



Licence MIAGE

Rapport de projet Graphes et OpenData

les gens avec beaucoup d'amis connectent-ils des groupes différents ?

Projet réalisé du 18 Decembre 2025 au 13 avril 2025

Membres du groupe

**SEIDE Cardly
ASLOUN Othman**

Table des matières

1	Introduction	3
2	Environnement de travail	3
3	Description du projet et objectifs	3
3.1	Jeu de données	3
3.2	Traitement des données	4
3.2.1	Transfomation en fichier csv	4
3.2.2	Suppression des boucles	4
3.2.3	Prise en compte uniquement des relation réciproque	4
3.3	Algorithme de parcours dans le graphe	5
3.3.1	Random Walk Sampling	5
3.3.2	Induced Subgraph from High-Degree Nodes	5
3.3.3	Breadth-First Search (BFS) Sampling	6
3.4	Identification des communautés	8
3.5	Détermination des ponts reliant plusieurs communautés	8
3.6	Identification des noeuds relais	8
3.7	Interprétation	8
4	Bibliothèques, Outils et technologies	9
5	Travail réalisé	9
5.1	Script Python permettant de nettoyer nos données	9
5.2	algorithmes de parcours	9
6	Détection de communautés	10
6.1	Résultats aperçus	12
6.2	Analyse des liens inter-communautés et des noeuds ponts	13
6.3	Identification et visualisation des connexions clés entre les communautés	15
6.4	Identification des rôles et visualisation d'un sous-graphe optimisé	18
7	Difficultés rencontrées	21
7.1	difficultés de traitement des données	21
7.2	Difficultés matérielles	21
7.2.1	génération du graphe	21
7.2.2	contrainte logicielle	21
8	Bilan	21
8.1	Conclusion	21
8.2	Perspectives	22

1 Introduction

D'après certaines études en sociologie, chaque individu appartient en moyenne à au moins sept groupes sociaux différents : famille, amis proches, collègues, communautés en ligne, clubs, etc. Aujourd'hui, dans un monde plus interconnecté que jamais, ces appartenances multiples se reflètent dans les réseaux sociaux numériques, où les utilisateurs peuvent faire le lien entre des sphères sociales très diverses. Dans ce contexte, une question se pose naturellement : certains individus peuvent-ils servir de ponts entre différentes communautés ?

Dans ce projet, nous analyserons le réseau social LiveJournal, une plateforme qui permettait aux utilisateurs de créer des blogs et de se connecter entre eux. Le jeu de données utilisé contient plusieurs millions de connexions entre utilisateurs, sous forme de relations d'amitié. Notre objectif principal est de répondre à la question suivante : Les personnes ayant beaucoup d'amis jouent-elles un rôle de connecteurs entre différentes communautés ?

Pour y répondre, nous avons mis en œuvre plusieurs étapes allant du nettoyage et traitement des données, à la détection de communautés et à l'analyse des nœuds à fort degré.

2 Environnement de travail

Python 3.11 : langage principal utilisé pour le traitement des données, la génération de sous-graphes et la visualisation.

VS Code : environnements de développement utilisés pour coder, tester et documenter les scripts.

Pandas : bibliothèque pour le traitement des fichiers CSV, manipulation de données tabulaires.

NetworkX : bibliothèque pour la création, la manipulation et l'analyse de graphes.

Matplotlib : utilisée pour la visualisation des graphes.

Neo4j Desktop : base de données orientée graphe utilisée pour importer les données et effectuer des requêtes Cypher.

3 Description du projet et objectifs

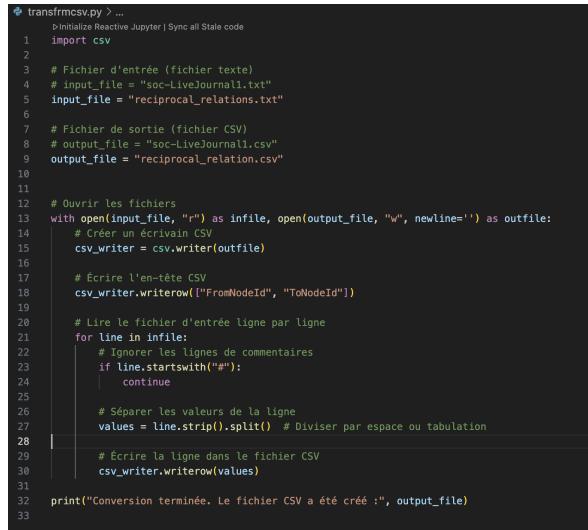
3.1 Jeu de données

Nous avons utilisé la base de données soc-LiveJournal1.txt, qui provient de SNAP (Stanford Network Analysis Project), une initiative de l'Université de Stanford. Cette base de données représente un réseau social en ligne, LiveJournal, et contient des informations sur les relations d'amitié entre les utilisateurs. Elle se compose de 4 847 571 nœuds (utilisateurs) et 68 993 773 arêtes (relations d'amitié).

3.2 Traitement des données

3.2.1 Transfomation en fichier csv

Nous avons utilisé un script python (transfrmcsv) pour convertir le fichier initial au format .txt en un fichier .csv, dans le but de simplifier son traitement et d'en faciliter l'analyse.



```
# transfrmcsv.py > ...
#-Initialise Reactive Jupyter | Sync all Stale code
1 import csv
2
3 # Fichier d'entrée (fichier texte)
4 # input_file = "soc-LiveJournal1.txt"
5 input_file = "reciprocal_relations.txt"
6
7 # Fichier de sortie (fichier CSV)
8 # output_file = "soc-LiveJournal1.csv"
9 output_file = "reciprocal_relation.csv"
10
11
12 # Ouvrir les fichiers
13 with open(input_file, "r") as infile, open(output_file, "w", newline='') as outfile:
14     # Créer un écrivain CSV
15     csv_writer = csv.writer(outfile)
16
17     # Écrire l'en-tête CSV
18     csv_writer.writerow(["FromNodeId", "ToNodeId"])
19
20     # Lire le fichier d'entrée ligne par ligne
21     for line in infile:
22         # Ignorer les lignes de commentaires
23         if line.startswith("#"):
24             continue
25
26         # Séparer les valeurs de la ligne
27         values = line.strip().split() # Diviser par espace ou tabulation
28
29         # Écrire la ligne dans le fichier CSV
30         csv_writer.writerow(values)
31
32 print("Conversion terminée. Le fichier CSV a été créé :", output_file)
33
```

FIGURE 1 – Script transformation en CSV

3.2.2 Suppression des boucles

Nous avons utilisé cette fonction proposée par networkX afin d'enlever les boucles du graphe.

```
# Supprimer les boucles
G.remove_edges_from(nx.selfloop_edges(G))
```

FIGURE 2 – Suppression des boucles

3.2.3 Prise en compte uniquement des relation réciproque

Afin de garder une certaine cohérence dans nos données, nous avons fait le choix de garder uniquement les relations réciproques, c'est-à-dire les gens qui se suivent mutuellement. Pour ce faire, nous avons créé un script Python permettant de générer un fichier CSV contenant uniquement les nœuds réciproques (ce qui peut aussi pallier le biais des influenceurs, c'est-à-dire des gens qui sont suivis par beaucoup de monde mais qui ne suivent pas autant en retour).

```

◆ c1.py > ...
>--Initialise Reactive Jupyter | Sync all Stale code
1 import pandas as pd
2
3 # Charger le fichier CSV
4 df = pd.read_csv('reciprocal_relation.csv')
5
6 # Compter les connexions de chaque nœud
7 node_degrees = df['FromNodeId'].value_counts().reset_index()
8 node_degrees.columns = ['Node', 'Degree']
9
10 # node_degrees = df['fromnodeid'].value_counts().reset_index()
11 # node_degrees.columns = ['Node', 'Degree']
12 # print(node_degrees.head()) # Vérifie si les données sont correctes
13
14 # print(df.columns)
15
16 # node_degrees = df['FromNodeId'].value_counts().reset_index()
17 # node_degrees.columns = ['Node', 'Degree']
18 # print(node_degrees.head())
19
20
21
22 # Sélectionner les 10 000 nœuds les plus connectés
23 top_nodes = node_degrees.nlargest(10000, 'Degree')[['Node']]
24
25 # Filtrer les relations pour ne garder que celles entre ces nœuds
26 df_filtered = df[df['FromNodeId'].isin(top_nodes) & df['ToNodeId'].isin(top_nodes)]
27
28 # Sauvegarder le fichier échantilloné
29 df_filtered.to_csv('reciprocal_relation_sample.csv', index=False)
30

```

FIGURE 3 – relation réciproque

3.3 Algorithme de parcours dans le graphe

Étant donnée les limites matérielles auxquelles nous sommes confronté, nous ne pouvons pas générer le graphe entier avec tous les nœuds. Afin de palier à ce problème nous avons implémenté des algorithmes nous permettant de prendre des échantillons représentatif de notre population (les utilisateurs de ce réseau sociale) pour nos test.

3.3.1 Random Walk Sampling

Principe : on démarre à un nœud au hasard, puis on choisit un voisin au hasard à chaque étape.

```

◆ main.py > ...
1 import pandas as pd
2 import networkx as nx
3 import matplotlib.pyplot as plt
4 import random
5
6 # Chemin du fichier
7 # file_path = "soc-LiveJournal.txt"
8 # file_path = "reciprocal_relations.txt"
9 # file_path = "subgraphhdm.csv"
10 file_path = "reciprocal_relation_sample.csv"
11
12 # Lire le fichier tout en ignorant les lignes de commentaires
13 # data = pd.read_csv(file_path, sep="\t", comment="#" , header=None, names=["FromNodeId", "ToNodeId"])
14 data = pd.read_csv(file_path, sep=",", comment="#", header=None, names=["FromNodeId", "ToNodeId"])
15
16 # Afficher un aperçu des données
17 print(data.head())
18
19 # nrows=100000
20 # Créer un graphe orienté
21 G = nx.DiGraph()
22
23 # Ajouter les arêtes au graphe
24 G.add_edges_from(data.values)
25
26 G.remove_edges_from(nx.selfloop_edges(G))
27
28 # Afficher des informations sur le graphe
29 print("Nombre de nœuds ::", G.number_of_nodes())
30 print("Nombre d'arêtes ::", G.number_of_edges())
31 # Détection des communautés avec l'algorithme de Louvain
32 # Extraire un sous-graphe pour visualisation
33 # subgraph = G.subgraph(list(G.nodes)[1:100]) # Par exemple, les 100 premiers nœuds
34 random_nodes = random.sample(list(G.nodes), 100) # 100 nœuds aléatoires
35 subgraph = G.subgraph(random_nodes) # Sous-graphe de 100 nœuds
36
37 # Visualiser le sous-graphe
38 plt.figure(figsize=(12, 10))
39 nx.draw(subgraph, with_labels=True, node_size=50, font_size=8)
40 plt.title("Sous-graphe de 100 nœuds")
41 plt.show()

```

FIGURE 4 – Parcours aléatoire

3.3.2 Induced Subgraph from High-Degree Nodes

On choisit quelques noeuds avec degré élevé + tous leurs voisins Intéressant pour cibler les "super connecteurs"

```

yep2.py > ...
1 import pandas as pd
2 import networkx as nx
3 import matplotlib.pyplot as plt
4 import random
5 import community
6 import community.community_louvain as community_louvain # Algorithme de Louvain
7 import numpy as np
8
9
10 file_path = "reciprocal_relation_sample.csv"
11 df = pd.read_csv(file_path)
12
13
14 node_counts = df["FromNodeId"].value_counts() + df["ToNodeId"].value_counts()
15 node_counts = node_counts.fillna(0).astype(int)
16
17
18 node_counts.sort_values(ascending=False)
19 high_degree_nodes = set(node_counts[node_counts > 30].index)
20
21 # Filtrer les relations où les deux nœuds ont un degré > 5
22 df_filtered = df[
23     df["FromNodeId"].isin(high_degree_nodes) & df["ToNodeId"].isin(high_degree_nodes)
24 ]
25
26 # Supprimer les relations avec eux-mêmes (self-loops)
27 df_filtered = df_filtered[df_filtered["FromNodeId"] != df_filtered["ToNodeId"]]
28
29 # Afficher le DataFrame filtré
30 df_filtered
31 # Création du graphe
32 G = nx.Graph()
33 for _, row in df_filtered.iterrows():
34     G.add_edge(row["FromNodeId"], row["ToNodeId"])
35
36 # Détection des communautés avec Louvain
37 partition = community_louvain.best_partition(G)
38
39 # Attribution des couleurs aux communautés
40 communities = list(set(partition.values()))
41 colors = plt.cm.rainbow(np.linspace(0, 1, len(communities))) # Génération de couleurs distinctes
42 node_colors = [colors[communities.index(partition[node])] for node in G.nodes()]
43
44 # Positionnement des nœuds
45 pos = nx.spring_layout(G, seed=42)
46
47 # Affichage du graphe avec coloration par communauté
48 plt.figure(figsize=(10, 8))
49 nx.draw(G, pos, with_labels=False, node_color=node_colors, edge_color='black',
50         node_size=0.8, font_size=6, width=0.08)
51 plt.title("Graphe des liens d'amitié (coloration par communauté)")
52 plt.show()

```

FIGURE 5 – Script degré

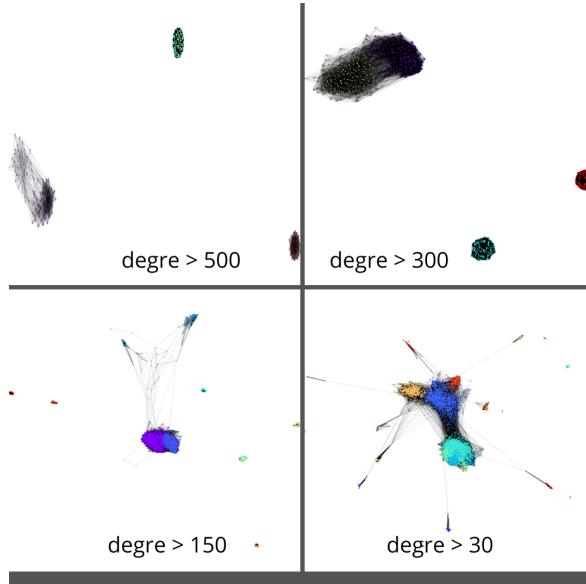


FIGURE 6 – degré graphe

3.3.3 Breadth-First Search (BFS) Sampling

Principe : on démarre à un nœud, on explore tous ses voisins, puis les voisins des voisins. Avantage : bon pour capturer une communauté locale

```

❸ yep.py >...
    o initialise Reactive Jupyter | Sync all State code
1     import pandas as pd
2     import networkx as nx
3     import matplotlib.pyplot as plt
4     import random
5     import community
6     import community.community_louvain as community_louvain # Algorithme de Louvain
7     import numpy as np
8
9     file_path = "reciprocal_relation_sample.csv"
10    df = pd.read_csv(file_path, sep=",", comment="#", header=None, names=["FromNodeId", "ToNodeId"])
11
12    start_node = df["FromNodeId"].sample(1).values[0]
13
14
15    # Récupération des arêtes où ce nœud apparaît
16    sample_df = df[(df["FromNodeId"] == start_node) | (df["ToNodeId"] == start_node)]
17
18    # Expansion pour inclure les voisins des voisins (augmenter la connectivité)
19    for _ in range(3): # Profondeur d'exploration
20        neighbor_nodes = sample_df["ToNodeId"].unique()
21        new_links = df[df["FromNodeId"].isin(neighbor_nodes)]
22        sample_df = pd.concat([sample_df, new_links]).drop_duplicates()
23
24    # Limiter à 10000 lignes
25    sample_df = sample_df.head(10000)
26
27    print(sample_df)
28
29
30    # Création du graphe
31    G = nx.Graph()
32    for _, row in sample_df.iterrows():
33        G.add_edge(row['FromNodeId'], row['ToNodeId'])
34
35    # Détection des communautés avec Louvain
36    partition = community_louvain.best_partition(G)
37
38    # Attribution des couleurs aux communautés
39    communities = list(set(partition.values()))
40    colors = plt.cm.rainbow(np.linspace(0, 1, len(communities))) # Génération de couleurs distinctes
41    node_colors = [colors[communities.index(partition[node])] for node in G.nodes()]
42
43    # Positionnement des nœuds
44    pos = nx.spring_layout(G, seed=42)
45
46    # Affichage du graphe avec coloration par communauté
47    plt.figure(figsize=(10, 8))
48    nx.draw(G, pos, with_labels=False, node_color=node_colors, edge_color='gray',
49            node_size=1, font_size=6, width=0.05)
50    plt.title("Graphe des liens d'amitié (coloration par communauté)")
51    plt.show()

```

FIGURE 7 – Script Voisin

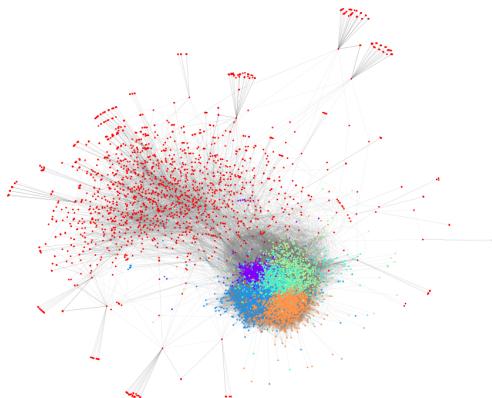


FIGURE 8 – Voisin Graphe

3.4 Identification des communautés

- Outils utilisés :

- Louvain Method pour la détection des communautés, avec la librairie `community`, `louvain`, `NetworkX`.
- Matplotlib et Numpy pour la visualisation et le traitement des couleurs des communautés.

- **Détails :** Les communautés dans le graphe sont identifiées à l'aide de la méthode de Louvain, qui optimise la modularité pour déterminer des groupes de nœuds fortement connectés entre eux. Cette approche permet de diviser le graphe en sous-graphes denses tout en maximisant la séparation entre ces groupes.

3.5 Détermination des ponts reliant plusieurs communautés

- Outils utilisés :

- Détection des ponts via NetworkX, qui analyse les connexions entre les communautés pour identifier les nœuds qui relient des communautés distinctes.
- Spring layout de NetworkX pour positionner les nœuds de manière intuitive.

- **Détails :** Les nœuds ponts sont détectés comme étant des nœuds qui connectent des communautés différentes. Pour chaque nœud, nous vérifions s'il existe des voisins appartenant à une communauté différente. Ces nœuds jouent un rôle clé dans la transmission d'informations entre les communautés et sont donc essentiels pour comprendre les dynamiques globales du réseau.

3.6 Identification des nœuds relais

- Outils utilisés :

- Sélection des nœuds relais parmi les nœuds ayant une forte centralité (degré élevé) et appartenant à des communautés différentes.
- Random.shuffle() pour sélectionner de manière aléatoire certains voisins intra-communauté pour l'esthétique de la visualisation.

- **Détails :** Les nœuds relais sont les nœuds qui servent de point de connexion entre différentes communautés. Nous avons sélectionné les *top-k nœuds ponts* par communauté, où k est un paramètre défini, afin de limiter le nombre de nœuds à afficher tout en maintenant une représentation significative des connexions inter-communautés. Ces nœuds sont cruciaux pour faciliter les échanges d'informations à travers le réseau.

3.7 Interprétation

- Outils utilisés :

- Visualisation du graphe via Matplotlib et NetworkX.
- Utilisation de la palette de couleurs rainbow pour distinguer les différentes communautés.

- **Détails :** La visualisation montre les nœuds des différentes communautés avec des couleurs distinctes, tandis que les nœuds ponts, qui relient différentes communautés, sont mis en évidence. Les nœuds relais, ayant un degré élevé et un rôle de connecteurs entre les groupes, sont essentiels pour comprendre la structure du réseau. En observant le graphe, nous pouvons identifier les principales interactions entre les communautés et les nœuds influents qui jouent un rôle crucial dans la circulation de l'information au sein du réseau.

4 Bibliothèques, Outils et technologies

5 Travail réalisé

5.1 Script Python permettant de nettoyer nos données

Pour filtrer les données en raison de leur grande taille, nous avons choisi de ne conserver que les relations réciproques, c'est-à-dire celles où deux individus s'influencent mutuellement. Par exemple, si A suit B et que B suit également A, cette relation est maintenue. En revanche, si A suit C mais que C ne suit pas A, cette relation est supprimée, car elle n'est pas réciproque. De plus, nous avons éliminé toutes les relations où un individu suit lui-même, c'est-à-dire les relations de type "A suit A". Ce filtrage permet de réduire la taille des données et de conserver uniquement les interactions mutuelles pertinentes.

5.2 algorithmes de parcours

- Random Walk Sampling
- Induced Subgraph from High-Degree Nodes
- Breadth-First Search (BFS) Sampling

6 Détection de communautés

Le script Python présenté ci-dessous permet de détecter des communautés dans un réseau social à partir de relations réciproques. Il filtre d'abord les données en ne gardant que les noeuds ayant un degré supérieur à 30 et les relations réciproques entre ces noeuds, en éliminant les relations où un noeud suit lui-même. Ensuite, un graphe est construit avec ces données filtrées. L'algorithme de Louvain est utilisé pour détecter les communautés.

```
1 import pandas as pd
2 import networkx as nx
3 import matplotlib.pyplot as plt
4 import community
5 import community.community_louvain as community_louvain
6 import numpy as np
7
8 file_path = "reciprocal_relation_sample.csv"
9 df = pd.read_csv(file_path)
10
11 node_counts = df["FromNodeId"].value_counts() + df["ToNodeId"].
12     value_counts()
13 node_counts = node_counts.fillna(0).astype(int)
14
15 node_counts.sort_values(ascending=False)
16 high_degree_nodes = set(node_counts[node_counts > 30].index)
17
18 df_filtered = df[
19     df["FromNodeId"].isin(high_degree_nodes) & df["ToNodeId"].
20         isin(high_degree_nodes)
21 ]
22
23 df_filtered = df_filtered[df_filtered["FromNodeId"] != 
24     df_filtered["ToNodeId"]]
25
26 G = nx.Graph()
27 for _, row in df_filtered.iterrows():
28     G.add_edge(row['FromNodeId'], row['ToNodeId'])
29
30 partition = community_louvain.best_partition(G)
31
32 communities = list(set(partition.values()))
33 colors = plt.cm.rainbow(np.linspace(0, 1, len(communities)))
34 node_colors = [colors[communities.index(partition[node])] for
35     node in G.nodes()]
36
37 pos = nx.spring_layout(G, seed=42)
38
39 plt.figure(figsize=(10, 8))
40 nx.draw(G, pos, with_labels=False, node_color=node_colors,
41         edge_color='black',
42             node_size=0.8, font_size=6, width=0.08)
43 plt.title("Graphe des liens d'amitié (coloration par
```

```
39 |     c o m m u n a u t ) " )  
39 |     plt . show ( )
```

6.1 Résultats aperçus

Après l'application de l'algorithme de Louvain, nous obtenons un graphe où chaque noeud représente un individu et les arêtes entre les noeuds représentent les relations réciproques. Les noeuds sont colorés en fonction des communautés auxquelles ils appartiennent, facilitant la visualisation des groupes interagissant fortement.

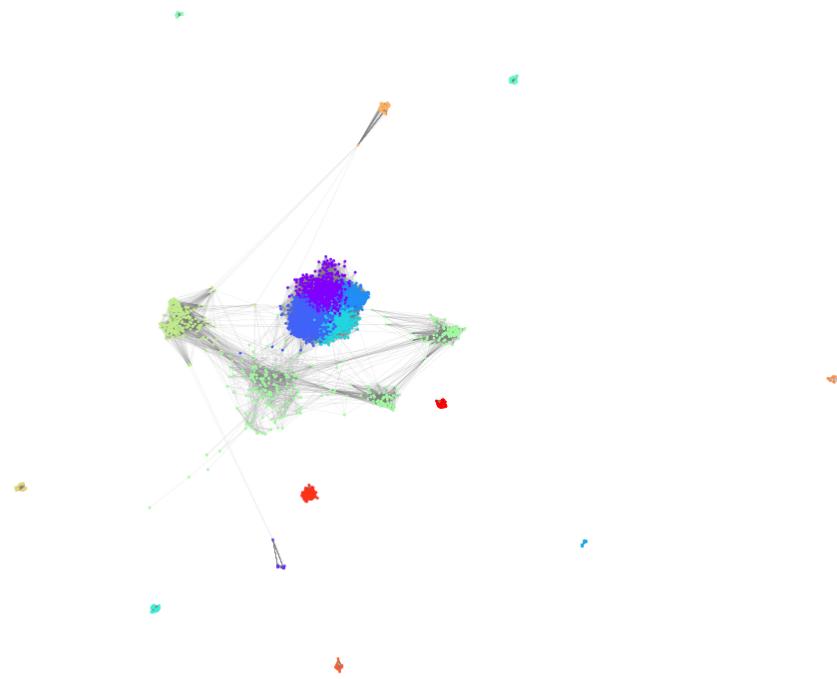


FIGURE 9 – Détection des communautés dans le réseau d'amitié

6.2 Analyse des liens inter-communautés et des nœuds ponts

Dans cette section, nous analysons les liens inter-communautés et identifions les nœuds ponts. Ces derniers sont essentiels pour relier différentes communautés et favoriser la circulation de l'information. Voici le code Python permettant de détecter et visualiser ces nœuds.

```
1 # Identification des nœuds ponts
2 bridge_nodes = set()
3 for node in G.nodes():
4     comm = partition[node]
5     for neighbor in G.neighbors(node):
6         if partition[neighbor] != comm:
7             bridge_nodes.add(node)
8             break
9
10 # Visualisation des connexions inter-communautés
11 pos = nx.spring_layout(G, seed=42)
12 subG = G.subgraph(bridge_nodes)
13 node_colors = [colors[communities.index(partition[node])] for
14     node in subG.nodes()]
15
16 plt.figure(figsize=(12, 10))
17 nx.draw_networkx_nodes(subG, pos, node_color=node_colors,
18     node_size=50)
19 nx.draw_networkx_edges(subG, pos, edge_color='black', width
20     =0.4)
21 plt.title("Connexions entre communautés via les nœuds ponts")
22 plt.axis("off")
23 plt.tight_layout()
24 plt.savefig("inter_community_links.png", dpi=300)
25 plt.show()
```

Connexions entre communautés via les nœuds ponts

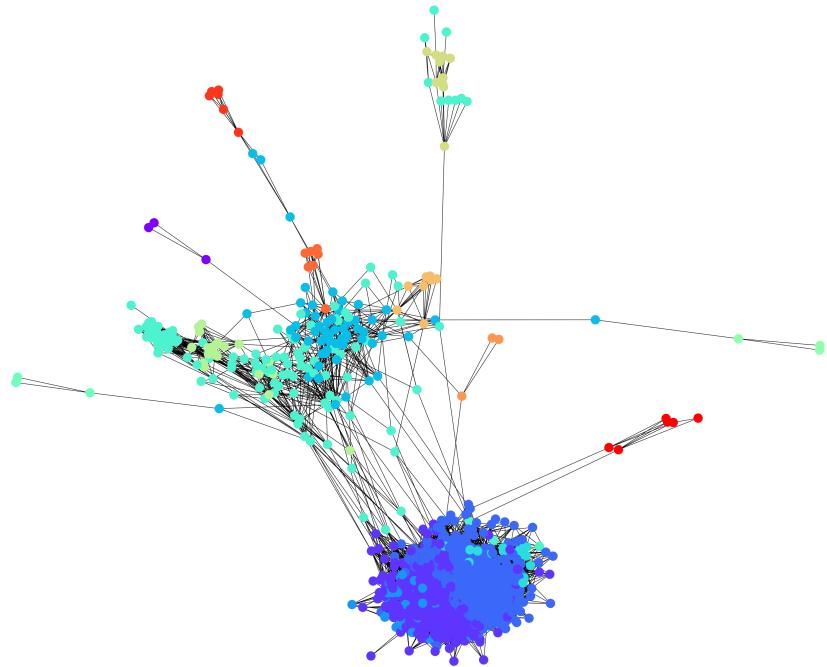


FIGURE 10 – Liens inter-communautés via les nœuds ponts

6.3 Identification et visualisation des connexions clés entre les communautés

Pour mieux comprendre la dynamique globale du réseau, nous analysons dans cette section les **connexions inter-communautés**, c'est-à-dire les liens reliant des nœuds appartenant à des groupes différents. Ces connexions sont essentielles, car elles révèlent les points de contact stratégiques entre les communautés, souvent représentés par des nœuds occupant un rôle de liaison ou d'ambassadeur.

Les étapes principales de cette analyse sont les suivantes :

- Identifier les arêtes connectant des nœuds de communautés différentes.
- Extraire les sous-graphes formés par ces connexions inter-groupes.
- Générer une visualisation pour chaque paire de communautés connectées, en mettant en valeur les arêtes clés.

Le script Python suivant a été utilisé pour mettre en œuvre cette analyse :

```
1 # Couleurs des communautés
2 unique_communities = sorted(set(partition.values()))
3 colors = plt.cm.rainbow(np.linspace(0, 1, len(
4     unique_communities)))
5 community_color_map = {comm: colors[i] for i, comm in enumerate(
6     unique_communities)}
7
8 # Détection des arêtes entre communautés
9 inter_edges = [
10     (u, v, partition[u], partition[v]) for u, v in G.edges() if
11         partition[u] != partition[v]
12 ]
13
14 # Paires de communautés uniques
15 pairs = set((min(c1, c2), max(c1, c2)) for _, _, c1, c2 in
16     inter_edges)
17
18 for comm1, comm2 in pairs:
19     connector_nodes = set()
20     edges_between = []
21
22     for u, v, c1, c2 in inter_edges:
23         if set([c1, c2]) == set([comm1, comm2]):
24             connector_nodes.update([u, v])
25             edges_between.append((u, v))
26
27     # Ajout des voisins immédiats
28     neighbor_nodes = set()
29     for node in connector_nodes:
30         neighbor_nodes.update(G.neighbors(node))
31
32     total_nodes = connector_nodes.union(neighbor_nodes)
33     subG = G.subgraph(total_nodes)
34
35     # Position et couleurs
36     pos = nx.spring_layout(subG, seed=42)
```

```

33     node_colors = [community_color_map[partition[n]] for n in
34         subG.nodes()]
35     edge_colors = ['black' if (u, v) in edges_between or (v, u)
36         in edges_between else 'gray'
37             for u, v in subG.edges()]
38
39     # Dessin
40     plt.figure(figsize=(10, 8))
41     nx.draw_networkx_nodes(subG, pos, node_color=node_colors,
42         node_size=100)
43     nx.draw_networkx_edges(subG, pos, edge_color=edge_colors,
44         width=1.5)
45     plt.title(f"Connexions entre communautés {comm1} et {comm2}")
46     plt.axis("off")
47     plt.tight_layout()
48     plt.savefig(f"liaison_{comm1}_{comm2}.png", dpi=300)
49     plt.show()

```

Listing 1 – Identification des connexions inter-communautés

Les visualisations ci-dessous illustrent quelques-unes de ces connexions inter-communautaires identifiées :

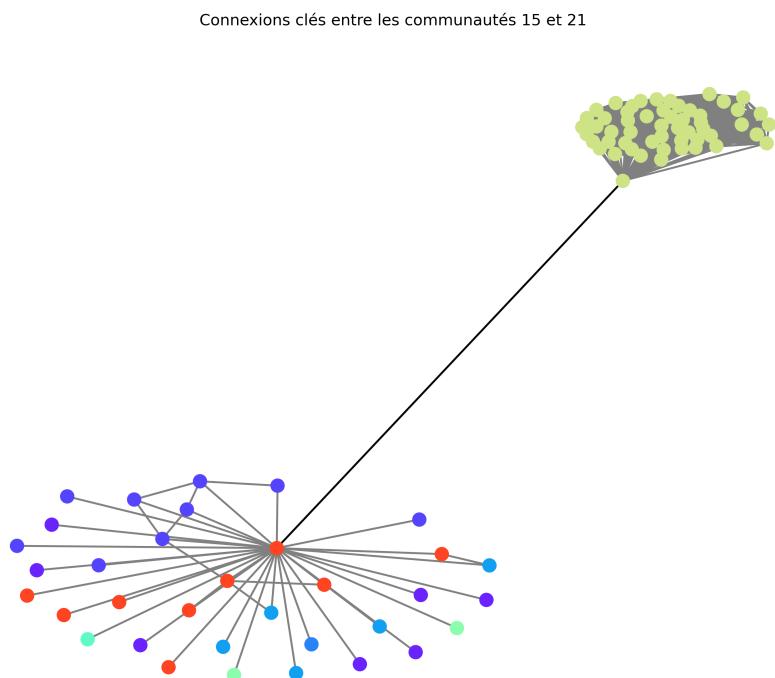


FIGURE 11 – Connexions clés entre les communautés 15 et 21

Connexions clés entre les communautés 12 et 22

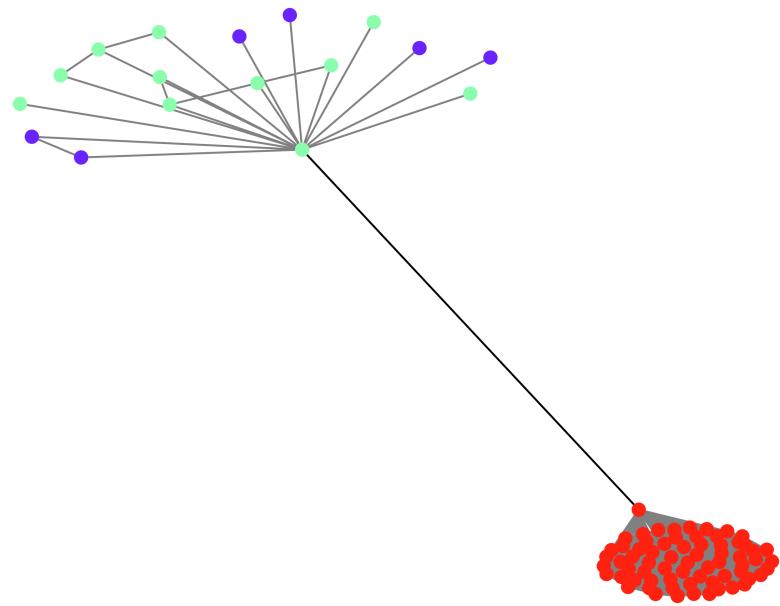


FIGURE 12 – Connexions clés entre les communautés 12 et 22

Connexions clés entre les communautés 12 et 13

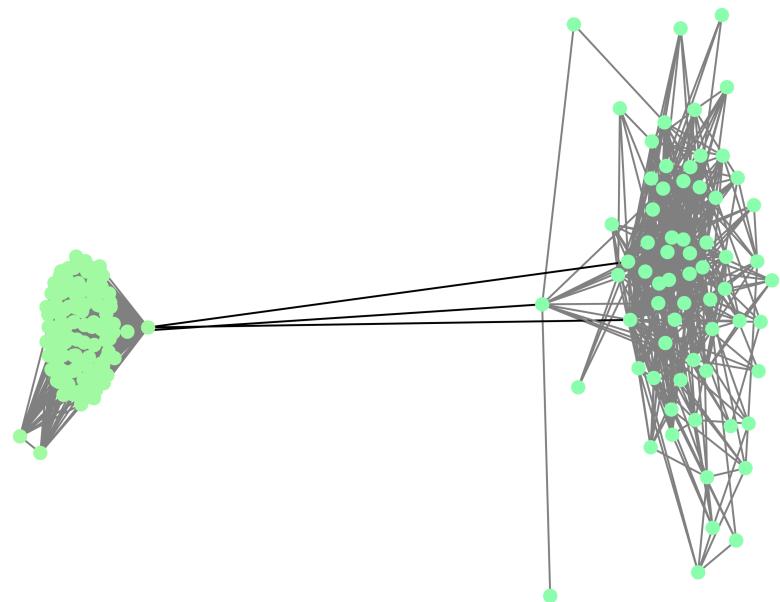


FIGURE 13 – Connexions clés entre les communautés 12 et 13

Chaque image représente un sous-graphe où :

- Les nœuds sont colorés selon leur communauté d'origine.
- Les **arêtes noires** correspondent aux connexions directes entre deux communautés.

- Les **arêtes grises** représentent les connexions locales (intra-communautaires) servant de contexte.

Interprétation. Ces visualisations permettent d'identifier les noeuds stratégiques jouant un rôle de *connecteurs inter-groupes*. Ces derniers constituent souvent des relais d'information importants dans un contexte social, professionnel ou informationnel. L'étude de ces ponts peut s'avérer précieuse pour notre problématique.

6.4 Identification des rôles et visualisation d'un sous-graphe optimisé

Dans cette section, nous allons explorer les rôles des noeuds dans le réseau à l'aide de deux métriques clés : le **degré** et la **centralité d'intermédiation** (*betweenness*). Ces indicateurs permettent de classer les noeuds en trois catégories :

- **Leader local** : noeuds très connectés au sein de leur communauté.
- **Ambassadeur** : noeuds jouant un rôle de pont entre communautés.
- **Satellite** : noeuds périphériques avec moins de connexions.

L'objectif est de générer un **sous-graphe optimisé** mettant en lumière les connexions stratégiques entre ces différents rôles.

Étapes de l'analyse

1. Création d'un DataFrame contenant pour chaque noeud : son degré, sa centralité d'intermédiation et sa communauté.
2. Normalisation des métriques avec **MinMaxScaler**.
3. Attribution d'un rôle à chaque noeud selon des seuils prédéfinis.
4. Sélection des *Leaders* et *Ambassadeurs*, et ajout de leurs *Satellites* les plus connectés.
5. Visualisation du sous-graphe ainsi construit.

Code Python

```

1 from sklearn.preprocessing import MinMaxScaler
2
3 # Creation du DataFrame des nuds
4 df_nodes = pd.DataFrame({
5     "Node": list(G.nodes),
6     "Degree": [degree[n] for n in G.nodes],
7     "Betweenness": [betweenness[n] for n in G.nodes],
8     "Community": [partition[n] for n in G.nodes]
9 })
10
11 # Normalisation
12 scaler = MinMaxScaler()
13 df_nodes[["NormDegree", "NormBetweenness"]] = scaler.
14     fit_transform(
15         df_nodes[["Degree", "Betweenness"]]
16     )
17
18 # Attribution des rôles
def get_role(row):

```

```

19     if row["NormBetweenness"] > 0.6:
20         return "Ambassadeur"
21     elif row["NormDegree"] > 0.75:
22         return "Leader local"
23     else:
24         return "Satellite"
25
26 df_nodes["Role"] = df_nodes.apply(get_role, axis=1)
27
28 # Selection des nuds classés et satellites les plus connectés
29 key_nodes = df_nodes[df_nodes["Role"].isin(["Leader local", "Ambassadeur"])]["Node"]
30 selected_nodes = set(key_nodes)
31
32 for node in key_nodes:
33     neighbors = list(G.neighbors(node))
34     satellites = [n for n in neighbors if df_nodes.loc[df_nodes["Node"] == n, "Role"].values[0] == "Satellite"]
35     top_satellites = sorted(satellites, key=lambda x: degree[x], reverse=True)[:20]
36     selected_nodes.update(top_satellites)
37
38 # Sous-graphe final
39 subG = G.subgraph(selected_nodes)
40 df_sub = df_nodes[df_nodes["Node"].isin(subG.nodes)].copy()
41
42 # Visualisation
43 pos = nx.spring_layout(subG, seed=42)
44 color_map = {
45     "Leader local": "#e74c3c",
46     "Ambassadeur": "#2980b9",
47     "Satellite": "#95a5a6"
48 }
49 colors = df_sub["Role"].map(color_map)
50 sizes = 200 + df_sub["NormDegree"] * 10
51
52 labels = {
53     row["Node": row["Role"]
54     for _, row in df_sub.iterrows()
55     if row["Role"] != "Satellite"
56 }
57
58 plt.figure(figsize=(16, 12))
59 nx.draw_networkx_nodes(subG, pos, node_color=colors, node_size=sizes, alpha=0.9)
60 nx.draw_networkx_edges(subG, pos, alpha=0.25, width=0.7)
61
62 # Légende
63 from matplotlib.patches import Patch
64 legend_elements = [
65     Patch(color=color_map["Leader local"], label="Leader local"),

```

```

66     Patch(color=color_map["Ambassadeur"], label="Ambassadeur"),
67     Patch(color=color_map["Satellite"], label="Satellite")
68 ]
69 plt.legend(handles=legend_elements, loc="upper right", fontsize=12)
70
71 plt.title("Leadership et satellites filtrés", fontsize=16)
72 plt.axis("off")
73 plt.tight_layout()
74 plt.savefig("smart_roles_graph_reduced_satellites.png", dpi=300)
75 plt.show()

```

Résultat visuel

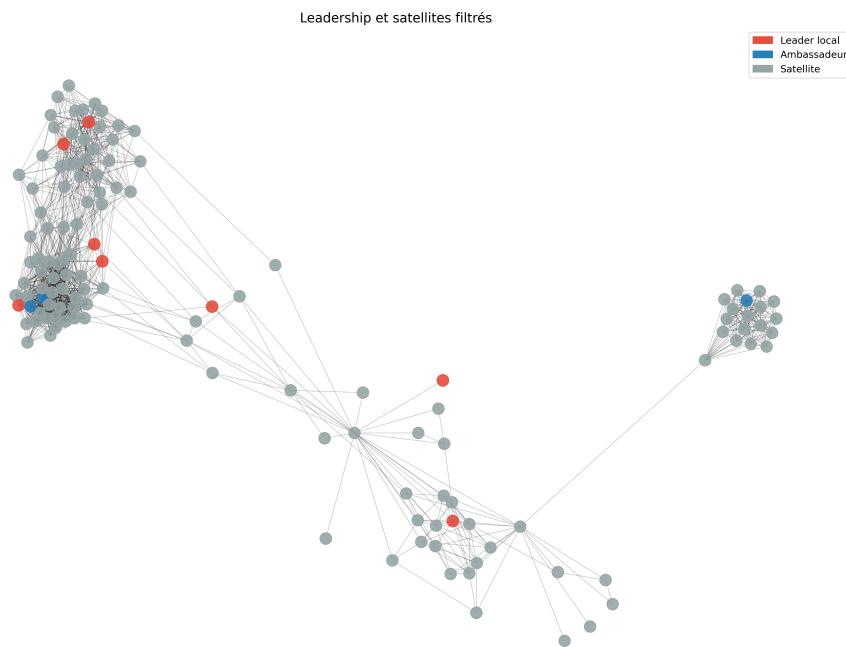


FIGURE 14 – Visualisation des rôles dans le réseau : Leaders locaux, Ambassadeurs et Satellites

Interprétation

- **Leaders locaux** : au centre des interactions dans leur communauté, ils assurent une forte cohésion interne.
- **Ambassadeurs** : véritables ponts entre communautés, ils favorisent la circulation de l'information dans le réseau.
- **Satellites** : moins connectés, ils restent néanmoins pertinents en tant que soutiens locaux.

Cette représentation offre une lecture stratégique de la structure du réseau. En distinguant les rôles, elle met en évidence les points de pouvoir et les passerelles essentielles à l'interconnexion des communautés.

7 Difficultés rencontrées

Au fil du projet, nous avons été confrontés à plusieurs types de difficultés, aussi bien liées aux données qu'aux outils ou aux ressources matérielles.

7.1 difficultés de traitement des données

Les données brutes que nous avons utilisées étaient massives (plus de 50 millions de lignes) et au format .txt, ce qui les rendait peu exploitables dans leur état initial. Nous avons donc dû commencer par les nettoyer et les transformer. Cela a impliqué de convertir le fichier en .csv, de supprimer les boucles, de filtrer les doublons, et surtout de conserver uniquement les relations réciproques, c'est-à-dire celles où deux utilisateurs sont amis dans les deux sens.

7.2 Difficultés matérielles

7.2.1 génération du graphe

L'un des principaux défis a été la construction du graphe complet, qui demandait beaucoup de mémoire. Sur un ordinateur personnel, il était tout simplement impossible de charger les 4 millions de noeuds et 50 millions de relations dans leur totalité. Nous avons donc dû adapter notre méthode en extrayant des sous-graphes représentatifs à partir de différentes stratégies d'échantillonnage. Cela nous a permis de travailler sur des volumes plus raisonnables tout en conservant la structure du réseau global.

7.2.2 contrainte logicielle

Nous avons remarqué que NetworkX, bien qu'intuitif et pratique pour des graphes de taille moyenne, ne supporte pas bien les très gros volumes. Les calculs devenaient très lents, voire impossibles à exécuter sans planter. Pour contourner cette limite, nous avons tenté d'utiliser Neo4j, qui est beaucoup mieux adapté à la gestion de grands graphes. Cela dit, même si Neo4j s'est révélé très prometteur, nous n'avons pas pu en exploiter son potentiel. Nous avons alors continué sur NetworkX.

8 Bilan

8.1 Conclusion

Les résultats de nos analyses montrent que, bien que tous les utilisateurs très connectés ne soient pas nécessairement des ponts entre communautés, une proportion significative d'entre eux occupe cette fonction stratégique. Grâce à la combinaison des mesures de centralité, des liens inter-communautés et de la visualisation du réseau, nous avons pu mettre en évidence plusieurs "ambassadeurs" qui assurent la liaison entre des groupes qui, autrement, seraient peu ou pas connectés.

Ces individus jouent un rôle fondamental dans la diffusion de l'information, la structuration du réseau et le maintien de la cohésion globale. Ils constituent des cibles potentielles pour des actions comme la communication virale, l'analyse d'influence ou la prévention de fragmentation du réseau. Nous pouvons donc conclure que, oui, les

personnes ayant beaucoup d'amis ont souvent un rôle de connecteurs entre différentes communautés, même si ce rôle dépend également de la position topologique du nœud dans le graphe et non uniquement de son degré.

8.2 Perspectives

Pour aller plus loin nous pourrions étudier l'évolution temporelle du graphe, afin de voir comment les rôles de connecteurs évoluent dans le temps : certains utilisateurs deviennent-ils des ponts à mesure qu'ils se connectent à de nouveaux groupes ? Inversement, d'autres perdent-ils cette position stratégique ?