de cœurs utilisés de 1 à 60 et avons calculé le temps de lecture pour chaque degré de parallélisme.

— **Observation :** Le graphique illustre une décroissance exponentielle du temps de lecture. En effet, plus nous disposons de cœurs, plus la lecture est rapide. Cependant, nous remarquons une inflexion autour de 10 cœurs. Nous en déduisons qu'au-delà de 10 cœurs, le temps de lecture reste pratiquement le même. Par conséquent, nous pouvons conclure qu'il n'est plus nécessaire d'utiliser davantage de cœurs.

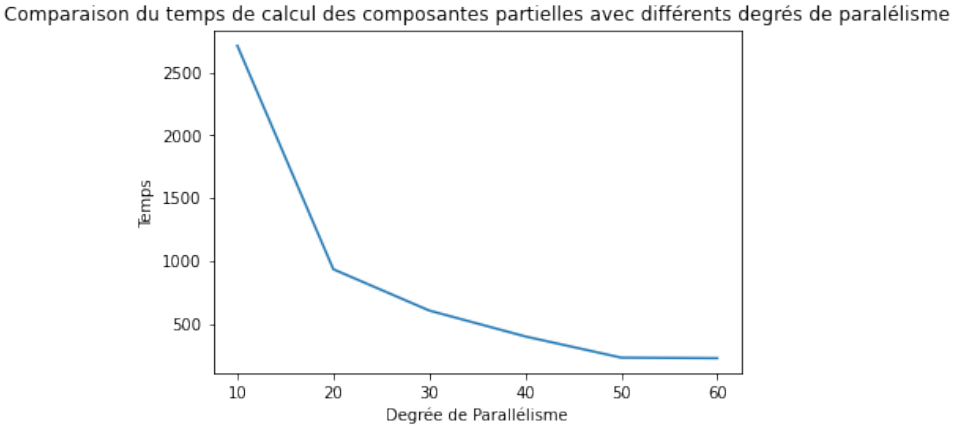
3.4.2 Temps de calcul des composantes partielles :

Ensuite, comme mentionné précédemment la première étape majeure du calcul des composantes totales consiste à calculer les composantes partielles :
Afin d'accomplir cette tâche, nous avons partitionné le graphe de plusieurs façons différentes. Nous disposions de six dossiers, chacun représentant respectivement la partition



(a) Variation du temps lors d'une lecture simple de fichier avec différents degrés de parallélisme

du graphe en 10, 20, 30, 40, 50 et 60 partitions. Ensuite, nous avons également effectué un calcul des composantes partielles en variant le nombre de cœurs (10,20,30,40,50,60) afin d'identifier les variations du temps en fonction du nombre de partitions et de cœurs utilisés.

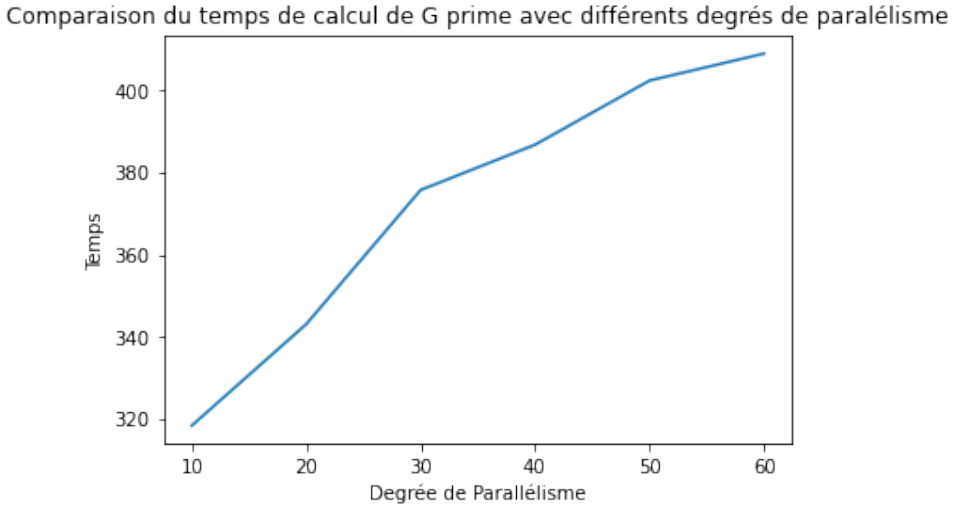


(a) Variation du temps de calcul des composantes partielles avec différents degrés de parallélisme

— **Observation :** Le graphique démontre une décroissance exponentielle du temps de calcul. En effet, à mesure que le nombre de cœurs augmente, la vitesse de calcul s'accroît. Cependant, nous remarquons deux inflexions, la première autour de 20 cœurs, suivie d'une autre vers 50 cœurs. La diminution du temps de calcul peut être expliquée par le fait qu'avec un degré de parallélisme faible, par exemple égal à 10, nous disposons de dix cœurs, chacun ayant suffisamment d'espace mémoire pour charger des partitions de taille égale à la taille du grand graphe divisée par 10. Cette taille est assez grande, et le calcul des composantes connexes prendra plus de temps que sur une partition ayant un cinquantième de la taille du grand graphe. Dans ce cas, le calcul des composantes partielles sera plus rapide.

3.4.3 Calcul de G prime :

Le calcul des composantes connexes de G' représente la deuxième partie cruciale du calcul total des composantes :

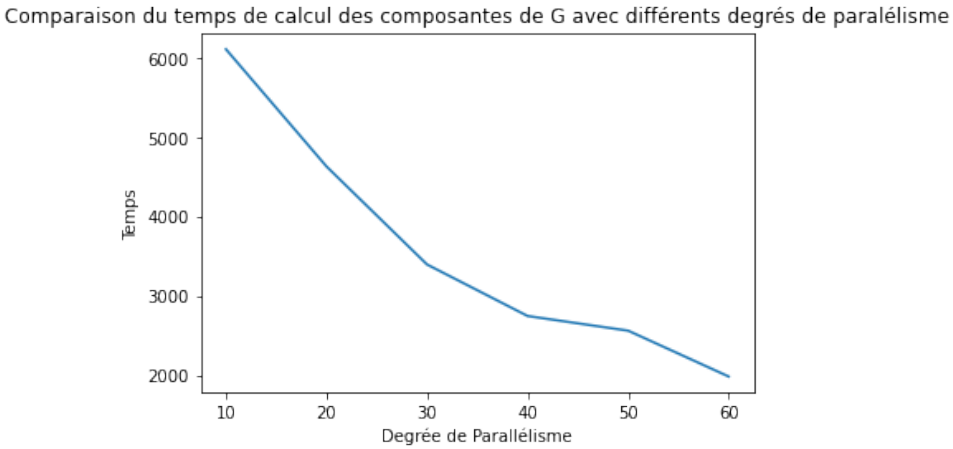


(a) Variation du temps de calcul des composantes de g prime avec différents degrés de parallélisme

— **Observation :** Nous observons une augmentation du temps d'exécution. En effet, plus nous utilisons de cœurs et donc de plus de partitions, nous remarquons une croissance du temps de calcul de G'. Cela s'explique par le fait que plus le nombre de partitions est élevé, plus le graphe G' devient grand, ce qui entraîne également une augmentation du temps de calcul des composantes connexes.

3.4.4 Calcul total :

Enfin, ce dernier graphique représente les variations du temps total de calcul des composantes.



(a) Variation du temps total de calcul avec différents degrés de parallélisme

— **Observation :** On constate que dans l'ensemble, plus le degré de parallélisme est élevé, plus le processus est rapide. Cependant, à un certain seuil, il devient moins nécessaire d'utiliser davantage de cœurs car la diminution de la vitesse devient moins significative.

3.4.5 Conclusion sur le choix du degré de paralélisme

Pour conclure, on constate qu'il existe un compromis entre l'utilisation de CPU et de l'espace mémoire lors du choix du degré de parallélisme à utiliser. En effet, pour accélérer le calcul des composantes partielles, nous souhaitons augmenter au maximum le nombre de cœurs et de partitions afin que chaque cœur puisse traiter une partition de taille relativement petite. Cependant, cette approche conduit à un graphe G' assez grand, car plus nous partitionnons le graphe, plus les composantes du graphe sont dispersées dans différentes partitions.

D'autre part, moins nous utilisons de cœurs, plus nous disposons d'espace mémoire disponible pour chaque cœur. Ainsi, nous pouvons partitionner le graphe en moins de partitions et chaque cœur aura plus d'espace mémoire, ce qui lui permettra de supporter des partitions de taille plus grande. Par conséquent, nous aurons un graphe G' relativement petit, ce qui accélérera le temps de calcul de la deuxième partie.

Par conséquent, l'objectif était de trouver le bon nombre de cœurs qui pourront supporter des partitions en calculant plus rapidement les composantes partielles, tout en aidant à avoir un graphe relativement petit pour qu'il puisse être pris en charge par la bibliothèque iGraph qui calcule assez rapidement ses composantes.

Enfin, pour notre choix final du degré de parallélisme à utiliser, nous avons décidé de ne pas nécessairement utiliser le même degré de parallélisme tout au long de la méthode, mais plutôt de partitionner le graphe en 20 partitions et d'utiliser 20 cœurs. Cependant, pour le calcul de G' ainsi que pour les opérations de jointure et de regroupement, nous avons augmenté le nombre de cœurs à 60. Cette décision a été prise pour accélérer ces opérations qui bénéficient d'un degré de parallélisme plus élevé et qui sont plus rapides avec plus de cœurs disponibles.

4 Prédiction des labels dans une composantes

Étant donné que le graphe de protéines révèle des similarités entre les séquences de protéines, notre objectif est de déterminer les annotations des séquences non annotées en se basant sur celles des protéines déjà annotées qui leur sont similaires. Dans cette section, nous exposerons nos diverses stratégies visant à prédire les annotations fonctionnelles des séquences non annotées au sein d'une composante donnée.

4.1 Attribution de labels basée sur les composantes connexes

L'approche d'attribution de labels basée sur les composantes connexes consiste à analyser la structure connectée du graphe pour prédire les labels des nœuds non étiquetés. Tout d'abord, les composantes connexes du graphe sont identifiées, regroupant ainsi les protéines similaires ayant des interactions étroites. Ensuite, pour chaque composante connexe, les labels présents parmi ses membres sont examinés. Le label dominant dans chaque composante est alors déterminé en comptant le nombre d'occurrences de chaque

label. Enfin, les nœuds non étiquetés de chaque composante se voient attribuer le label dominant de la composante correspondante.

4.2 Attribution de labels basée sur les voisins

Dans l'approche d'attribution de labels basée sur les voisins, les labels des nœuds non étiquetés sont prédits en se basant sur les labels de leurs voisins directs dans le graphe. Pour chaque nœud non étiqueté, les labels de ses voisins sont examinés et le nombre d'occurrences de chaque label est compté. En alternative, le label dominant parmi les voisins peut être attribué au nœud non étiqueté.

4.3 Attribution de labels basée sur le voisin le plus similaire

Dans cette approche, chaque nœud non étiqueté se voit attribuer le label de son voisin avec qui il possède le plus grand degré de similarité. Sur chaque arc du graphe, nous disposons du degré de similarité entre les protéines qu'il relie. Ainsi, pour chaque nœud non étiqueté, nous examinons ses voisins directs dans le graphe et identifions le voisin avec le plus grand degré de similarité. Ensuite, le nœud non étiqueté se voit attribuer le label de ce voisin le plus similaire. Cette approche repose sur l'hypothèse que les protéines ayant une similarité élevée partagent souvent des caractéristiques et des propriétés fonctionnelles similaires.

4.4 Attribution de labels basée sur un modéle de machine learning

La dernière approche adoptée pour prédire les labels des séquences de protéines non annotées repose sur l'utilisation d'un modèle de machine learning. Cette méthode nous offre la possibilité d'automatiser cette tâche en tirant parti des techniques d'apprentissage supervisé. Voici les étapes détaillées que nous avons suivies :

— **Prétraitement des données :** Nous avons débuté par la sélection d'une composante suffisamment grande dans notre graphe de protéines. Les caractéristiques de cette composante sont résumées dans le tableau 2

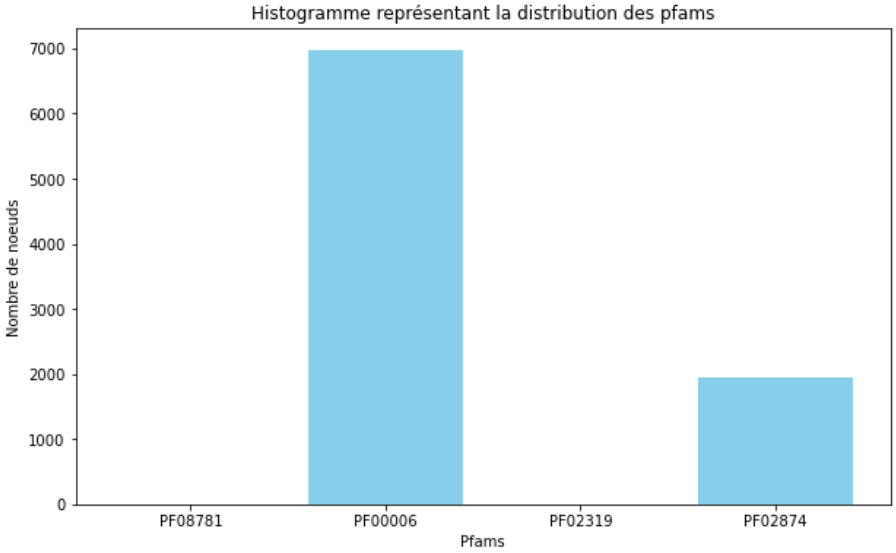
Caractéristique de la composante choisie	Valeur
Nombre d'arêtes	600 863
Nombre de noeuds	110 442
Nombre de noeuds annotés	7 444
Nombre de pfam différentes	4

TABLE 2 – Caractéristique de la composante selectionnée

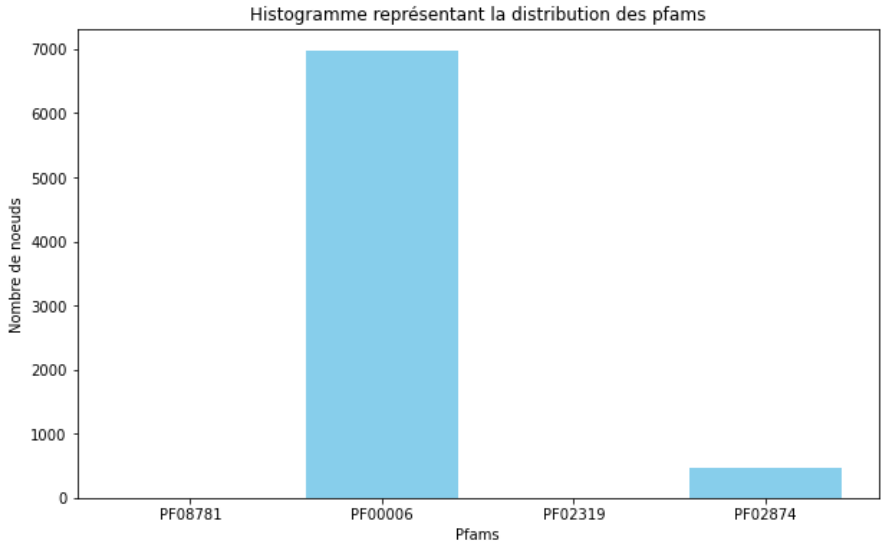
Cette composante, considérée comme homogène, offre un environnement propice à l'entraînement d'un modèle de classification.

Pour transformer notre problème de prédiction en une tâche de classification multi-classes plutôt qu'un problème multi-labels, nous avons adopté une approche où chaque nœud annoté se voit attribuer un seul label. Pour ce faire, nous avons mis en œuvre un algorithme de sélection des annotations, privilégiant les annotations les

plus représentatives dans le cas de nœuds avec plusieurs annotations. Si un nœud avait plusieurs annotations différentes, nous en avons sélectionné une en choisissant, dans l'idéal, une annotation déjà choisie précédemment, ou sinon celle avec la plus grande fréquence dans la composante. Les deux graphiques suivants illustrent la fréquence des annotations Pfam dans la composante avant et après l'application de cet algorithme.



(a) Avant l'application de l'algorithme



(b) Aprés l'application de l'algorithme

FIGURE 34 – La fréquence des annotations Pfam dans la composante sélecionnée

— **Calcul des embeddings :** Après la phase de prétraitement, nous avons calculé les embeddings pour chaque nœud de notre composante de graphe. Pour cela, nous avons utilisé l'algorithme **Node2Vec** (4), une méthode d'apprentissage non supervisé, pour générer des représentations vectorielles des nœuds de notre graphe de protéines. Node2Vec fonctionne en effectuant des marches aléatoires contrôlées sur le graphe, capturant ainsi les structures de voisinage des nœuds. En ajustant les

paramètres de ces marches aléatoires. Les embeddings ainsi obtenus pour chaque nœud capturent les informations topologiques et structurelles du graphe, les représentant de manière numérique dans un espace vectoriel. Ces embeddings peuvent ensuite être utilisés comme des features dans notre modèle de machine learning pour prédire les labels des nœuds non annotés, en se basant sur les relations complexes entre les nœuds présents dans le graphe.

— **Entraînement du modèle :** Une fois les embeddings calculés, nous avons préparé nos données pour l'entraînement du modèle. Nous avons utilisé les embeddings des nœuds annotés comme features et leurs labels correspondants comme cibles. Ensuite, nous avons divisé notre ensemble de données en ensembles d'entraînement et de test pour évaluer les performances du modèle. Pour la modélisation, nous avons choisi d'utiliser **MLPClassifier** (5), un classificateur à réseau de neurones multicouches, réputé pour sa capacité à capturer des relations complexes entre les données. Nous avons ensuite entraîné le modèle sur l'ensemble d'entraînement en ajustant itérativement les poids du réseau de neurones pour minimiser la fonction de perte.

— **Évaluation des performances :** Une fois le modèle entraîné, nous l'avons évalué en le testant sur l'ensemble de test. L'évaluation des performances de notre modèle de classification a révélé des résultats prometteurs. Nous avons utilisé des métriques standard telles que la précision, le rappel et le F1-score pour évaluer la performance du modèle sur les deux classes d'annotations Pfam : PF00006 et PF02874.

	precision	recall	f1-score	support
PF00006	0.99	0.98	0.99	1403
PF02874	0.75	0.83	0.78	86
accuracy			0.97	1489
macro avg	0.87	0.90	0.89	1489
weighted avg	0.98	0.97	0.97	1489

FIGURE 35 – Performance du modéle

— Pour la classe majoritaire, PF00006, le modèle a obtenu des scores exceptionnellement élevés avec une précision de 0.99, un rappel de 0.98 et un F1-score de 0.99. Cela indique que le modèle est capable de prédire cette classe avec une grande précision et de rappeler la majorité des instances réelles de cette classe.

— Pour la classe minoritaire, PF02874, les performances sont légèrement inférieures mais toujours acceptables. Le modèle a atteint une précision de 0.75, un rappel de 0.83 et un F1-score de 0.78. Bien que ces scores soient plus modestes que pour PF00006, ils restent dans des plages acceptables pour une classification efficace.

— L'analyse globale des performances du modèle révèle une précision globale pondérée de 0.98 et un rappel pondéré de 0.97, ce qui indique une performance globale élevée. Cependant, il est important de noter que ces performances sont influencées par la distribution déséquilibrée des classes, comme le montre l'accuracy globale de 0.97.

— Pour améliorer les performances du modèle, des techniques telles que le ré-échantillonnage ou la pondération des classes peuvent être envisagées pour mieux gérer les classes minoritaires. En outre, une analyse approfondie des erreurs spécifiques du modèle pourrait fournir des insights précieux pour son amélioration continue.

En résumé, notre modèle a démontré des performances solides dans la classification des annotations Pfam, avec des scores élevés pour la classe majoritaire et des performances acceptables pour la classe minoritaire.

Cette approche nous permet d'automatiser efficacement le processus d'attribution de labels aux séquences de protéines non annotées, ce qui peut être crucial dans l'analyse des grands ensembles de données biologiques.

5 Conclusion

Dans le cadre de ce projet de recherche, nous avons exploré des méthodes big data pour l'analyse de très grands graphes de protéines. Notre objectif était de proposer une solution efficace pour le calcul de composantes connexes de notre énorme graphe de protéines contenant 20 milliards d'arcs, et ensuite d'utiliser des modèles de machine learning pour prédire des labels aux protéines non annotées.

Nous avons notamment étudié l'approche de réduction du graphe avec un seuillage sur le poids des arcs et la détection des composantes connexes. Cependant, nous avons identifié des limitations dans les solutions existantes pour le calcul des composantes connexes, en particulier en raison de la taille massive des données non supportée par la bibliothéque iGraph et qui demande beaucoup trop de temps à la bibliothéque GraphFrames pour calculer les composantes connexes.

En réponse à ces défis, nous avons proposé une méthode de calcul parallèle pour déterminer les composantes connexes d'un graphe en tenant compte du compromis entre l'espace mémoire et le degré de parallélisme, et donc de la vitesse de calcul. De plus, nous avons utilisé un modèle de machine learning pour la prédiction de labels.

6 Perspectives

Pour aller plus loin, plusieurs pistes de recherche peuvent être explorées :

— Pour améliorer la prédiction des annotations fonctionnelles des séquences de protéines non annotées, une approche prometteuse consiste à exploiter les communautés au sein des composantes connexes du graphe. Cette méthode repose sur la division de chaque composante connexe en communautés plus petites à l'aide d'algorithmes de détection de communautés tels que Louvain, Infomap ou Girvan-Newman. Ces algorithmes permettent de partitionner le graphe en sous-graphes plus compacts, où les nœuds (protéines) sont plus densément connectés entre eux qu'avec le reste du graphe.

Les séquences de protéines au sein de la même communauté partagent souvent des annotations fonctionnelles similaires. Pour prédire les annotations des nœuds non annotés, des techniques comme la propagation de labels peuvent être utilisées : les annotations majoritaires au sein de chaque communauté sont attribuées aux

séquences non annotées. Cette approche permet de tirer parti des connexions locales pour améliorer la précision des prédictions fonctionnelles.

— Étendre la solution proposée pour traiter des graphes encore plus volumineux en optimisant les performances et en réduisant la consommation de ressources.

— Revoir le filtrage du graphe en utilisant l'attribut de taille(size) plutôt que la similarité. En effet, cela permettrait de regrouper des organismes ayant la même taille ensemble, ce qui pourrait révéler des informations biologiques importantes.

— Utiliser d'autres techniques d'apprentissage supervisé, telles que les forêts aléatoires, pour classer les différentes protéines de chaque composante finale selon leurs annotations fonctionnelles. Cela pourrait permettre d'identifier des patterns et des associations entre les protéines, contribuant ainsi à une meilleure compréhension des processus biologiques

— Pour les composantes hétérogènes où les modèles de classification ne sont pas performants, nous pouvons effectuer un clustering sur ces composantes et propager l'annotation majoritaire aux nœuds non annotés dans chaque cluster.

En conclusion, ce projet ouvre de nouvelles perspectives passionnantes dans le domaine de l'analyse de graphes de protéines à grande échelle, avec des implications potentielles pour la recherche en biologie structurale et fonctionnelle.

7 Références

Références

[1] https://spark.apache.org/.

[2] https://igraph.org/r/doc/components.html.

[3] https://graphframes.github.io/graphframes/docs/_site/user-guide.html.

[4] https://snap.stanford.edu/node2vec/.

[5] https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html.