Compte-rendu du projet de bases de données spécialisées

Interactions entre protéines humaines clusterisées

Dorothée Huynh, numéro 71804295, année 2020 - 2021

Préparation de la base de données

Sélection des informations pertinentes pour la base de données

La base de données originale, accessible sur le site https://string-db.org, contient des données sur 5000 organismes vivants. Puisqu'elle est trop volumineuse pour mon projet (fichiers d'au moins une centaine de Go), j'ai sélectionné et téléchargé la partie de la base de données spécifique à l'espèce humaine.

Les fichiers de données téléchargées sont au format txt. Ils ressemblent à des fichiers csv dans le sens où les données sont rangées par ligne et dans chaque ligne, les valeurs sont séparées par un séparateur. Ce séparateur n'est pas une virgule comme dans les fichiers csv, mais est une tabulation ou un espace selon le fichier. Ce formatage de données facilitera l'extraction des données pour remplir la base de données du projet.

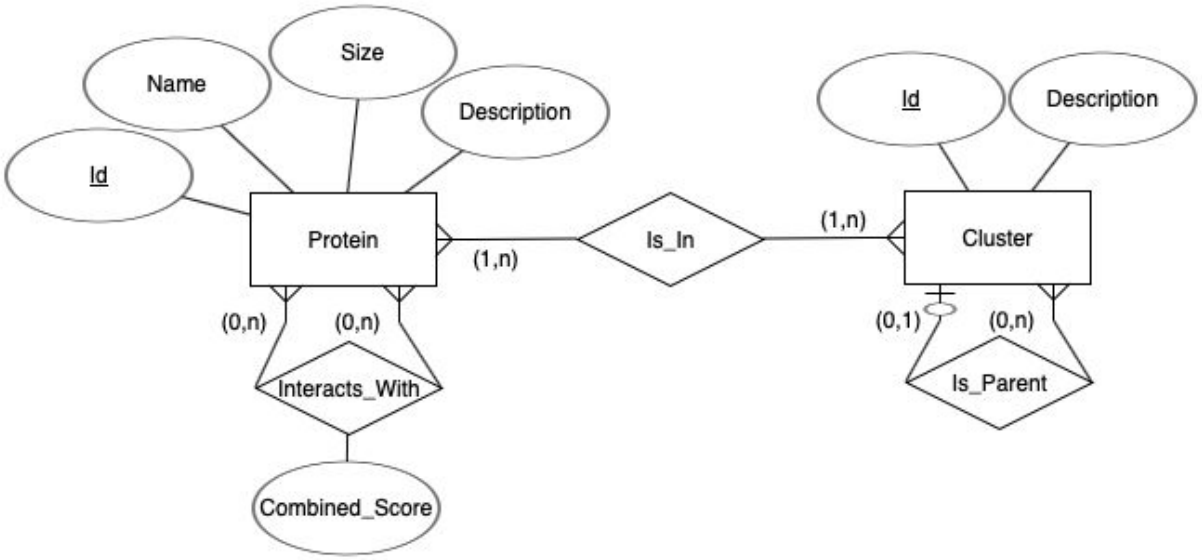
Les fichiers téléchargés ne seront pas importés tels quels dans la base de données SQL ou Cypher. J'enlève les données calculables par une requête sur la base de données comme par exemple la taille des clusters qui est calculable par une requête qui compte le nombre d'occurrence des id de clusters dans la table qui liste les protéines dans les clusters. J'enlève aussi les informations inutiles comme l'id taxonomique donné par le NCBI pour l'espèce humaine (9606) dans la base de données complète.

Dans le fichier initial de données sur les interactions entre protéines, il y a 16 colonnes de données. Certaines colonnes ont un nom contenant le mot "transferred" qui signifie "transféré d'une autre espèce". Étant donné que je ne travaille que sur l'espèce humaine, je choisis de ne pas importer ces colonnes dans ma base de données. Par ailleurs, la colonne neighborhood ne contient que des 0, donc je n'importe pas cette colonne non plus.

Les informations que je sélectionne pour le projet sont les informations sur les protéines humaines, les clusters de protéines, la hiérarchie entre les clusters (cluster parent et cluster enfant) et l'interaction entre protéines.

Le détail des informations retenues se trouve dans le modèle entité - relation ci-dessous.

Modèle Entité-Relation de la base de données



J'ai écrit le modèle en anglais car les données sont en anglais.

La légende du modèle entité - relation est la suivante :

- Une entité est dans un rectangle
- Un attribut d'une entité est dans une ellipse. Le souligné d'un attribut signifie que la valeur de cet attribut est unique.
- Une relation est dans un losange.

Les cardinalités des relations sont :

- Une protéine interagit avec 0 à plusieurs protéines
- Une protéine est dans un à plusieurs clusters
- Un cluster contient une à plusieurs protéines
- Un cluster est parent de zéro à plusieurs clusters
- Un cluster est enfant de zéro à un cluster

Remarques sur les données

La taille des protéines est exprimée en nombre d'acides aminés.
Le score combiné est une combinaison des scores des preuves biologiques d'interaction (fusion, homologie, cooexpression, cooccurrence) et des preuves d'interactions référencés dans des bases de données ou abstracts d'articles scientifiques (bases de données d'expériences, "curated databases", textmining d'abstract d'articles de recherche scientifique).
Les scores des preuves d'interaction (fusion, cooccurrence, homologie, cooexpression, score combiné, experiment, database, text mining) sont des entiers exprimés en pour 1000 (le score décimal a été multiplié par 1000 pour obtenir un score entier).
Les scores peuvent provenir de sources diverses :

- source "experiment" : une base de données expérimentale parmi BIND, DIP, GRID, HPRD, IntAct, MINT, et PID
- source "database" : une base de données de type "curated database" telle que Biocarta, BioCyc, GO, KEGG, et Reactome
- source "textmining" : abstract d'une publication scientifique

Implémentation de la base de données

Modélisation des données en SQL

Remarque : les clés primaires sont soulignées

La table Protein contient les colonnes suivantes : Id (type char(20)), Name (type varchar(15), not null), Size (type int, check(Size > 0), not null), Description (type text, not null).

La table Cluster contient les colonnes suivantes : Id (type varchar(8)), Description (type text, not null).

La table Is_In contient les colonnes suivantes : Protein_Id (type char(20), clé étrangère), Cluster_Id (type varchar(8), clé étrangère).

La table Interacts_With contient les colonnes suivantes : Protein1_Id (type char(20), clé étrangère), Protein2_Id (type char(20), clé étrangère), Combined_Score (type smallint, check(Combined_Score >= 0 and Combined_Score <= 1000), not null).

La table Is_Parent contient les colonnes suivantes : Child_Cluster_Id (type varchar(8), clé étrangère), Parent_Cluster_Id (type varchar(8), clé étrangère).

Création de la base de données SQL dans PostgreSQL

Créer les fichiers csv :

1. Se placer dans le répertoire src
2. Exécuter dans un terminal la commande “python3 txt_to_csv.py”. Cela prend 30 minutes.

Créer et remplir les tables :

1. Se connecter à psql
2. Créer une base de données nommée proteins à l’aide de la commande “CREATE DATABASE proteins;”
3. Sortir de psql avec la commande “exit”
4. Se placer dans le répertoire src
5. Exécuter les commandes sql du fichier tables.sql avec la commande “psql proteins -f ‘tables.sql’” . Cela prend 8 minutes.

Modélisation des données en Cypher

L’étiquette Protein a les propriétés suivantes : id (type String, unique, not null), Name (type String, not null), size (type Integer, not null), description (type String, not null).

L’étiquette Cluster a les propriétés suivantes : id (type String, unique, not null), description (type String, not null).

La relation Is_In se fait entre un nœud étiqueté Protein et un nœud étiqueté Cluster.

La relation Interacts_With se fait entre deux nœuds étiquetés Protein. Elle a la propriété combined_score (type Integer, not null).

La relation Is_Parent se fait entre deux nœuds étiquetés Cluster.

Création de la base de données Cypher dans Neo4J Desktop

1. Créer une base de données locale nommée Proteins
2. Copier les fichiers csv dans le dossier import de la base de données : Dans la fenêtre d'accueil de Neo4J Desktop, cliquez sur les 3 points de suspension en haut à droite de la tuile de la base de données Proteins, puis cliquez sur "Manage", puis cliquez sur "Open Folder" puis allez dans le dossier import et copiez dedans les fichiers csv.
3. Remplir la base de données. Cela prend 12 à 15 minutes. Dans la page d'accueil de Neo4J Desktop, cliquez sur "Start" dans la tuile de la base de données Proteins. Lorsque la base de données Proteins aura été démarrée, cliquez sur "Open", à gauche de "Start". Dans la fenêtre qui s'ouvre (Neo4J Browser), copiez-collez les instructions "CREATE CONSTRAINT" du fichier db.cql dans la barre de requête de Neo4J Brower et exécutez ces instructions. Faites de même avec chaque instruction "LOAD CSV" (commençant par ":auto" et se terminant par un point virgule) une par une.

J'aurais pu utiliser le programme cypher-shell qui se trouve dans le répertoire ./bin pour exécuter le code que j'ai écrit dans db.cql mais je n'ai pas réussi à le faire.

Stratégie d'import de données

SQL : écriture d'un script en langage Python pour créer des fichiers csv à partir des fichiers txt. Écriture d'instructions en SQL dans un fichier .sql pour créer les tables et les remplir par copie du contenu des fichiers csv (instruction \copy). Puis création de la base de données PostgreSQL dans un terminal. Ensuite, exécution d'une commande dans ce terminal pour exécuter les instructions SQL.

Cypher : Écriture d'un script en langage Python pour créer un fichier csv des informations des protéines (id, nom, taille, description) accepté par Cypher. En effet, Cypher n'accepte pas les chaînes de caractères qui contiennent des guillemets non échappés, à part les guillemets qui délimitent la chaîne de caractères. Puis écriture d'instructions Cypher dans un autre fichier pour créer les contraintes puis importer les données via l'instruction "LOAD CSV". Exécution des instructions Cypher dans Neo4J Browser par copier-coller de chaque instruction dans la barre de requête et raccourci clavier "cmd + enter" (équivalant : clic sur le triangle bleu à droite de la barre de requête de Neo4J Browser).

Requêtes sur la base de données Neo4J

Liste des requêtes

Requêtes nécessitant de travailler sur une étiquette ou une relation :

- protéines (3) : id, nom, description et taille de la protéine de taille minimale / maximale, taille moyenne d'une protéine
- hiérarchie de clusters (3) : le cluster qui n'a pas de cluster parent, le classement des ids de clusters par ordre décroissant de nombre de descendants directs et indirects, le nombre moyen de descendants de clusters

- interactions protéine-protéine (4) : les protéines qui interagissent avec aucune protéine, nombre moyen d'interactions entre protéines en ne prenant en compte que les interactions de la base de données, nombre moyen d'interactions entre protéines en prenant en compte toutes les protéines de la base de données, nombre d'interactions par niveau de confiance
- clusters de protéines (5) : les protéines dans aucun cluster, les protéines dans un seul cluster (on peut dire qu'elles sont spécifiques), les protéines qui sont dans le plus de clusters (elles ont plein de rôles, interviennent partout), classement (ordre décroissant) des clusters selon leur taille, taille moyenne des clusters

Requêtes nécessitant de travailler sur une étiquette et une relation (2) :

- classement (ordre décroissant) des clusters selon le nombre d'interactions intra-cluster / inter-cluster des protéines qu'ils contiennent, sans prendre en compte les interactions entre protéines de clusters liés par une relation cluster enfant - cluster parent

Formulation des requêtes, leurs résultats et temps de requêtes

Requêtes sur les protéines

Requête : id, nom, description et taille de la protéine de taille minimale

```
MATCH (p:Protein)
WITH min(p.size) as min_size
MATCH (p_min: Protein {size: min_size})
RETURN p_min.id, p_min.name, p_min.size, p_min.description;
```

Résultat : 2 protéines de taille minimale de 21 acides aminés

p_min.id	p_min.name	p_min.size	p_min.description
"9606.ENSP00000485197"	"ENSG00000279963"	21	"annotation not available"
"9606.ENSP00000485486"	"ENSG00000280329"	21	"annotation not available"

Temps de requête : Started streaming 2 records after 2 ms and completed after 48 ms.

Requête : id, nom, description et taille de la protéine de taille maximale

```
MATCH (p:Protein)
WITH max(p.size) as max_size
MATCH (p_max: Protein {size: max_size})
RETURN p_max.id, p_max.name, p_max.size, p_max.description;
```

Résultat : La protéine titine d'une taille de 35 991 acides aminés

p_max.id	p_max.name	p_max.size	p_max.description
"9606.ENSP00000467141"	"TTN"	35991	"Titin; Key component in the assembly and functioning of vertebrate striated muscles. By providing connections at the level of individual microfilaments, it contributes to the fine balance of forces between the two halves of the sarcomere. The size and extensibility of the cross-links are the main determinants of sarcomere extensibility properties of muscle. In non-muscle cells, seems to play a role in chromosome condensation and chromosome segregation during mitosis. Might link the lamina network to chromatin or nuclear actin, or both during interphase; Fibronectin type III domain containing"

Temps de requête : Started streaming 1 records after 14 ms and completed after 54 ms.

Requête : taille moyenne d'une protéine

MATCH (p:Protein) RETURN avg(p.size);
Résultat :

avg(p.size)
571.6900235101725

Temps de requête : Started streaming 1 records after 1 ms and completed after 40 ms.

Requêtes sur la hiérarchie des clusters

Requête : Les clusters sans cluster parent
MATCH (c:Cluster) WHERE NOT ()-[:Is_Parent]->(c) RETURN c.id, c.description
Résultat : 1 cluster d'id "CL:1" et de description "cell"
Temps de requête : Started streaming 1 records after 5 ms and completed after 117 ms.

Requête : Le classement des id de cluster avec leur nombre de descendants directs et indirects par ordre décroissant de nombre de descendants directs et indirects
MATCH p=(c:Cluster)-[:Is_Parent*]->() RETURN DISTINCT c.id, COUNT(p) as nb_desc ORDER BY nb_desc DESC
Résultat : 2808 clusters avec le nombre de clusters descendants (de 4810 à 1)
Temps de requête : Started streaming 2808 records after 2 ms and completed after 5 ms.

Requête : Le nombre moyen de descendants directs et indirects de clusters
MATCH p=(c:Cluster)-[:Is_Parent*]->() WITH c.id as cid, COUNT(p) as nb_desc RETURN AVG(nb_desc)
Résultat : 140.8493589743589
Temps de requête : Started streaming 1 records after 3 ms and completed after 6733 ms.

Requêtes sur les interactions protéine-protéine

Requête : Les protéines (id et nom) qui interagissent avec aucune protéine
MATCH (p:Protein) WHERE NOT (p)-[:Interacts_With]-(:Protein) RETURN p.id, p.name
212 Résultats. 5 premiers résultats :

p.id	p.name
"9606.ENSP00000242109"	"KIAA0087"
"9606.ENSP00000278779"	"C20orf78"
"9606.ENSP00000295896"	"C3orf49"
"9606.ENSP00000298298"	"C10orf25"
"9606.ENSP00000303234"	"C14orf183"

Temps de requête : Started streaming 212 records after 6 ms and completed after 768 ms.

Requête : Le nombre moyen (par protéine) d'interactions entre protéines en prenant en compte seulement les interactions entre protéines de la base de données
MATCH path=(p:Protein)-[:Interacts_With]-() WITH p.id as pid, COUNT(path) as nb_interactions RETURN AVG(nb_interactions)
Résultat : 1215.1962385036709
Temps de requête : Started streaming 1 records after 6 ms and completed after 20012 ms.

Requête : Le nombre moyen d'interactions entre protéines en prenant en compte toutes les protéines de la base de données

MATCH path=(p:Protein)-[:Interacts_With]-() WITH COUNT(path) as nb_interactions MATCH (p2:Protein) RETURN toFloat(nb_interactions) / COUNT(p2.id)
Résultat : 1202.029438822447
Temps de requête : Started streaming 1 records after 3 ms and completed after 16345 ms.

Requête : Le nombre d'interactions par niveau de confiance
Les niveaux de confiance sont : low confidence - 0.15 (or better), medium confidence - 0.4, high confidence - 0.7, highest confidence - 0.9.

```
MATCH ()-[iwl:Interacts_With]->() WHERE iwl.combined_score >= 150 AND iwl.combined_score < 400
WITH COUNT(iwl) as nb_low
MATCH ()-[iwm:Interacts_With]->() WHERE iwm.combined_score >= 400 AND iwm.combined_score < 700
WITH COUNT(iwm) as nb_medium, nb_low
MATCH ()-[iwh:Interacts_With]->() WHERE iwh.combined_score >= 700 AND iwh.combined_score < 900
WITH COUNT(iwh) as nb_high, nb_medium, nb_low
MATCH ()-[iwht:Interacts_With]->() WHERE iwht.combined_score >= 900
RETURN nb_low as nb_interactions_low_confidence, nb_medium as nb_interactions_medium_confidence, nb_high as nb_interactions_high_confidence, COUNT(iwht) as nb_interactions_highest_confidence
```

Résultat :

nb_interactions_low_confidence	nb_interactions_medium_confidence	nb_interactions_high_confidence	nb_interactions_highest_confidence
9758898	1159488	192764	648304

Temps de requête : Started streaming 1 records after 2 ms and completed after 527023 ms.

Remarque : Ce n'est pas plus rapide d'exécuter séparément une requête pour un niveau de confiance.

```
MATCH ()-[iwl:Interacts_With]->() WHERE iwl.combined_score >= 150 AND iwl.combined_score < 400
RETURN COUNT(iwl) as nb_interactions_low_confidence
```

Résultat :

nb_interactions_low_confidence
9758898

Temps de requête : Started streaming 1 records after 17 ms and completed after 1044223 ms.
Cette requête est deux fois plus lente que les 4 requêtes successives pipées entre elles plus haut

Requêtes sur les clusters de protéines

Requête : Les protéines (id, nom) dans aucun cluster

```
MATCH (p:Protein) WHERE NOT (p)-[:Is_In]->() RETURN p.id, p.name
```

213 résultat. 5 premiers résultats :

p.id	p.name

"9606.ENSP00000242109"	"KIAA0087"
"9606.ENSP00000278779"	"C20orf78"
"9606.ENSP00000295896"	"C3orf49"
"9606.ENSP00000298298"	"C10orf25"
"9606.ENSP00000303234"	"C14orf183"

Temps de requête : Started streaming 213 records after 4 ms and completed after 2068 ms.

Requête : Les protéines (id, name) dans un seul cluster (on peut dire qu'elles sont spécifiques)
MATCH (p:Protein)-[:Is_In]->(c:Cluster) WITH p, COUNT(c) as nb_clusters WHERE nb_clusters = 1 RETURN p.id, p.name

Résultat :
	p.id	p.name
"9606.ENSP00000477118"	"ENSG00000273047"
"9606.ENSP00000475146"	"C17orf50"
"9606.ENSP00000457511"	"CCDC179"
"9606.ENSP00000399075"	"C18orf42"
"9606.ENSP00000332389"	"BRICD5"
Temps de requête : Started streaming 5 records after 1105 ms and completed after 4735 ms.

Requête : Les protéines qui sont dans le plus grand nombre de clusters
(elles ont plein de rôles, interviennent dans plus de mécanismes biologiques)
MATCH (p:Protein)-[:Is_In]->(c:Cluster)
WITH p, COUNT(c) as nb_clusters
WITH MAX(nb_clusters) as max
MATCH (p:Protein)-[:Is_In]->(c:Cluster)
WITH p, COUNT(c) as nb_clusters, max
WHERE nb_clusters = max
RETURN p.id, p.name

15 Résultats : 5 premiers résultats :
	p.id	p.name
"9606.ENSP00000454836"	"TBL3"
"9606.ENSP00000423067"	"WDR36"
"9606.ENSP00000422392"	"LSM6"
"9606.ENSP00000364813"	"LSM2"
"9606.ENSP00000363746"	"WDR46"

Temps de requête : Started streaming 15 records after 11 ms and completed after 4098 ms.

Requête : Classement (ordre décroissant) des clusters (id) selon leur taille
MATCH (c:Cluster)<-[:Is_In]-(p:Protein)
WITH c, COUNT(p) as nb_prot
ORDER BY nb_prot DESC
RETURN c.id, nb_prot

Résultat : 4 811 résultats de clusters dont la taille va de 19353 pour le cluster d'id CL:1 à 5 pour le cluster d'id CL:10107

5 premiers résultats :

c.id	nb_prot
"CL:1"	19353
"CL:6"	19348
"CL:11"	19343
"CL:15"	19338
"CL:20"	19333

Temps de requête : Started streaming 4811 records after 3 ms and completed after 8 ms, displaying first 1000 rows.
Remarque : les protéines appartenant à plusieurs clusters se retrouvent plusieurs fois dans le comptage

Requête : Taille moyenne des clusters

```
MATCH (c:Cluster)<-[:Is_In]-(p:Protein)
WITH c, COUNT(p) as nb_prot
WITH AVG(nb_prot) as avg_nb_proteins
RETURN avg_nb_proteins;
```

Résultat : 341.25940552899584

Temps de requête : Started streaming 1 records after 3 ms and completed after 1407 ms.
Remarque : les protéines appartenant à plusieurs clusters se retrouvent plusieurs fois dans le comptage

Requêtes à la fois sur les clusters de protéines et les interactions protéine-protéine

Ces requêtes sont plus complexes car elles font intervenir la relation Interacts_With et un label. Il y a 11 millions de relations Interacts_With dans la base de données, ce qui est très volumineux. Les requêtes présentées dans cette partie n'ont pas pu aboutir dans un délai raisonnable (moins d'une heure).

Requête : Classement (ordre décroissant) des clusters selon le nombre d'interactions intra-cluster des protéines qu'ils contiennent, sans prendre en compte l'interaction d'une protéine dans un cluster avec une protéine dans un cluster descendant ou parent

Première proposition de requête

```
MATCH (p1:Protein)-[iw:Interacts_With]->(p2:Protein), (p1)-[:Is_In]->(c1:Cluster), (p2)-[:Is_In]->(c1)
WITH c1, COUNT(iw) as nb_interactions
ORDER BY nb_interactions DESC
RETURN c1.id as id_cluster, nb_interactions
```

Deuxième proposition de requête

```
MATCH (c:Cluster)
CALL {WITH c MATCH (p1:Protein)-[iw:Interacts_With]->(p2:Protein), (p1)-[:Is_In]->(c), (p2)-[:Is_In]->(c) RETURN COUNT(iw) as nb_interactions}
RETURN c.id as id_cluster, nb_interactions
ORDER BY nb_interactions DESC
```

Comparaison des plans d'exécution des deux propositions

La deuxième proposition de requête est moins volumineuse en termes de nombre de lignes estimées que la première proposition : 4000 lignes estimées au maximum dans une étape contre 560 millions pour la première proposition. Cette différence de nombre de lignes estimées indique une plus grande consommation de mémoire de la première proposition. Elle provient du fait que dans la première proposition, on garde en mémoire tout le chemin (c1)<-[:Is_In]-(p1)-[:Interacts_With]->(p2)-[:Is_In]->(c1) pour faire le comptage d'interactions protéine-protéine intra-cluster pour chaque cluster alors que dans la deuxième proposition on effectue ce comptage sur les chemins spécifiques au cluster c1 importé par la clause WITH dans la sous-requête de comptage.
Son plan d'exécution est également moins complexe (2 branches au lieu de 3 pour la première proposition) et est composé de moins d'étapes (13 étapes au lieu de 17 pour la première proposition).
Son plan d'exécution se distingue également par l'absence de l'opération NodeHashJoin, présente à deux reprises dans le plan d'exécution de la première proposition. Ces deux jointures coûteuses en mémoire sont remplacées par deux opérations Expand, un Expand(All) et un Expand(Into), dans le plan d'exécution de la deuxième proposition. L'opération Expand(All) parcourt toutes les relations [:Is_In] entrantes sur le cluster commun de p1 et p2 afin de trouver p1 et cette même opération est répétée pour trouver p2. L'opération Expand(Into) parcourt toutes les relations entre deux nœuds donnés. Dans la deuxième proposition, elle parcourt toutes les relations entre p1 et p2 pour retrouver la relation Interacts_With. Une opération EagerAggregation pour compter le nombre de relations Interacts_With (et donc le nombre d'interactions) est effectuée sur le résultat de ce Expand(Into) dans la deuxième proposition (donc seulement sur les noeuds p1, p2, c1 actuellement traités) alors qu'elle est effectuée sur une table de jointure (issue de la deuxième opération NodeHashJoin) contenant tous les noeuds p1, p2 et c1 (toutes les protéines interagissant entre elles et étant dans le même cluster) pour la première proposition. En ce qui concerne la première opération de jointure NodeHashJoin de la première proposition, elle est effectuée sur les chemins (p1)-[:Is_in]->(c1) et (p2)-[:Is_in]->(c1) qui ont été retrouvés indépendamment l'un de l'autre donc à ce moment on fait une jointure sur toutes les combinaisons de chemins (p1)-[:Is_in]->(c1) et (p2)-[:Is_in]->(c1) retrouvés en parcourant deux fois la "liste" des relations Interacts_With. Cela est différent dans la deuxième proposition où le chemin (p1)-[:Is_in]->(c1) et le chemin (p2)-[:Is_in]->(c1) sont retrouvés l'un après l'autre, en utilisant le même noeud c1, ce qui fait que seuls les relations Is_in du cluster c1 sont parcourues pour trouver p1 et p2.

Le plan d'exécution de la première proposition de requête est le fichier plan1.png du répertoire plans_execution. Celui de la deuxième proposition de requête est le fichier plan2.png du répertoire plans_execution.

Requête : Classement (ordre décroissant) des clusters selon le nombre d'interactions inter-cluster des protéines qu'ils contiennent, sans prendre en compte l'interaction d'une protéine dans un cluster avec une protéine dans un cluster descendant ou parent

```
MATCH (c:Cluster)
CALL {WITH c MATCH (p1:Protein)-[iw:Interacts_With]->(p2:Protein), (p1)-[:Is_In]->(c) WHERE NOT EXISTS((p2)-[:Is_In]->(c)) RETURN COUNT(iw) as nb_interactions}
RETURN c.id as id_cluster, nb_interactions
ORDER BY nb_interactions DESC
```

Cette requête utilise une sous-requête comme dans la deuxième proposition pour la requête précédente. Ici, la condition du WHERE indique qu'on garde seulement les match (c)<-[:Is_In]-(p1)-[:Interacts_With]->(p2) tels qu'il n'y a pas de relation (p2)-[:Is_in]->(c). Cette formulation évite d'utiliser une variable c2 pour un deuxième cluster et ensuite mettre NOT c2.id = c.id comme condition dans le WHERE. Le choix de ne pas introduire de deuxième variable de cluster permet de ne pas demander à Neo4J d'aller parcourir toutes les relations IS_In pour garder les clusters c2 différents de c. Ici, nous vérifions simplement l'absence de relation entre p2 et c pour en déduire que les matchs gardés après l'application du filtrage par WHERE sont des match où le cluster de p1 est différent du cluster de p2.
Cependant, malgré cette optimisation, le nombre de lignes énumérées lors de l'exécution de la requête reste très grand étant donné le nombre de relations Is_in et le nombre de relations Interacts_With.

Le plan d'exécution de cette requête est le fichier plan3.png du répertoire plans_execution.

Efficacité des requêtes sur la base de données

Comparaison de quelques plans d'exécution avec et sans index

Étant donné que les seuls index de ma base de données Neo4J sont des index intrinsèques de contraintes d'unicité, je dois supprimer ces contraintes pour retirer les index.

Comparaison de plans d'exécutions avec/sans index sur l'id des protéines

Je prends ici l'exemple de la requête des protéines de taille maximale.

```
MATCH (p:Protein)
WITH max(p.size) as max_size
MATCH (p_max: Protein {size: max_size})
RETURN p_max.id, p_max.name, p_max.size, p_max.description;
```

Sans index, le parcours des nœuds se fait avec l'opération NodeByLabelScan alors qu'avec index elle se fait avec l'opération NodeIndexScan. Dans le premier cas, le scan se fait sur les nœuds en suivant l'index "node label index" de Neo4J et on cherche les nœuds qui ont le label cherché, ici le label Protein. Dans le deuxième cas, le scan se fait sur toutes les valeurs stockées dans un index et retourne tous les nœuds avec le label et la propriété recherchés. Ici, l'index est sur les id de protéines donc le scan parcourt les protéines en cherchant celles qui ont un id (équivalent de rechercher (p:Protein) WHERE EXISTS (id)). Il est intéressant de remarquer que l'index des id de protéines est utilisé lors de l'évaluation de la partie MATCH (p: Protein) alors que l'id n'est pas mentionné.

Cette vérification supplémentaire d'existence d'id pour chaque protéine de l'index d'id de protéine, n'a pas une grande conséquence sur le temps d'exécution de la requête. La requête sans index se fait en 43 ms (moyenne sur 5 exécutions entre 27 ms et 63 ms) alors que la requête avec index se fait en 40 ms (moyenne sur 5 exécutions entre 35 ms et 46 ms).

Le plan d'exécution de la requête avec index est le fichier plan4.png du répertoire plans_execution.
Le plan d'exécution de la requête sans index est le fichier plan5.png du répertoire plans_execution.

Comparaison de plans d'exécutions avec/sans index sur l'id des clusters

Je prends ici l'exemple de la requête de taille moyenne d'un cluster en nombre de protéines.

```
MATCH (c:Cluster)<-[:Is_In]-(p:Protein)
WITH c, COUNT(p) as nb_prot
WITH AVG(nb_prot) as avg_nb_proteins
RETURN avg_nb_proteins
```

Avec index, le plan d'exécution de la requête contient une opération NodeByLabelScan pour retrouver le nœud Cluster des relations Is_In matchées. L'index n'est ici pas utilisé car les clusters ne sont pas le cœur de la requête : le cœur de la requête est la relation Is_In et le cluster intervient comme extrémité d'une relation Is_in et comme variable permettant l'agrégation du comptage de nombre de protéines (le comptage est agrégé par cluster). Cette requête est intéressante à étudier car elle nous apprend que l'index n'est pas toujours utilisé bien qu'on regarde des clusters.
Avec index, le temps moyen d'exécution de cette requête est de 994 ms (moyenne sur 10 exécutions entre 742 ms et 1488 ms).

Le plan d'exécution de la requête sans index sur l'id des clusters est identique au plan d'exécution de la requête avec index sur l'id des clusters. Ce n'est pas étonnant puisque cet index n'est pas utilisé pendant l'exécution de cette requête.

Le plan d'exécution de la requête est le fichier plan6.png du répertoire plans_execution.

Comparaison de requêtes récursives en SQL par rapport à leur équivalent en Cypher

Je prends ici l'exemple de la requête qui donne le classement des clusters par ordre décroissant de nombre de clusters descendants directs (clusters enfants) et indirects : son résultat est le nombre de descendants directs et indirects de chaque id de cluster.

Sa formulation en langage Cypher est la suivante :

```
MATCH p=(c:Cluster)-[:Is_Parent*]->() RETURN DISTINCT c.id, COUNT(p) as nb_desc ORDER BY nb_desc DESC
```

Sa formulation en langage SQL est la suivante :

```
WITH RECURSIVE all_parents(Parent_Cluster_Id, Child_Cluster_Id) AS (
SELECT Parent_Cluster_Id, Child_Cluster_Id
FROM Is_Parent
UNION ALL
SELECT ip.Parent_Cluster_Id, ip.Child_Cluster_Id
FROM Is_Parent ip, all_parents p
WHERE p.Parent_Cluster_Id = ip.Child_Cluster_Id
)
SELECT Parent_Cluster_Id, COUNT(Child_Cluster_Id) as nb_childs
FROM all_parents
GROUP BY Parent_Cluster_Id
ORDER BY nb_childs DESC;
```

Le temps d'exécution de la requête en Cypher est de 30 ms et peut descendre jusqu'à 5 ms en moyenne (moyenne sur 10 exécutions successives avec des temps d'exécution allant de 1 à 12

ms) lorsqu'elle est exécutée souvent. Le temps d'exécution de la requête en SQL est de 407,5791 ms en moyenne (moyenne sur 10 exécutions successives avec des temps d'exécution allant de 381,268 ms à 492,128 ms). La requête en Cypher met donc dix fois moins de temps pour s'exécuter que son équivalent en SQL.

Pour tenter de comprendre pourquoi la requête en SQL prend tant de temps, j'affiche son plan d'exécution :

```
 Sort  (cost=39565.19..39565.69 rows=200 width=42) (actual time=688.038..688.372 rows=2808 loops=1)
   Sort Key: (count(all_parents.child_cluster_id)) DESC
   Sort Method: quicksort  Memory: 290kB
   CTE all_parents
     ->  Recursive Union  (cost=0.00..27410.30 rows=485810 width=16) (actual time=0.074..332.022
rows=395505 loops=1)
           ->  Seq Scan on is_parent  (cost=0.00..78.10 rows=4810 width=16) (actual time=0.063..1.048
rows=4810 loops=1)
           ->  Hash Join  (cost=138.22..1761.60 rows=48100 width=16) (actual time=0.027..1.666 rows=3101
loops=126)
                 Hash Cond: ((p.parent_cluster_id)::text = (ip.child_cluster_id)::text)
                 ->  WorkTable Scan on all_parents p  (cost=0.00..962.00 rows=48100 width=34) (actual
time=0.000..0.388 rows=3139 loops=126)
                 ->  Hash  (cost=78.10..78.10 rows=4810 width=16) (actual time=3.186..3.186 rows=4810 loops=1)
                       Buckets: 8192  Batches: 1  Memory Usage: 296kB
                       ->  Seq Scan on is_parent ip  (cost=0.00..78.10 rows=4810 width=16) (actual
time=0.012..0.624 rows=4810 loops=1)
   ->  HashAggregate  (cost=12145.25..12147.25 rows=200 width=42) (actual time=686.227..686.855
rows=2808 loops=1)
         Group Key: all_parents.parent_cluster_id
         Batches: 1  Memory Usage: 385kB
         ->  CTE Scan on all_parents  (cost=0.00..9716.20 rows=485810 width=68) (actual time=0.076..551.830
rows=395505 loops=1)
 Planning Time: 3.740 ms
 Execution Time: 690.373 ms
(18 rows)
```

Je remarque que le coût de la requête est réparti comme suit : 27410 pour le Recursive Union (la partie union récursive de la requête) (soit ⅔ du coût total) et 12147 pour le HashAggregate (la partie agrégation par comptage des clusters enfants à agréger par id de cluster parent/ancêtre) (soit ⅓ du coût total). Dans la partie Recursive Union, le hash join représente une grande partie du temps d'exécution. Dans la partie Hash Aggregate, c'est le scan de tous les résultats de la table temporaire all_parents qui concentre la majeure partie du temps d'exécution.

Je ne sais pas comment est réparti le coût des différentes parties de l'exécution de la requête en Cypher, puisque nous n'avons que les informations sur le nombre de lignes estimées et les opérations qui sont faites. J'ai cependant remarqué l'absence de jointure dans le plan d'exécution de la requête en Cypher et j'ai aussi remarqué qu'il y a un seul scan dans le plan d'exécution (scan des clusters). Le plan d'exécution en Cypher est plus simple :

1. NodeIndexScan sur les clusters pour parcourir tous les clusters (4810 clusters)
2. VarLengthExpand(All) pour étendre tous les chemins partant d'un cluster ancêtre à ses clusters descendants. Ces chemins étant de longueur variable.
3. EagerAggregation pour agréger par comptage COUNT du nombre de nœuds dans les chemins (ce qui donne le nombre de descendants) et renvoyer des couples (id de cluster, nombre de descendants)

4. Sort : tri des couples (id de cluster, nombre de descendants) par ordre décroissant du nombre de descendants de chaque cluster
5. ProduceResult : production du résultat : des lignes de couples (id de cluster, nombre de descendants) ordonnés dans l'ordre décroissant du nombre de descendants de chaque cluster
6. Résultat

Ce plan d'exécution est le fichier plan7.png du répertoire plans_execution.

Exemples de requêtes plus efficaces en SQL qu'en Cypher

Je n'ai pas réussi à trouver une requête sur les données plus efficace et rapide en SQL qu'en Cypher. Les requêtes de matching de motif ou de nœud sont généralement plus rapides que les requêtes de sélection en SQL.

J'ai réussi à trouver un cas où PostgreSQL fait mieux que Cypher : lorsqu'on ajoute une propriété à une relation (Cypher) ou une table (PostgreSQL) et que la relation ou table contient déjà un grand nombre de valeurs. Ici je vais modifier dans PostgreSQL la table Interacts_With pour ajouter la colonne Homology de type SMALLINT avec les contraintes "CHECK (Homology >= 0 AND Homology <= 1000)" et "DEFAULT 0 NOT NULL" puis copier dans chaque ligne la valeur de la colonne Combined_Score dans la colonne Homology . Je modifie la base de données Neo4J en ajoutant la propriété homology à la relation Interacts_With.

En PostgreSQL, la requête de création de colonne "ALTER TABLE interacts_with ADD COLUMN Homology SMALLINT CHECK(Homology >= 0 AND Homology <= 1000) DEFAULT 0 NOT NULL;" s'exécute en 2655, 096 ms (2 minutes) et la requête de copie de colonne "UPDATE Interacts_With SET Homology = Combined_Score;" s'exécute en 289 858, 694 ms (4 minutes).

Avec Neo4J, la requête de copie de valeur "MATCH (:Protein)-[iw:Interacts_With]->(:Protein) SET iw.homology = iw.combined_score return iw" fait planter la fenêtre Neo4J Brower (elle devient toute grise) et elle consomme beaucoup de mémoire dans mon ordinateur. Je dois utiliser la fonction de periodic rock n roll (mise à jour des données par batch) de la bibliothèque APOC pour faire fonctionner la requête : "CALL apoc.periodic.rock_n_roll("match (:Protein)-[iw:Interacts_With]->(:Protein) RETURN id (iw) as id_iw", "MATCH (:Protein)-[iw]->(:Protein) WHERE id(iw) = $id_iw SET iw.homology = iw.combined_score", 1000)". Cette procédure est d'ailleurs dépréciée et sera supprimée dans une version ultérieure de la bibliothèque APOC. Cette requête prend beaucoup plus de temps à s'exécuter que dans PostgreSQL. Au bout de deux heures, la requête dans Neo4J n'est toujours pas terminée, même avec le traitement par batch. Après arrêt de cette requête et comptage des homologies, je m'aperçois que 4 508 000 propriétés d'homologie ont été ajoutées, donc la requête d'ajout devrait prendre 5 heures pour s'exécuter.

Pour la suppression, c'est la même situation. PostgreSQL exécute très rapidement la suppression de la colonne Homology contrairement à Neo4J. Avec les deux SGBD la requête de suppression s'exécute plus rapidement que la requête d'ajout. Ainsi PostgreSQL exécute la requête "ALTER TABLE Interacts_With DROP COLUMN Homology;" de manière quasi instantanée en 934,911 ms (soit moins d'une seconde). À titre de comparaison, Cypher exécute la requête "MATCH (:Protein)-[iw:Interacts_With]->(:Protein) REMOVE iw.homology" pour supprimer 230 000 scores d'homologies en 54 110 ms soit 50 fois plus lentement que SQL. Supprimer les scores d'homologies sur les 11 millions d'arêtes Interacts_With prendrait donc 2 705,5 secondes, soit 45 minutes.

Je suppose que la différence est due au fait que dans Neo4J sans utiliser APOC, pour ajouter ou supprimer une propriété d'une relation il faut d'abord parcourir l'index des nœuds de la relation : soit l'index de tous les nœuds NodeIndex si les extrémités de la relation ne sont pas spécifiées (c'est un NodeIndexScan) ou dans mon cas, l'index des nœuds ayant le label désiré/cherché (c'est un NodeByLabelScan). Ensuite il y a une opération d'expansion (Expand) pour trouver la relation Interacts_With et enfin ajouter/supprimer la propriété homology. Tout ceci consomme beaucoup de mémoire.

Je conclus donc que lorsque nous travaillons avec un grand volume de données, Cypher est plus rapide que PostgreSQL pour effectuer des requêtes de matching de motif sur les données, alors que PostgreSQL est infiniment plus efficace que Cypher pour "administrer" les données en ajoutant ou supprimant des colonnes.

Analyse de graphe via la Data Science Graph Library

Mon analyse de graphe avec la bibliothèque Data Science Graph Library est très limitée en raison du grand nombre d'interactions entre protéines (11 millions, dont 600 000 dans le plus haut niveau de confiance) et de nombre de protéines dans au moins un cluster (1 million de relations protéine - cluster).

Les protéines contenues dans le plus grand nombre de clusters

Dans un premier temps, je choisis de me concentrer sur l'étude des protéines contenues dans le plus grand nombre de clusters.

Id des quinze protéines contenues dans le plus grand nombre de clusters : ["9606.ENSP00000454836", "9606.ENSP00000423067", "9606.ENSP00000422392", "9606.ENSP00000364813", "9606.ENSP00000363746", "9606.ENSP00000363642", "9606.ENSP00000355541", "9606.ENSP00000327179", "9606.ENSP00000300413", "9606.ENSP00000297990", "9606.ENSP00000261708", "9606.ENSP00000264279", "9606.ENSP00000252622", "9606.ENSP00000249299", "9606.ENSP00000225298"]

Je commence avec l'algorithme Common Neighbors pour calculer un score de proximité entre ces protéines, basé sur le nombre de voisins communs. Cet algorithme me semble adapté à mon graphe car de manière intuitive, plus les protéines interagissent entre elles (nombre d'interactions à haut score), plus elles sont proches.

J'effectue la requête suivante :

```
WITH ["9606.ENSP00000423067", "9606.ENSP00000422392", "9606.ENSP00000364813", "9606.ENSP00000363746", "9606.ENSP00000363642", "9606.ENSP00000355541", "9606.ENSP00000327179", "9606.ENSP00000300413", "9606.ENSP00000297990", "9606.ENSP00000261708", "9606.ENSP00000264279", "9606.ENSP00000252622", "9606.ENSP00000249299", "9606.ENSP00000225298"] as list
WITH list, apoc.coll.combinations(list, 2, 2) AS pairs
UNWIND pairs as pair
MATCH (p1: Protein {id: pair[0] }), (p2: Protein {id: pair[1]})
RETURN p1.id, p2.id, gds.alpha.linkprediction.commonNeighbors(p1, p2) AS score ORDER BY score DESC
```

Les scores calculés sont variables : les protéines d'id "9606.ENSP00000364813" et "9606.ENSP00000300413" ont le score le plus haut de 1108.0 et le score le plus bas de 305.0 est obtenu pour les protéines d'id "9606.ENSP00000252622" et "9606.ENSP00000423067" ainsi que les protéines d'id "9606.ENSP00000327179" et "9606.ENSP00000252622.

Je calcule également les scores de proximité entre ces protéines en prenant en compte seulement les relations Interacts_With.

J'effectue la requête suivante :

```
WITH ["9606.ENSP00000423067", "9606.ENSP00000422392", "9606.ENSP00000364813", "9606.ENSP00000363746", "9606.ENSP00000363642", "9606.ENSP00000355541", "9606.ENSP00000327179", "9606.ENSP00000300413", "9606.ENSP00000297990", "9606.ENSP00000261708", "9606.ENSP00000264279", "9606.ENSP00000252622", "9606.ENSP00000249299", "9606.ENSP00000225298"] as list
WITH list, apoc.coll.combinations(list, 2, 2) AS pairs
UNWIND pairs as pair
MATCH (p1: Protein {id: pair[0] }), (p2: Protein {id: pair[1]})
RETURN DISTINCT p1.id, p2.id, gds.alpha.linkprediction.commonNeighbors(p1, p2, {relationshipQuery: "Interacts_With"}) AS score ORDER BY score DESC
```

Les scores calculés sont variables et plus faibles que les scores précédents : les protéines d'id "9606.ENSP00000364813" et "9606.ENSP00000300413" ont le score le plus haut de 981.0 et le score le plus bas de 203.0 est obtenu pour les protéines d'id "9606.ENSP00000252622" et "9606.ENSP00000423067" ainsi que les protéines d'id "9606.ENSP00000327179" et "9606.ENSP00000252622.

Si je ne prends en compte que les relations Interacts_With pour calculer ces scores, ces scores sont légèrement plus faibles (entre 10 à 30 % plus faibles). Je vais appeler le score calculé en prenant toutes les relations en compte le score 1 et le score calculé en prenant en compte seulement les relations Interacts_With le score 2. Étant donné qu'il y a 10 fois plus d'interactions Interacts_With que de relations Is_In, plus le score 1 est haut plus il dépend des interactions entre protéines et moins il dépend des autres relations (Is_In et Is_Parent), donc le score 2 sera peu diminué (10 %) par rapport au score 1. À l'inverse, plus ce score 1 est bas, plus il dépend des interactions autres que Interacts_With et plus le score 2 est faible par rapport au score 1.

Je m'intéresse maintenant aux interactions de score maximal 999 de ces protéines. Il y a 273 interactions concernées.

J'effectue la requête suivante :

```
MATCH (p:Protein)-[:Is_In]->(c:Cluster)
WITH p, COUNT(c) as nb_clusters
WITH MAX(nb_clusters) as max
MATCH (p:Protein)-[:Is_In]->(c:Cluster)
WITH p, COUNT(c) as nb_clusters, max
WHERE nb_clusters = max
MATCH (p1:Protein {id:p.id})-[iwht:Interacts_With]->(p2:Protein) WHERE iwht.combined_score = 999
RETURN p1.name, p2.name, gds.alpha.linkprediction.resourceAllocation(p1, p2, {relationshipQuery: "Interacts_With"}) AS score ORDER BY score DESC
```

Le score calculé est le score de proximité entre les protéines de ces 273 interactions, calculé par l'algorithme resourceAllocation. Cet algorithme calcule un score de proximité en se basant sur le voisinage partagé entre des nœuds.

Le score est variable : le score maximum est de 0,8137 entre les protéines nommées "NOP58" et "NOP56", et le score minimum est de 0,3075 entre les protéines nommées "UTP18" et "UTP3". Il est intéressant de noter que même si deux protéines ont un nom proche, cela ne signifie pas que leur similarité est grande. Je note aussi que malgré le score maximal d'interaction, cela ne signifie pas que les protéines soient considérées comme proches du point de vue de leur "voisinage". Cela signifie que deux protéines qu'on peut considérer comme interagissant entre elles car elles ont score maximal de confiance sur l'interaction n'interagissent pas forcément avec les mêmes autres protéines.

J'étudie la centralité dans le sous-graphe composé de ces protéines. Étant donné que les algorithmes de la bibliothèque Data Science Graph Library ne prenne pas en compte les poids des arêtes en considérant que plus le poids est fort mieux c'est, je devrais exécuter ces algorithmes sur un sous-graphe ayant toutes les arêtes au même score de confiance d'interaction, soit ici le score maximal et en prenant en compte seulement les protéines présentes dans le plus grand nombre de clusters. Ces protéines sont intuitivement le centre du graphe. L'idée est alors de tester plusieurs algorithmes de centralité afin de déterminer le centre dans ce "centre". Je choisirais l'algorithme de closeness centrality, qui calcule la proximité selon le nombre de plus courts chemins puis en inversant le score trouvé et d'après la documentation de cet algorithme sur le site de Neo4J cet algorithme fonctionne au moins sur un graphe non valué (ce qui est mon cas si je ne choisis que des arêtes de même poids). J'aurais aimé appliquer le betweenness centrality (qui fonctionne également sur un graphe non valué) sur toutes les protéines pour trouver les "ponts" entre les interactions de niveau de confiance maximal. Par contre cet algorithme nécessite une certaine quantité de mémoire et il faut vérifier que l'on a une quantité suffisante de mémoire avant de lancer l'algorithme.
Je n'ai cependant pas pu faire cette étude à cause d'un problème technique sur ma base de données : pour la 3ème fois dans ce projet, j'ai perdu la connexion sur ma base de données au cours d'une session et je ne peux plus m'y connecter et je n'ai pas le temps de recréer la base de données et faire cette étude.

Les protéines qui ne sont dans aucun cluster

Je vais maintenant étudier les 213 protéines qui ne sont dans aucun cluster.

Requête calculant un score de proximité (score basé sur le nombre de voisins en commun) entre des paires de protéines sans cluster et classant le score par ordre décroissant :

```
MATCH (p:Protein) WHERE NOT (p)-[:Is_In]->()
WITH collect(p) as proteins
WITH apoc.coll.combinations(proteins, 2, 2) AS pairs
UNWIND pairs as pair
RETURN pair[0].name, pair[1].name, gds.alpha.linkprediction.commonNeighbors(pair[0], pair[1]) AS score ORDER BY score DESC
```

Cette requête donne un score de zéro entre toutes les paires de protéines sans cluster. Ce qui signifie que les protéines de chaque paire n'ont aucun voisin en commun. Puisque ces protéines

ne sont dans aucun cluster, je sais déjà qu'elles ne sont pas liées par des relations Is_In ni que leurs clusters sont liés par une relation Is_Parent. Maintenant je sais aussi que ces protéines n'ont absolument rien en commun via les interactions qu'elles pourraient avoir.

J'effectue une requête pour calculer le nombre d'interactions de ces protéines. La formulation de la requête est la suivante :

```
MATCH (p:Protein) WHERE NOT (p)-[:Is_In]->()
WITH collect(p) as proteins
UNWIND proteins as protein
MATCH (protein)-[iw:Interacts_With]->(p2:Protein)
RETURN protein.name, COUNT(iw) as nb_interactions
```

Le résultat est que seule la protéine nommée SMIM10 a une interaction avec une autre protéine. Puisque les scores de proximité entre les protéines sans cluster est de 0, je peux en déduire que cette autre protéine est dans un cluster. Le résultat de cette requête me permet également de déduire que les protéines sans cluster ne sont pas considérées comme proches d'autres protéines car à l'exception de SMIM10, elles n'interagissent pas avec d'autres protéines.

Mon intuition est que ce sont probablement des protéines de structure (par exemple, une protéine qui interviennent dans la structure physique d'une cellule). La requête "MATCH (p:Protein) WHERE NOT (p)-[:Is_In]->() RETURN distinct p.description" m'apprend que la plupart sont des protéines non caractérisées ou des protéines de structure (pore nucléaire, membrane plasmique).