INSTITUT DE FORMATION ET DE RECHERCHE EN INFORMATIQUE
IFRI-UAC

Big Data:

Analyse des structures de graphe des réseaux sociaux pour la recherche de chemins les plus courts entre les noeuds via MapReduce

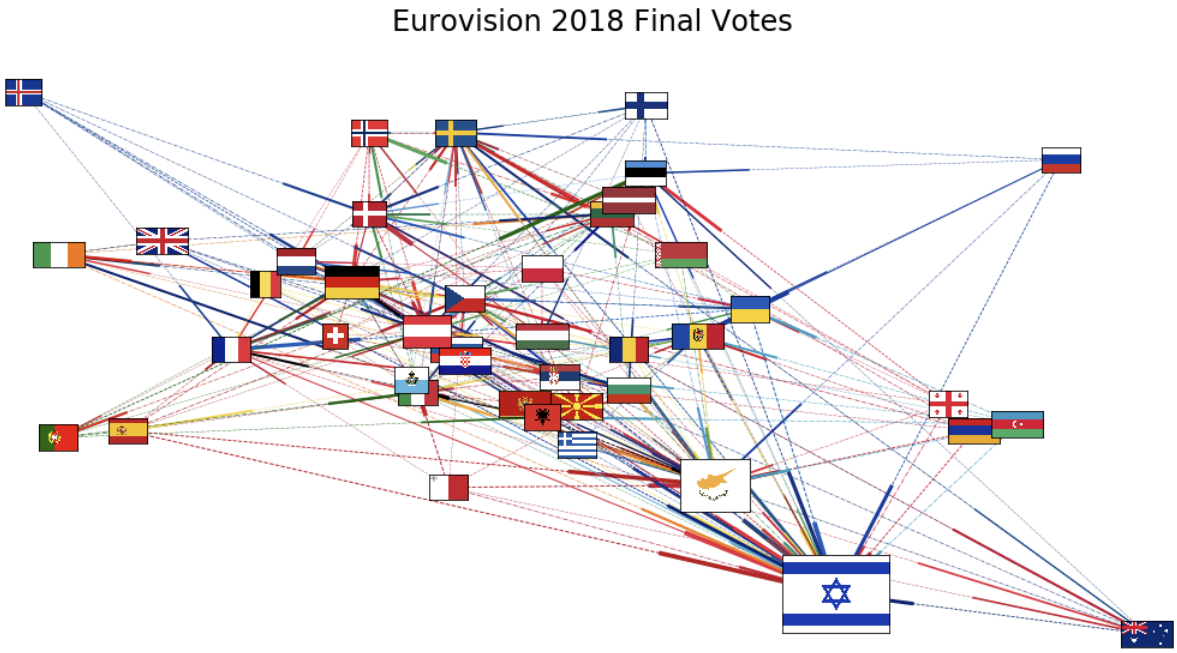
Membres du groupe:
AZAGBA Roméo
HONON DADJO Kelvine
MAMA Abbas

Supervisé par:
Dr AOGA John

Juin 2024

1- Présentation du problème

Le problème abordé concerne la diffusion d'information dans un réseau, modélisé par un graphe non orienté. Chaque sommet représente un individu et chaque arête représente une connexion entre deux individus. L'objectif est de déterminer les chemins les plus courts entre un nœud source et tous les autres nœuds du graphe, afin de comprendre comment l'information se propage dans le réseau.



Dans un graphe comme celui-ci, il faudra déterminer pour chaque pays votant, le nombre de connexions minimales possibles le séparant de chacun des autres pays.

2- Méthodes classiques de résolution du problème

Traditionnellement, la recherche des plus courts chemins dans un graphe est réalisée à l'aide d'algorithmes comme Dijkstra ou la recherche en largeur (BFS). Ces algorithmes sont efficaces pour des graphes de taille modeste. Cependant, lorsque les graphes deviennent très grands, ces méthodes deviennent rapidement inefficaces en termes de temps de calcul et de mémoire.

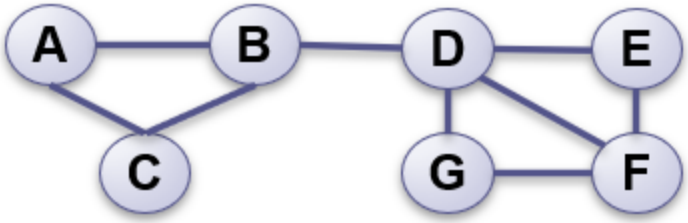
3- Intérêt pour MapReduce

MapReduce est un modèle de programmation distribué qui permet de traiter et de générer de grands ensembles de données avec un algorithme parallèle et distribué. Son intérêt réside dans sa capacité à :

- Traiter de très grandes quantités de données en parallèle.
- Utiliser des ressources de calcul distribuées pour accélérer le traitement.
- Simplifier la gestion des tâches complexes de calcul distribué grâce à son modèle de programmation simple (mappers et reducers).

4- Présentation de la démarche de résolution avec MapReduce

Soit le graphe social suivant:



Dans ce graphe, les nœuds représentent des personnes et les arêtes représentent des connexions sociales entre elles. Nous choisirons le nœud source A pour trouver les plus courts chemins vers tous les autres nœuds.

Mappers et Reducers

Initialisation de l'Algorithme

Mapper : La première étape consiste à initialiser le graphe en émettant des paires pour chaque arête. Pour chaque ligne représentant une arête, le mapper émet deux paires, assurant que le graphe est non orienté.

Par exemple, pour l'arête A -- B :

- Mapper(A, B) émet (A, B) et (B, A)

Le processus est répété pour chaque arête du graphe :

- A -- B : (A, B) et (B, A)
- B -- C : (B, C) et (C, B)
- A -- C : (A, C) et (C, A), etc…

Reducer : Ensuite, le reducer regroupe les voisins de chaque nœud et initialise la distance du nœud source A à 0 et des autres nœuds à l'infini. Le nombre de plus courts chemins est initialisé à 0 ou 1 selon le nœud.

- A : distance = 0, chemins = 1
- B, C, D, E, F, G : distance = ∞, chemins = 0

Déroulement de l'Algorithme

Mapper : Pour chaque entrée de la liste d'adjacence, le mapper émet des triplets (nœud, distance + 1, nombre de chemins) pour chaque voisin.

Par exemple, à la première itération, à partir du nœud A (distance = 0, chemins = 1) :

- (A, 0, 1) émet pour chaque voisin (B, 1, 1) et (C, 1, 1)

Reducer : Le reducer regroupe ces triplets pour chaque nœud et détermine :

1. La distance minimale.
2. La somme des plus courts chemins menant à ce nœud.

À la première itération, les résultats intermédiaires pourraient être :

- A B on affecte (B, 1, 1) et distance = 1, chemins = 1
- A C on affecte (C, 1, 1) et distance = 1, chemins = 1

Le processus se répète pour les voisins des nœuds B et C.

Parallélisme

L'algorithme fonctionne de manière itérative et chaque étape est parallélisée. Les mappers et reducers sont exécutés simultanément sur différentes portions

du graphe. Cela permet de diviser le travail et de réduire considérablement le temps total de calcul.

En combinant cette approche de résolution avec une méthode d'exploration de graphe comme le BFS, on parvient à décomposer le problème de recherche des plus courts chemins dans un grand graphe en une série de tâches parallélisées et gérables. Chaque étape de l'algorithme (initialisation, mappage, réduction) est exécutée de manière distribuée, permettant de traiter efficacement de grandes quantités de données dans un temps relativement court.

5- Description de l'exécution de l'algorithme

L'algorithme commence par l'initialisation, créant une liste d'adjacences pour le graphe. Ensuite, une série d'itérations est exécutée où chaque mapper émet des triplets pour ses voisins, et chaque reducer met à jour les distances minimales et les nombres de plus courts chemins pour chaque nœud. Cette boucle continue jusqu'à ce que les distances convergent, c'est-à-dire qu'il n'y ait plus de changements dans les distances et les chemins.

6- Implémentation

L'implémentation du modèle de MapReduce pour notre problème a été réalisée avec Python et sa bibliothèque Apache Beam. Nous avons utilisé le dataset Dolphins qui modélise les associations entre dauphins sous forme de graphe social.

Voici les fonctions implémentées et du code réalisé

```python
def parse_edge(line):
    if line.startswith('%') or line.strip() == '':
        return None
    tokens = line.split()
    return int(tokens[0]), int(tokens[1])
```

Cette fonction prend une ligne d'entrée du fichier de graphes, ignore les lignes de commentaires et les lignes vides, et retourne un tuple représentant une arête du graphe.

```python
def create_graph(edge_list):
    graph = {}
    for u, v in edge_list:
        if u not in graph:
            graph[u] = []
        if v not in graph:
            graph[v] = []
        graph[u].append(v)
        graph[v].append(u)
    return graph
```

Cette fonction prend une liste d'arêtes et crée un dictionnaire représentant le graphe sous forme de liste d'adjacence. Chaque nœud est associé à une liste de ses voisins.

```python
def bfs_shortest_path(graph, start):
    queue = deque([(start, [start])])
    visited = set()
    shortest_paths = {start: [start]}

    while queue:
        node, path = queue.popleft()
        visited.add(node)

        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                new_path = path + [neighbor]
                if neighbor not in shortest_paths or len(new_path) < len(shortest_paths[neighbor]):
                    shortest_paths[neighbor] = new_path
                queue.append((neighbor, new_path))

    return shortest_paths
```

Cette fonction utilise l'algorithme de recherche en largeur (BFS) pour trouver les plus courts chemins depuis un nœud de départ vers tous les autres nœuds du graphe. Elle retourne un dictionnaire où les clés sont les nœuds et les valeurs sont les plus courts chemins sous forme de listes de nœuds.

```python
def map_shortest_paths(node, graph):
    return node, bfs_shortest_path(graph, node)
```

Cette fonction applique l'algorithme BFS à un nœud donné dans le graphe, et retourne le nœud avec ses plus courts chemins.

```python
def plot_initial_graph(edge_list):
    G = nx.Graph()
    G.add_edges_from(edge_list)
    plt.figure(figsize=(12, 12))
    nx.draw(G, with_labels=True, node_size=700, node_color='skyblue', font_size=15)
    plt.title("Graphe des Connexions Initiales")
    plt.savefig("image/graph_initial.png")
    plt.show()
```

Cette fonction utilise NetworkX et Matplotlib pour dessiner et enregistrer l'image du graphe initial.

```python
def plot_shortest_paths(graph, shortest_paths):
    G = nx.Graph()
    G.add_edges_from([(u, v) for u in graph for v in graph[u]])
    plt.figure(figsize=(12, 12))
    pos = nx.spring_layout(G)

    nx.draw(G, pos, with_labels=True, node_size=700, node_color='skyblue', font_size=15, alpha=0.3)

    for start_node, paths in shortest_paths.items():
        for end_node, path in paths.items():
            path_edges = [(u, v) for u, v in zip(path[:-1], path[1:]) if u in pos and v in pos]
            nx.draw_networkx_edges(G, pos, edgelist=path_edges, width=2, edge_color='r')

    plt.title("Chemins les Plus Courts")
    plt.savefig("image/graph_short_path.png")
    plt.show()
```

Cette fonction visualise les plus courts chemins sur le graphe. Les arêtes des chemins les plus courts sont dessinées en rouge pour les distinguer.

```python
def format_shortest_paths(kv):
    node, paths = kv
    formatted_paths = {
        "Start Node": node,
        "Paths": {
            end: {"Path": path, "Path Description": " -> ".join(map(str, path))}
            for end, path in paths.items()
        },
        "Summary": f"{len(paths)} shortest paths found for start node {node}"
    }
    return json.dumps(formatted_paths)
```

Cette fonction formate les résultats des plus courts chemins sous forme de JSON pour faciliter leur stockage et leur lecture ultérieure. Elle inclut des descriptions de chemin lisibles.

```python
options = PipelineOptions()

with beam.Pipeline(options=options) as p:
    lines = p | 'ReadFile' >> beam.io.ReadFromText('./soc-dolphins.mtx')

    edges = (
        lines
        | 'ParseLines' >> beam.Map(parse_edge)
        | 'FilterNone' >> beam.Filter(lambda x: x is not None)
    )

    edge_list = edges | 'CollectEdges' >> beam.combiners.ToList()

    graph = edge_list | 'CreateGraph' >> beam.Map(create_graph)

    start_nodes = (
        edge_list
        | 'GetNodes' >> beam.FlatMap(lambda edges: set([node for edge in edges for node in edge]))
    )

    shortest_paths = (
        start_nodes
        | 'MapShortestPaths' >> beam.Map(lambda node, graph: map_shortest_paths(node, graph),
                                         graph=beam.pvalue.AsSingleton(graph))
    )

    formatted_shortest_paths = (
        shortest_paths
        | 'FormatShortestPaths' >> beam.Map(format_shortest_paths)
    )

    formatted_shortest_paths | 'WriteResults' >> beam.io.WriteToText('./result/result')
```

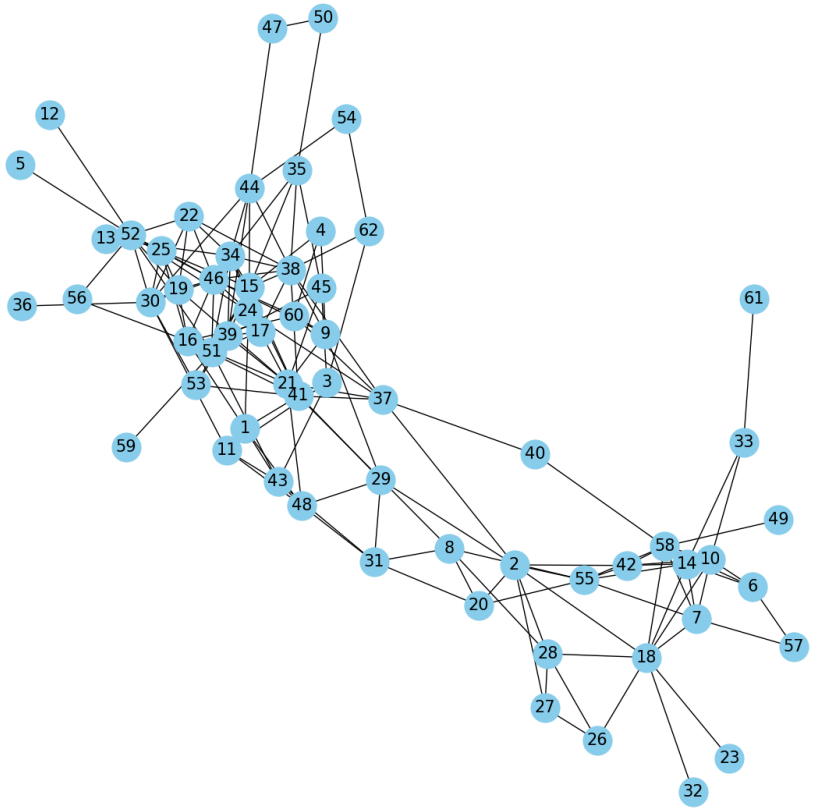
Pipeline Apache Beam :

- **`ReadFile`** : Lit les lignes du fichier de graphe.
- **`ParseLines`** : Parse les lignes en arêtes.
- **`FilterNone`** : Filtre les arêtes nulles.
- **`CollectEdges`** : Collecte toutes les arêtes dans une liste.
- **`CreateGraph`** : Crée le graphe à partir de la liste d'arêtes.
- **`GetNodes`** : Extrait tous les nœuds du graphe.
- **`MapShortestPaths`** : Calcule les plus courts chemins pour chaque nœud.
- **`FormatShortestPaths`** : Formate les résultats des plus courts chemins.
- **`WriteResults`** : Écrit les résultats dans un fichier.

```
with open('./soc-dolphins.mtx', 'r') as f:
    edges = [parse_edge(line) for line in f if parse_edge(line) is not None]
    plot_initial_graph(edges)

shortest_paths = {}
with open('./result/result-00000-of-00001', 'r') as f:
    for line in f:
        data = json.loads(line.strip())
        start_node = data["Start Node"]
        paths = data["Paths"]
        shortest_paths[start_node] = {int(end_node): path_data["Path"] for end_node, path_data in paths.items()}

graph = create_graph(edges)

plot_shortest_paths(graph, shortest_paths)
```

Cette partie du code montre le chargement du dataset et la création du graphe initial par les données de celui-ci, la lecture des données de sortie et l'affichage du graphe final avec les chemins les plus courts mis en surlignement.

A l’exécution, on obtient donc le graphe correspondant au dataset choisi:



Le fichier de résultat généré contient les informations sur les chemins les plus courts de chaque noeud et se présente comme suit (plusieurs résultats ont été

ignorés en raison de la taille du fichier) :


```
{
   "Start Node":1,
   "Paths":{
      "1":{
         "Path":[
            1
         ],
         "Path Description":"1"
      },
      "11":{
         "Path":[
            1,
            11
         ],
         "Path Description":"1 -> 11"
      },
      "61":{
         "Path":[
            1,
            41,
            8,
            55,
            14,
            33,
            61
         ],
         "Path Description":"1 -> 41 -> 8 -> 55 -> 14 -> 33 -> 61"
      }
   },
   "Summary":"62 shortest paths found for start node 1"
}
```


A partir de la lecture de ce fichier, ce graphe est généré et montre les chemins les plus courts obtenus: