RAPPORT DE TP HADOOP

Étudiant: Slimane AGOURAM (Monome)

Encadré par: M. Stéphane GENAUD

GROUPE: ENSIIE 3A/ILC-2A

INTRODUCTION

Dans ce TP, nous avions pour tache de concevoir un programme Hadoop, soit en map/reduce qui extrait le plus court chemin entre les sommets d'un graphe, soit l'algorithme de Dijkstra tant qu'on l'a appris en cours de théorie des graphes.

Notre programme utilisera un fichier input, contenant la liste des sommets et le coûts des arrêtes, pour traiter son contenu (dans un map puis dans un reduce de regroupement) avant de générer un fichier output contenant la solution du problème en entrée.

La spécifité de ce cas Map/Reduce est le fait qu'il se base non pas que sur un seul passage Map/Reduce, mais sur plusieurs .

En effet, le but étant de retrouver la solution finale, on s'arrêtera dés que notre solution intermédiaire ne change plus.

On expliquera ça plus tard.

Le Input:

Notre input est de la forme suivante :

<Sommet> <Cout de l'arrete {-1 ou 0}>_[<Liste des sommets voisins séparés par des ;>]

Exemple:

A 0_B,C,D

Le format d'input proposé est de cette forme là pour pouvoir exploiter le output d'un passage précèdent dans le passage suivant.

La tabulation semble donc assez intuitive, surtout qu'on sait que le Reducer sépare nos clés des valeurs par des tabulations.

Cela permettra de faire la boucle à plusieurs passages Map/Reduce dont on avait parlé dans l'introduction.

Maintenant ce dont il faut se rappeler c'est que durant la première phase, les coûts possibles sont **0** pour le sommet de départ, et le coût infini pour tout les autres, représenté par le chiffre **-1**.

Pour les sommets sans voisins, la liste des voisins est vide, on aura donc seulement un espace blanc.

Le Map:

Durant la phase de map on veillera à éclater les sommets de notre input de façon a pouvoir évaluer les coûts pour ceux là (tout en prenant en considération la contrainte des coûts infinis représentés par des blancs dans notre fichier input).

Le Reduce:

Dans la phase du Reduce, ce qu'on fait c'est qu'on regroupe les sommets en fusionnant les entrées d'un même noeud, mais en ne gardant que la valeur minimale du coût entre tous ses coûts possibles.

Bien évidemment, si le sommet n'a qu'un seul coût possible, alors c'est celui là qui est gardé sans comparaison nécessaire.

Problémes rencontrés:

Durant l'établissement du TP j'ai eu affaire a plusieurs problèmes, notamment le comportement du Reducer et la façon dont il écrit dans le contexte, pour pouvoir établir une méthode pour pouvoir ré-exploiter les fichiers de sorties.

Il y' a aussi le problème de l'arrêt de l'algorithme, et la définition du point d'arrêt.

Cette problématique a été résolue en proposant un mécanisme de comparaison de sorties.

en effet pour savoir que notre solution est bien la finale, on compare chaque fois la sortie du passage Map/Reduce précèdent avec la sortie du passage Map/Reduce courant (hors le cas du premier passage) et si la solution ne change pas, alors on s'arrête car on a retrouvé la solution finale.

Ceci est une bonne solution dans le sens ou ce qui nous concerne plus c'est le résultat en soit.

Ceci dit, si on est vraiment concerné pas les performances, et qu'on suppose qu'on travaille sur un très grand nombre de lignes en entrées, le fait qu'on ajoute un passage Map/Reduce qui ne sert finalement a rien pourrait avoir des conséquences sur les performances du Programme.

L'impact ne sera sûrement pas aussi conséquent, mais ceci reste nommions une question à se poser.

Limites du programme:

1. Définition des coûts intermédiaires de manière embarquée:

J'ai pensé à imbriquer la matrice de coûts intermédiaires dans le code. et donc pour pouvoir ré appliquer l'algorithme sur un graphe différent il faudrait rechanger la matrice de coûts dans le code-source .

Ceci dit cette situation est la limite la plus considérable du code. On pourra dans le cadre d'un évolutivité du code penser à utiliser un mécanisme de chargement de cette matrice depuis un fichier externe par exemple.

2. Nombre de noeuds possibles 27 au maximum:

Comme dans notre parseur on travaille sur un alphabet, avec au maximum 27 lettres associables. Cela nous mets dans la contrainte si on voudrait entrer un graphe de plus de 27 de sommets.

3. Pas d'émit lors du regroupement des coûts intermédiaires:

Dans le code, on regroupe tous les coûts possibles pour un sommet pour pouvoir en extraire le minimum.

Or, on aurait pu aller plus loin, et chaque fois qu'on lit le premier coût pour un sommet, on ignore tous les coûts qui sont supérieur à celui là, comme ça on aura moins de comparaison a faire et donc on gagnera en terme de performances d'exécution.

4. Les nominations des sommets doivent êtres consécutives:

Pour que le code marche comme il faut que que les sommets déclarés en voisins soient déclarés dans l'ordre alphabétique.

Cette contrainte est du à notre matrice de distances dans laquelle, à chaque fois on va rechercher les coûts intermédiaires.

CONCLUSION

Dans ce TP nous avons pu voir de prés comment peut on faire executer plusieurs passages de Map/Reduce pour résoudre un problème du type plus court chemin, tout en ayant qu'un seul Mappeur et un seul Reduceur.

Maintenant qu'on a pu résoudre ce problème d'optimisation on peut imaginer qu'on peut appliquer la méme méthodologie pour d'autres problèmes d'optimisation dans le meme contexte de graphes.

En effet, Google et LinkedIn entre autres, utilisent ce genres de mécansimes (Map/Reduce) en d'autres langages tel que Javascript.