RANDIMBIVOLOLONA Fabien **Groupe 7**
PERNIN Dylan
LAUPRETRE Julien
PAGET Jules

Projet de Programmation Stochastique

Rapport Organique

Modélisation Informatique Recuit Simulé

Modèle en MVC, interface à l'aide de JavaFX.

Classes, fonctions et attributs

- Model.City

Classe qui permet de modéliser une ville, comprend un id qui permet d'identifier cette ville de manière unique, ainsi que deux coordonnées x et y qui représentent les positions x et y de la ville. Cette classe est aussi dotée de trois getters pour les variables id, x et y ainsi qu'une fonction toString() qui permet d'afficher une ville.

- Model.ParserTSP

Classe qui permet de parser un fichier au format .tsp
La fonction read prend un string en paramètre qui est la localisation du fichier + le nom du fichier tsp (ex : "C:\Users\didif\Documents\Polytech\ET5\Programmation Stochastique\sourcesSymmetricTSP\a280.tsp"). Cette fonction read nous donne en sortie une HashMap<Integer, City>.

- Model.Pair

Classe qui permet de gérer une paire d'entiers qui représente les ID de deux villes. Cette classe est utilisée pour l'algorithme du recuit simulé.

- Model.AlgoRecuitSimule

Classe qui implémente l'algorithme du recuit simulé.
hashMapCities correspond à une HashMap<Integer, City> représentant les villes.
size est l'entier représentant la taille de la matrice.
temperature est la température de départ de l'algorithme.
nbrIterPalier est le nombre d'itération par palier.
coolingFactor est le facteur de refroidissement.
toleratedAcceptanceRate est le taux d'acceptation.
stoppingNumber est le nombre max d'itération sans amélioration.

- Application.Controller

Classe qui permet de gérer les interactions de l'utilisateur avec la fenêtre.
openFile : fonction qui permet d'ouvrir un fichier .tsp.
drawCities : fonction qui permet de dessiner les villes sur le canvas. Si le paramètre drawnPaths est a "true", les chemins entre les villes sont affichés.
startTSP : fonction qui permet de lancer l'algorithme du recuit simulé sur le fichier importé.

Les actions associées aux boutons “Show cities” et “Start the TSP” sont lancées seulement si un fichier a été précédemment chargé.

- Application.Vue

Classe qui représente notre application (l’interface).

Représentation des données

Dans la classe AlgoRecuitSimule, on a considéré que les graphs étaient complets et symétriques, et on les représente sous forme de matrice comme ceci :

	Ville 1	Ville 2	Ville 3	Ville4
Ville 1	0	12km	5km	10km
Ville 2	12km	0	7km	24km
Ville 3	5km	7km	0	33km
Ville 4	10km	24km	33km	0

La case M i,j de la matrice M donnent la distance entre la ville i et la ville j, avec M i,j = M j,i dans le cas symétrique.

Pour modéliser les solutions ont utilise aussi une représentation matricielle :
Par exemple pour la solution 4->3->1->2->4 celle ci est représentée ci-dessous

	Ville 1	Ville 2	Ville 3	Ville4
Ville 1	0	1	1	0
Ville 2	1	0	0	0
Ville 3	1	0	0	1
Ville 4	0	1	1	0

Les cases S i,j de la matrice sont égales à 1 si l’on va de la ville i à la ville j lors du parcours dans notre circuit solution, et 0 sinon.

Modélisation Informatique CPlex et Java

Classes, fonctions et attributs

Pour résoudre ce problème avec CPLEX, nous avons décidé d'utiliser JAVA et plus précisément la librairie java cplex (ilog.cplex)

Il a fallu dans un premier temps extraire les données qui étaient au format XML du fichier a280.xml par exemple. Pour cela nous avons utilisé un tableau de classe City qui contenait comme attribut un dictionnaire avec comme clé la ville à atteindre et comme valeur le coût associé.

Une fois ce tableau rempli nous pouvions construire le problème CPLEX (caractérisé en java comme objet *IloCplex*) avec les méthodes *addTerm, addMinimize, addEq et addLe* pour la fonction objective et les contraintes.

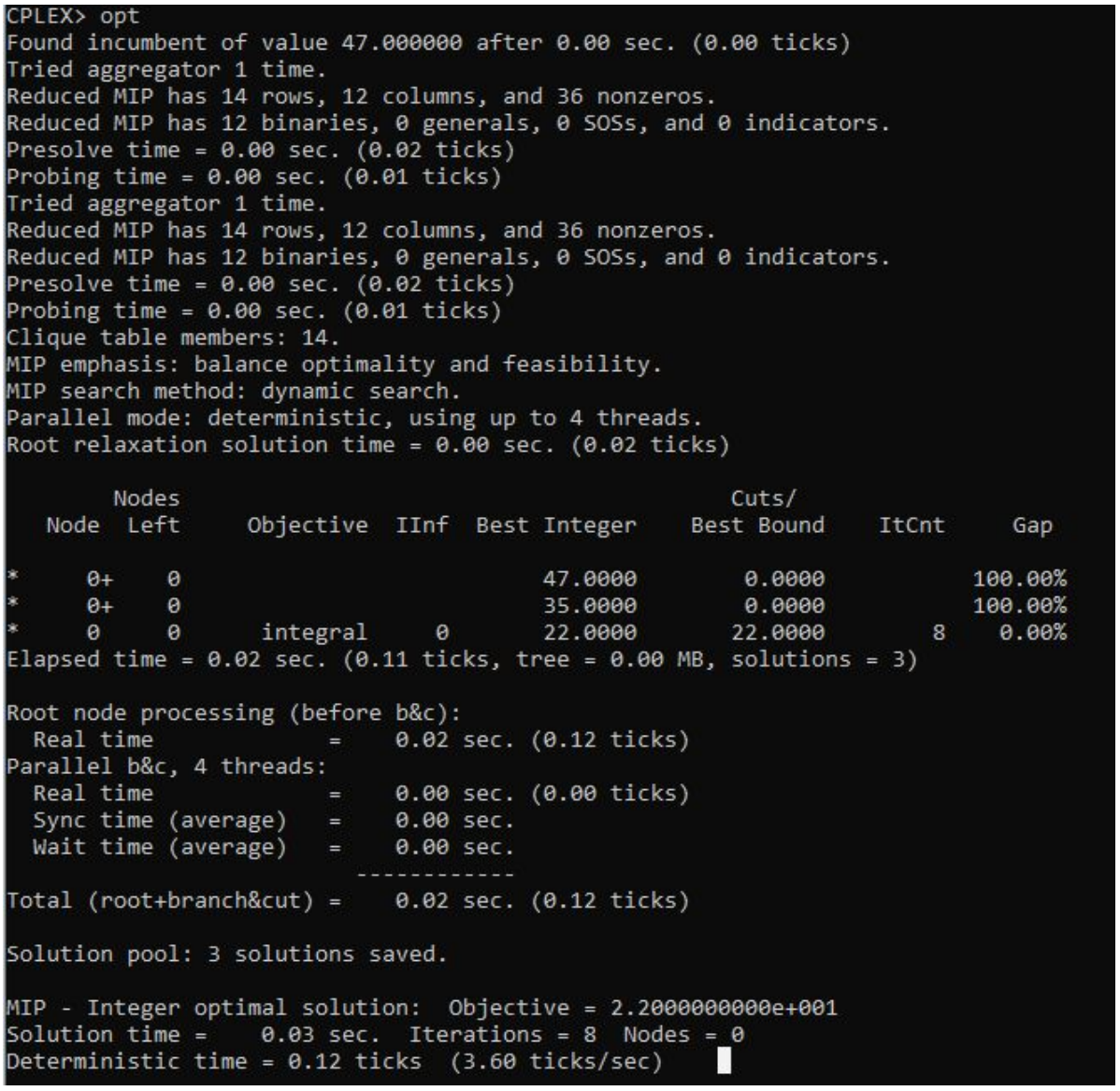
Une fois l'objet CPLEX correctement construit, il ne restait plus qu'à appeler la fonction *solve* sur cet objet pour lancer la résolution du problème.

Resultat de l’analyse CPLEX, avec peu de villes

Pour valider la théorie nous avons modélisé un problème du voyageur avec 4 villes et différents coûts (contenus dans la fonction objectif).

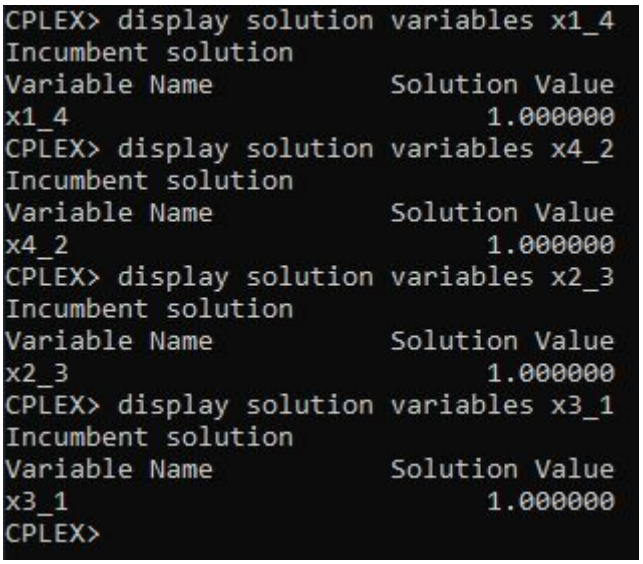
```
\Problem name: Exo.lp

Minimize
 obj: 10 x1_2 + 15 x1_3 + 2 x1_4 + 2 x2_4 + 3 x2_3 + 20 x3_4 + 10 x2_1 + 15 x3_1 + 2 x4_1 + 2 x4_2 + 3 x3_2 + 20 x4_3

Subject To

 c1: x1_2 + x1_3 + x1_4  = 1
 c2: x2_1 + x2_3 + x2_4  = 1
 c3: x3_1 + x3_2 + x3_4  = 1
 c4: x4_1 + x4_2 + x4_3  = 1

 c5: x2_1 + x3_1 + x4_1 = 1
 c6: x1_2 + x3_2 + x4_2 = 1
 c7: x1_3 + x2_3 + x4_3 = 1
 c8: x1_4 + x2_4 + x3_4 = 1

 c9: x1_2 + x2_1 <= 1
 c10: x1_3 + x3_1 <= 1
 c11: x1_4 + x4_1 <= 1
 c12: x2_4 + x4_2 <= 1
 c13: x2_3 + x3_2 <= 1
 c14: x3_4 + x4_3 <= 1

Bounds
  0 <= x1_2 <= 1
  0 <= x1_3 <= 1
  0 <= x1_4 <= 1
  0 <= x2_4 <= 1
  0 <= x2_3 <= 1
  0 <= x3_4 <= 1
  0 <= x2_1 <= 1
  0 <= x3_1 <= 1
  0 <= x4_1 <= 1
  0 <= x4_2 <= 1
  0 <= x3_2 <= 1
  0 <= x4_3 <= 1

Generals
 x1_2  x1_3  x1_4  x2_4  x2_3  x3_4  x2_1  x3_1  x4_1  x4_2  x3_2  x4_3

End
```

Programme CPLEX



Résultat CPLEX après la résolution

On trouve donc que le plus court chemin trouvé par CPLEX coute 22.



Variables des chemins à emprunter

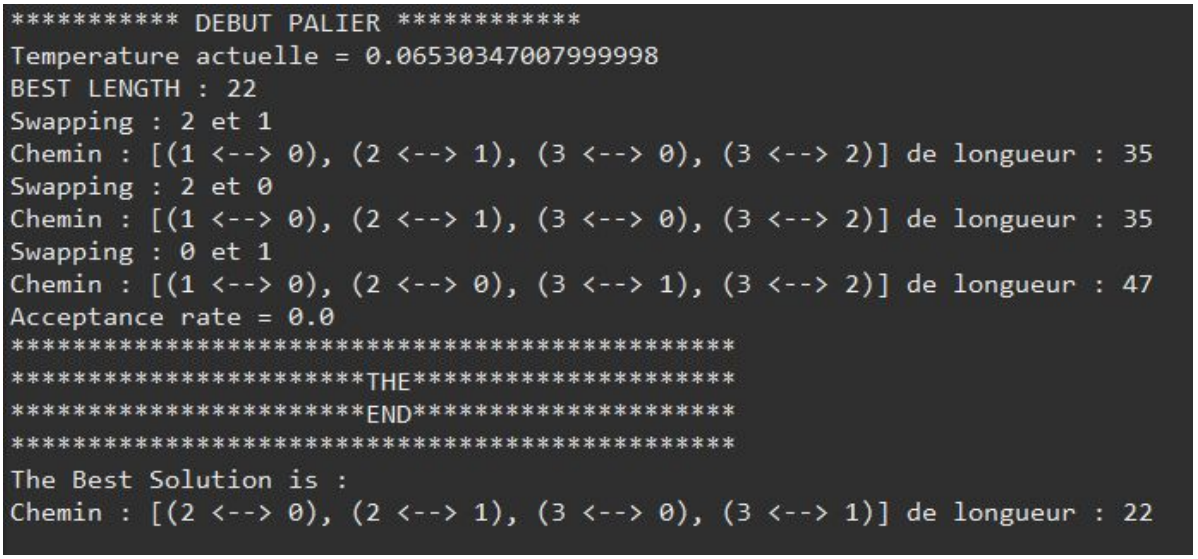
On obtient ici le chemin à prendre pour minimiser les coûts (en partant de 1) : **1-4-2-3-1.**

Résultat de l’analyse à l’aide de l'algorithme de Recuit Simulé, avec peu de villes

```
Matrice des distances entre villes :
[ 0 ][ 10 ][ 15 ][ 2 ]
[ 10 ][ 0 ][ 3 ][ 2 ]
[ 15 ][ 3 ][ 0 ][ 20 ]
[ 2 ][ 2 ][ 20 ][ 0 ]
Chemin : [(1 <--> 0), (2 <--> 1), (3 <--> 0), (3 <--> 2)] de longueur : 35
********** DEBUT PALIER ***********
Temperature actuelle = 50.0
BEST LENGTH : 35
Swapping : 0 et 3
Chemin : [(2 <--> 0), (2 <--> 1), (3 <--> 0), (3 <--> 1)] de longueur : 22
Solution : 22 acceptee
Swapping : 2 et 3
Chemin : [(2 <--> 0), (2 <--> 1), (3 <--> 0), (3 <--> 1)] de longueur : 22
Solution : 22 acceptee
Swapping : 0 et 3
Chemin : [(1 <--> 0), (2 <--> 1), (3 <--> 0), (3 <--> 2)] de longueur : 35
Acceptance rate = 0.6666666666666666
********** DEBUT PALIER ***********
Temperature actuelle = 30.0
BEST LENGTH : 22
Swapping : 1 et 0
Chemin : [(1 <--> 0), (2 <--> 0), (3 <--> 1), (3 <--> 2)] de longueur : 47
Swapping : 0 et 2
Chemin : [(2 <--> 0), (2 <--> 1), (3 <--> 0), (3 <--> 1)] de longueur : 22
Solution : 22 acceptee
Swapping : 2 et 0
Chemin : [(1 <--> 0), (2 <--> 0), (3 <--> 1), (3 <--> 2)] de longueur : 47
Solution : 47 acceptee
Acceptance rate = 0.6666666666666666
********** DEBUT PALIER ***********
Temperature actuelle = 18.0
BEST LENGTH : 47
Swapping : 2 et 0
Chemin : [(2 <--> 0), (2 <--> 1), (3 <--> 0), (3 <--> 1)] de longueur : 22
Solution : 22 acceptee
Swapping : 3 et 2
Chemin : [(2 <--> 0), (2 <--> 1), (3 <--> 0), (3 <--> 1)] de longueur : 22
Solution : 22 acceptee
Swapping : 3 et 0
Chemin : [(1 <--> 0), (2 <--> 1), (3 <--> 0), (3 <--> 2)] de longueur : 35
Acceptance rate = 0.6666666666666666
```

Déroulement de l’algorithme du recuit simulé avec comme paramètres :

- ***Température initiale** = 50*
- ***Nombre d’itérations par palier** = 3*
- ***Facteur de refroidissement** = 0.6*
- ***Seuil de taux d’acceptation** = 30%*
- ***Seuil d’arrêt** = 3*
- ***Solution de départ** = 0 -> 1 -> 2 -> 3 -> 0 de longueur 35*



Affichage de la solution après le déroulement de l'algorithme

On obtient dans ce cas ci que le chemin optimal est de 0 -> 2 -> 1 -> 3 -> 0
(soit 1 -> 3 -> 2 -> 4 -> 1) d'une longueur de 22. On trouve donc le même résultat qu'avec CPLEX.

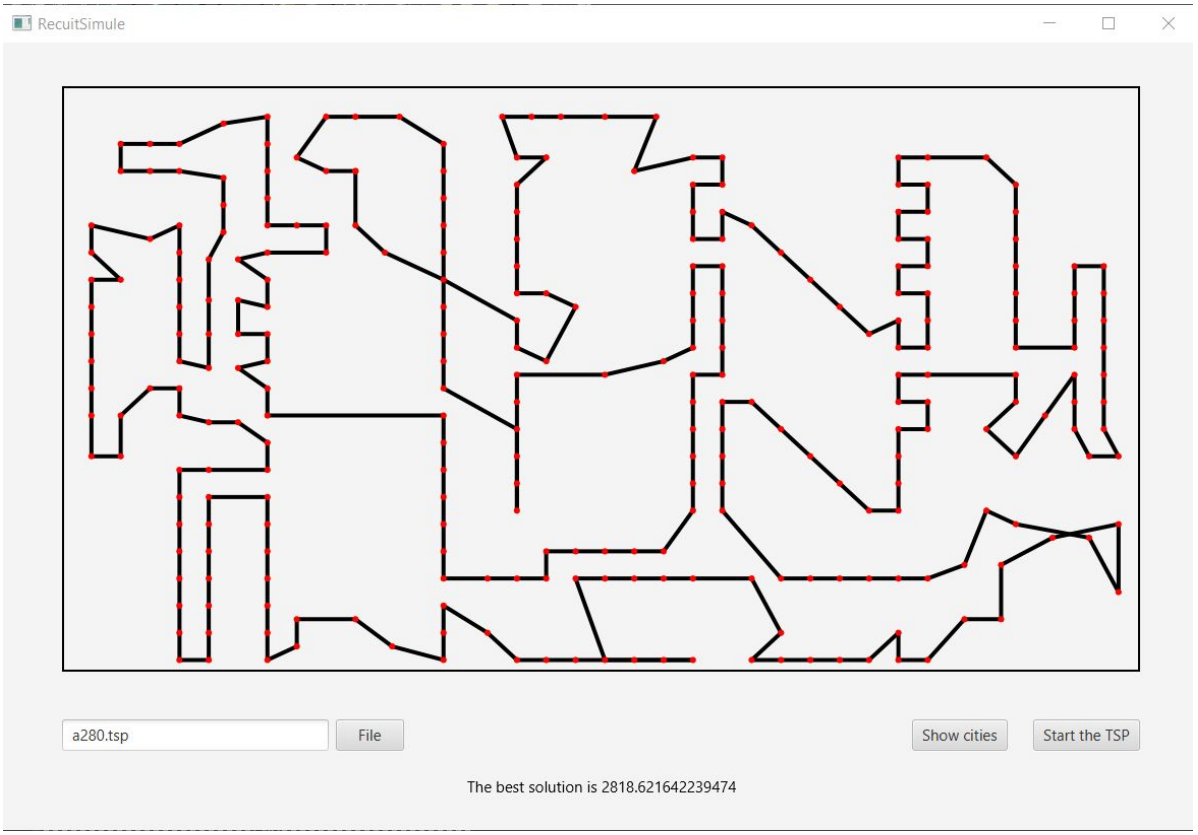
Analyse des deux méthodes avec plus de villes et affichage graphique

Nous avons pu voir que sur un exemple de taille modeste, il n'y avait aucune différence de temps mais pour la suite nous avons comparé l'efficacité des 2 approches sur des exemples bien plus conséquents.

Pour cela, nous avons essayé de créer un fichier de format lp pour CPLEX à partir des données en format XML, cependant nous rencontrons des difficultés et nous avons par la suite utilisé la librairie CPLEX directement dans java pour contourner ce problème.

Notre interface concerne seulement la partie Recuit Simulé. Elle permet de sélectionner un fichier tsp à importer dans la solution. On peut ensuite afficher seulement les villes ou lancer la résolution du TSP à l'aide de l'Algorithme du recuit simulé.

Recuit Simulé



Résultat obtenu pour le fichier à 280 villes avec une distance de 2818,62.

Pour ce résultat, nous avons utilisé :

- ***Température initiale** = 1000*
- ***Nombre d'itérations par palier** = 20;*
- ***Facteur de refroidissement** = 0.9*
- ***Seuil de taux d'acceptation** = 30%*
- ***Seuil d'arrêt** = 15*

Le résultat obtenu est de 2818, ce qui est supérieur au résultat de la littérature qui est lui de 2579.

Nous pourrions obtenir de meilleurs résultats en modifiant les paramètres d'entrée de l'algorithme :

- Modification de l'algorithme naïf de la solution initiale
- Température initiale
- Nombre d'itérations par palier
- Facteur de refroidissement
- Seuil de taux d'acceptation
- Seuil d'arrêt

→ Faire un grand nombre de tests pour optimiser ces paramètres en fonction du nombre de villes.

CPlex

Résultat CPLEX java déterministe pour différents fichiers de données :

Nom du fichier	Nombre de ville	Résultat optimal / trouvé	Temps
burma14	14	3323 / 3323 ✔	0.20 sec
att48	48	10628 / 10628 ✔	223.95 sec
a280	280	2579 / 2975 ❌	11035.39 sec (3 h) *

* Arrêt après une erreur de mémoire

On remarque donc que le temps d'exécution augmente exponentiellement, de plus le résultat pour le fichier contenant 280 villes n'est relativement pas optimal.

La résolution déterministe n'est donc pas optimale pour un grand nombre de villes.

Conclusion

Pour conclure, on peut donc dire que la méthode avec CPlex est plus précise pour des problèmes de petite taille, mais le calcul devient vite trop long.
La méthode du Recuit Simulé est quant à elle relativement efficace mais il n'y a pas de certitude de meilleur résultat. Ce qui est le plus important pour cette méthode, ce sont les paramètres.