Sujet Projet 2015

Cloud privé SAAS - Traveling salesman problem

Votre entreprise accroît son nombre de commerciaux ainsi que le nombre de villes et pays démarchés. Pour réduire les coûts liés à leurs déplacements, la direction cherche à optimiser les voyages des commerciaux, c'est à dire minimiser les distances parcourues.

À cette fin, votre entreprise vient d'acquérir un cluster de serveurs destinés à accueillir tous les calculs de l'entreprise. Pour en simplifier l'utilisation, vous êtes en charge de la mise en place d'une plateforme cloud privé de type SAAS (Service As A Service). L'objectif de ce cloud est de permettre aux utilisateurs, via une application développée par vos soins, de distribuer automatiquement et efficacement des calculs sur le cluster de serveurs. Dans ce cas précis, l'application dont vous êtes en charge a pour but de distribuer les calculs liés à l'optimisation des trajets des commerciaux de l'entreprise.

Votre application devra ainsi mettre en place les fonctionnalités suivantes :

- **Planification de trajets pour les commerciaux**
- **Visualisation des trajets**
- **Tolérance aux pannes**

Traveling salesman problem

On souhaite qu'un commercial démarche des clients dans les villes de la liste L={v1, .., vn}. Pour optimiser son voyage, on cherche le plus court chemin: le chemin de longueur totale minimale qui passe exactement une fois par chaque ville et revient à la ville de départ. Il s'agit en fait de résoudre un problème bien connu des informaticiens : le "traveling salesman problem".

L'entreprise souhaite que vous obteniez une solution optimale à ce problème et vous optez donc pour une résolution utilisant un algorithme de type Brute force.

Brute force

L'algorithme brute force est une méthode exacte qui explore tous les solutions et retourne la solution optimale. Dans notre cas, "toutes les solutions" signifie explorer tous les voyages possibles passant par toutes les villes et la "solution optimale" est en fait le voyage le plus court.

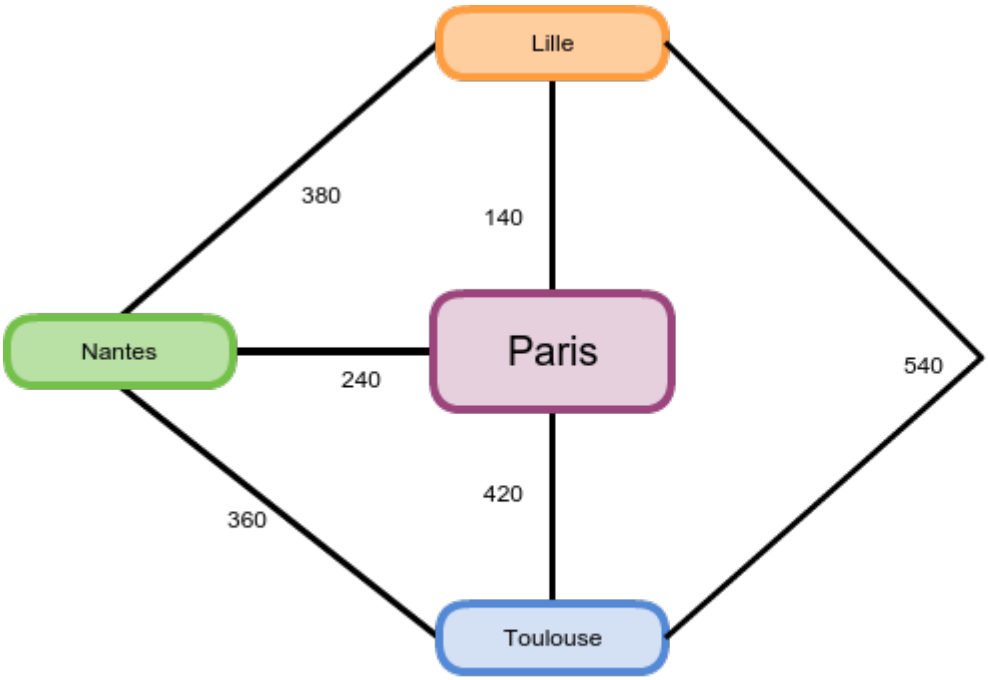
Pour résoudre le traveling salesman problem, nous générons donc toutes les permutations des villes. Chaque permutation correspond à un voyage. Nous calculons la longueur des chemins de chacune des permutations et gardons le chemin le plus court.

```
fonction brute_force (G : Graph, V(G): liste des villes)
 meilleur_permutation := null
 coût_minimal:= infini
 Pour chaque permutation pi de V(G) {
  Si (coût(pi, G)  < coût_minimal) {
   meilleur_permutation := pi
   coût_minimal:= coût(pi, G)
  }
 }
 retourner (meilleur_permutation, coût_minimal)
```

Implémentation

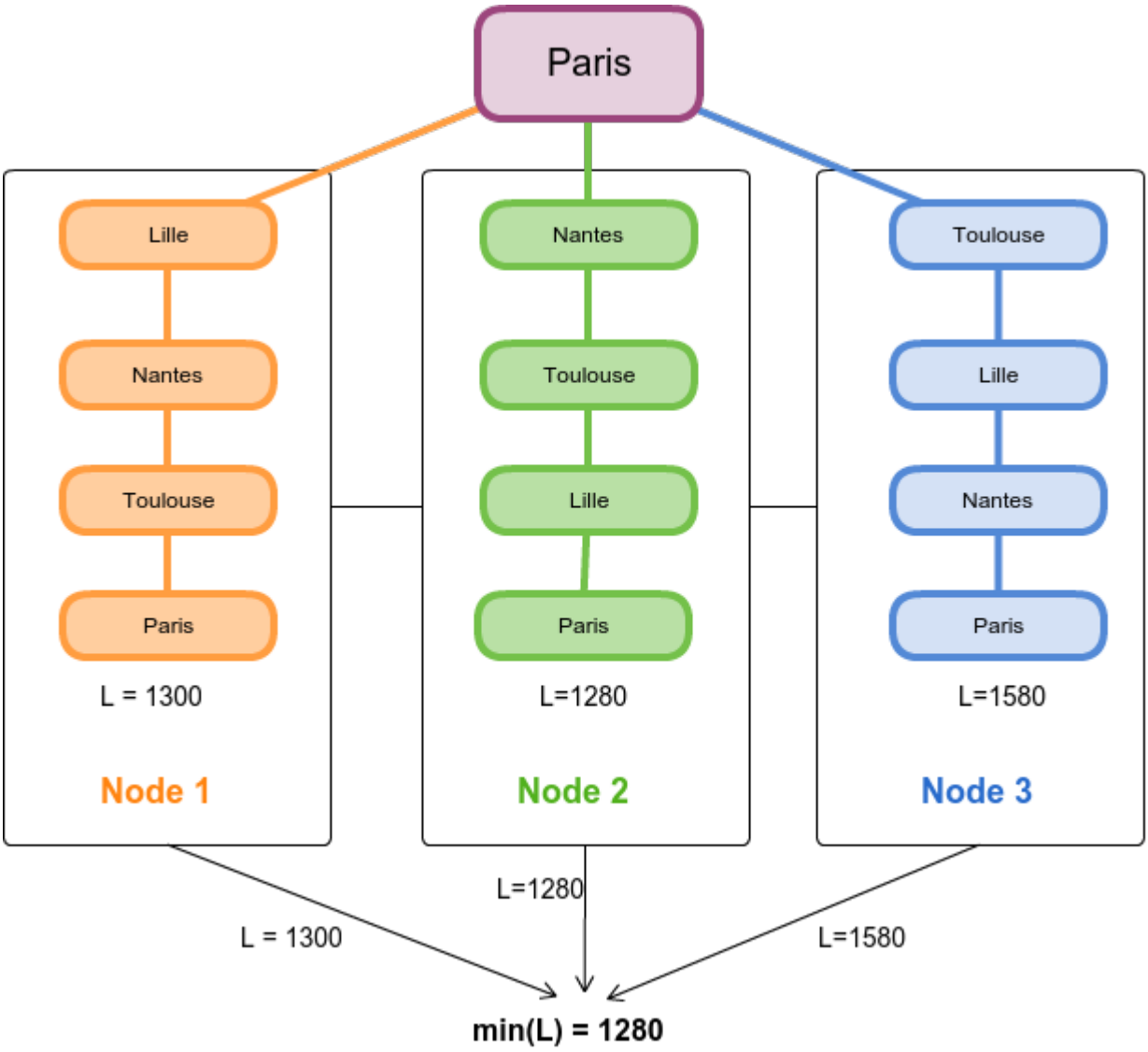
L'algorithme de résolution doit être implémenté de façon distribuée. Son implémentation doit être tolérante aux pannes : si une ou plusieurs machines tombent en panne, l'algorithme distribué doit retourner le résultat optimal et les permutations dont la longueur a déjà été calculée ne doivent pas être considérées une seconde fois.

Soit le graphe d'exemple suivant :



Pour résoudre le problème et trouver un chemin qui part de Paris, passe par toutes les villes et revient vers Paris, on utilise l'algorithme brute force cité ci-dessus de façon distribuée tel que ci-dessous.

Le premier noeud de calcul explore les chemins pour lesquels en partant de Paris, on choisit d'abord d'aller à Lille. Le second noeud explore quand à lui les chemins pour lesquels en partant de Paris, on choisit d'abord d'aller à Nantes. Enfin, le troisième noeud explore les chemins pour lesquels en partant de Paris, on choisit d'abord d'aller à Toulouse. Une fois tous les chemins explorés, c'est le chemin de longueur minimale qui est retenu.



Interface

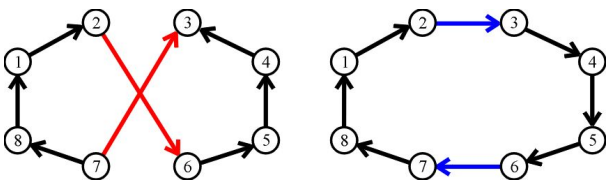
L'interface de l'application doit permettre de soumettre un graphe représentant une liste de villes par lequel les commerciaux doivent passer. Elle doit également être capable d'afficher les solutions retournées par l'algorithme distribué.

Par ailleurs, l'interface permet de visualiser les parcours calculés par l'algorithme, voire l'évolution des meilleures solutions obtenues lors du calcul du voyage le plus court. Pour cela, votre entreprise a retenu la librairie GraphStream qui permet de visualiser ce genre de données.

Bonus

L'entreprise est en pleine croissance. Les voyages des commerciaux seront bientôt de plus en plus longs. Les problèmes à résoudre seront donc de plus en plus difficile. Afin d'accélérer la résolution de ces problèmes, vous pouvez implémenter des heuristique de résolutions telles que 2-opt.

L'idée : on calcule un voyage qui va du départ à la destination puis on essaye itérativement d'échanger deux arêtes (route entre deux ville) du chemin comme présenté dans l'image ci-dessous :



L'algorithme est décrit ci-dessous. G décrit le graphe avec la liste de villes et les routes entre chaque ville. H est le voyage qu'on essaye d'optimiser.

```
fonction 2-opt ( G : Graphe, H : Voyage )
 amélioration : booléen := vrai
 Tant que amélioration = vrai faire
  amélioration := faux;
  Pour tout ville vi de H faire
   Pour tout ville vj de H, avec j différent de i-1 et i+1 faire
    Si distance(vi, vi+1) + distance(vj, vj+1) > distance(vi, vj) + distance(vi+1,
vj+1) alors
     Remplacer les arêtes (vi, vi+1) et (vj, vj+1) par (vi, vj) et (vi+1, vj+1)
dans H
     amélioration := vrai;
 retourner H
```

Données de test

Les jeux de test sont les fichiers text dans le format de TSPLIB:

http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/

Les jeux de test sont récupérés principalement dans:

http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/tsp/

Le format est présenté dans:

http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/DOC.PS

On ne considère que les bases du type de TSP (ligne 3 du fichier est: TYPE : TSP) et dans l'espace Euclidean 2D (ligne 5 du fichier est: EDGE_WEIGHT_TYPE : EUC_2D), par exemple la base http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/tsp/a280.tsp.gz

Dans ces bases, on suppose que le graphe des villes est complet. Les coordonnées des villes sont données dans l'espace Euclidean 2D et la distance entre deux villes vi(coordonnées: xi, yi) et vj(coordonnées: xj, yj) est calculée:

distance(vi, vj) = Racine_Carrée[(xi-xj)2+(yi-yj)2]

Pour mesurer la qualité de la solution trouvée par votre système, vous pouvez consulter les meilleur solutions pour ces jeux dans: http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/STSP.html

Pour chaque jeu de test, la sortie est exprimé dans un seul fichier text qui contient n+1 lignes, avec n le nombre de villes. La premiere ligne contient la longueur totale. Les lignes suivante donnent les villes dans le trajet.

Au départ, les tests peuvent être réalisées sur votre machine. Cependant, vous pourrez également tester votre réalisation en salle machines en utilisant des machines virtuelles Docker afin d'en vérifier la viabilité sur un système réellement distribué.

Des jeux de données plus petits et au même format sont à votre disposition ici :

http://www.math.uwaterloo.ca/tsp/world/countries.html
Ces bases de données contiennent des infos réelles :)
Les petites bases sont par exemple :

- 29 villes http://www.math.uwaterloo.ca/tsp/world/wi29.tsp
- 38 villes http://www.math.uwaterloo.ca/tsp/world/dj38.tsp

N'hésitez pas à tester votre algorithme sur de plus petites données encore au départ : des jeux que vous créez vous mêmes ou alors en effaçant certaines villes d'un jeu de données plus gros.

Sur ces petites bases que vous créez, vous pouvez trouver la solution optimale avec les logiciels suivants pour la comparer avec celle de votre propre algorithme :

http://www.math.uwaterloo.ca/tsp/concorde/

http://tspsg.info/en/download

Travail à réaliser

Vous devez constituer (librement) des groupes de 3 personnes. Chaque groupe devra développer le(s) application(s) décrite(s), en utilisant MAVEN sous l'environnement de son choix ; un repository Subversion mis à votre disposition ainsi qu'un déploiement continu de votre application et une analyse du code à l'aide de Jenkins/Sonar.
Pour le code, l'accent devra être mis sur une architecture modulaire et extensible à l'aide d'interfaces, la répartition équitable des

implémentations à réaliser au sein de l'équipe et la documentation du code (javadoc).
Vous serez évalués sur **l'ensemble de ces critères**.
A la fin du projet, vous devez fournir :

- le code source de l'application sur le repository subversion, compilable avec un mvn install
- une documentation technique (PAS utilisateur) décrivant l'architecture générale et le fonctionnement global de votre application, l'API d'accès aux données ainsi que l'ensemble des classes et les liens entre elles. Elle devra absolument décrire la manière dont vous gérez la distribution des calculs et la tolérance aux pannes qui constituent le coeur de votre application (environ 10 pages, à sauver en rapport.pdf à la racine du repository de votre projet).

Les serveurs Svn seront fermés le **17 mai à minuit.**

Ce projet donnera lieu à une soutenance orale de 20 minutes (démo comprise). Cette soutenance aura lieu le **18 ou le 19 mai.**