**Pattern 1. Graph de GraphX et aggregateMessages.**

**Stratégie :** Jointures et Maps

**Scaling :** 10/10

**Vitesse :** mauvaise

**1 Tableau pour les sommets, 1 tableau pour les arêtes.**

**Fonctions :** subgraph, joinVertices, aggregateMessages, isEmpty

Avec ce pattern, on crée d’abord notre graphe (1 RDD pour les sommets, 1 RDD pour les arêtes). Par la suite, on appelle Graph(sommets, aretes) pour obtenir un graphe GraphX.

Une fois qu’on a un graphe GraphX, on peut utiliser la fonction aggregateMessages.

La fonction aggregateMessages va nous permettre d'exécuter une fonction sur toutes nos triplets d’arêtes. Un triplet d’arête est une structure de donnée de courtoisie pour vous. Un triplet est composé des données de l’arête ainsi que des données du sommet source, et des données du sommet destination.

Une arête qui relie le sommet 1 et le sommet 5 va donc se faire donner les données du sommet 1 et du sommet 5.

Donc, lorsqu’on utilise aggregateMessages, on doit donner deux fonctions. La première fonction, sendMsg, spécifie comment on envoie les messages aux sommets. Chaque arête va donc se faire appelée une fois et va décider des messages qu’elle envoie, en ayant accès dans son contexte au données de la source et de la destination.

Ensuite, vous devez programmer une fonction qui s’appelle mergeMsg. Cette fonction vous donne un message A et un message B et vous demande de produire un message C qui aura le même type.

Finalement, vous pouvez spécifier une joinStrategy en 3ème paramètre pour optimiser un peu, mais vous n’aurez pas d’utilité pour ce paramètre optionnel dans ce TP.

Une fois que les messages sont générés, vous devez les joindre aux sommets existants. Et ensuite, chaque sommet “consomme” ses messages afin de se transformer.

Pour faire cela, vous aurez besoin de la fonction joinVertices. Cette fonction effectue essentiellement un join entre vos messages et vos sommets, et elle lance une fonction map() codée par vous par la suite pour consommer les messages.

Le pattern 1 est simple à utiliser et est très scalable. Pour arrêter votre programme avec ce pattern, on s’arrange pour ne plus envoyer de messages lorsque le calcul est terminé. Ce que vous pourrez détecter facilement en faisant la chose suivante :

***Val messages= aggregateMessages()***

***If (messages.isEmpty) return***

Vous pouvez également arrêter le programme en utilisant des accumulateurs Spark. Ces derniers se font mettre à jour lorsqu’une action Spark est déclenchée. Donc, si vous ne faitez que des transformations mais que vous voulez quand même vérifier l’état des accumulateurs, vous pouvez mettre une action bidon pour déclencher le travail, et ensuite vérifier le contenu de l’accumulateur.

Par exemple :    ***val rien = monRdd.take(1)***

***If (accumulateur.value == 0) return***

Vous avez un exemple complet d’un programme fonctionnant avec le pattern 1 avec le petit code de l’algorithme de coloriage de graphe FC2. (Voir les slides [Apprendre Apache Spark](https://docs.google.com/presentation/d/1haELw6JYHkvX7niwZBSXcjbFOV2fMIvrCiK08FqxNVI/edit#slide=id.g449f1c1309_0_5))