RAPPORT DU PROJET

Programmation par contraintes

1. Introduction

La programmation par contraintes (noté plus tard PPC) permet de résoudre des problèmes combinatoires. A l'instar de la programmation linéaire, qui consiste à rechercher un algorithme général capable de résoudre un problème à une situation donnée, la PPC, elle, sépare la partie modélisation de la partie résolution. C'est dans cette deuxième partie que seront énoncé les différentes contraintes à respecter.

2. Choix de la librairie

Deux choix s'offraient à nous : ChocoSolver et Or-Tools.
Le premier est une librairie gratuite et *open source* écrite en java et dédiée à la programmation par contraintes. Cet outil disposait d'une documentation assez fournie pour nous mettre « dans le bain » de la programmation par contraintes. Nous avons notamment pu résoudre le problème du sudoku, ainsi que celui du sac à dos. Cependant pour résoudre un problème tel que la planification d'une chaine de livraison qui se modélise par un graphe, ChocoSolver n'était pas très adapté notamment à cause de la manipulation des graphes, qui semblait simple au début mais uniquement pour des problèmes simples de planification tel que celui du voyageur de commerce qui ne se réduit qu'à un seul graphe orienté où l'on ne passe qu'une seul fois sur chaque nœud.

OR-Tools, quant à lui, est un outil qui permet l'optimisation de résolution de problème. Écrit en C++ et utilisable dans quatre langage (C++, Java, C# et Python) et développer en interne par google, ce dernier est, à première vue, plus compliqué à installer/utiliser, mais dispose d'une librairie spécifique aux problèmes de planification de trajets. C'est pourquoi nous nous sommes tournés vers ce deuxième choix dans le cadre du projet SmartUHA.

3. Résolution du problème

Le principe de résolution d'un problème en programmation par contraintes est le même pour les deux outils, ainsi que toutes les librairies de programmation par contraintes.
Il consiste à instancier un modèle dans lequel on y définit ses variables. Puis à définir les contraintes à imposer sur ces variables. Enfin, il suffit d'initialiser un solveur depuis le modèle. Dans ce solveur, on peut y paramétrer les conditions de recherche de solution, ou encore optimiser la solution en définissant les contraintes et/ou les valeurs à perfectionner.

Dans le cadre de projet plus conséquent, il est nécessaire de transformer les données reçus en entrée puisque les seuls types de variable acceptable dans le modèle sont les valeurs entières et les valeurs réelles.

4. SolverSmartUHA

4.1. Description générale du projet

Des véhicules électriques doivent être capable d'effectuer des missions de transport de colis sur le campus de l'Illberg. Des membres du personnel de l'UHA effectue des demandes pour transporter une marchandise (colis ou lettre à taille et poids variable) d'un point A vers un point B. Les véhicules, possédant un certain nombre de caractéristiques, s'occuperont des différentes demandes.

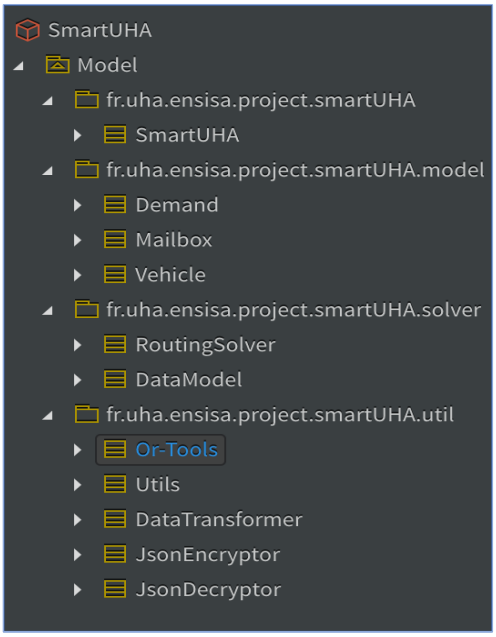
4.2. Notre tâche dans le projet

Notre objectif consiste donc à planifier le trajet des véhicules en fonction des demandes des clients. Cette planification est sujette à un certain nombre de contraintes :

- Les véhicules ont une certaine autonomie qu'ils ne doivent pas dépasser.
- Les véhicules ont une certaine capacité de stockage à ne pas surpasser.
- Une demande ne peut être traitée que par un véhicule.
- Pour traiter une demande, un véhicule doit d'abord passer par le point de chargement avant d'aller au point de déchargement.
- Le véhicule doit arriver au point de chargement à une heure précise définit par l'utilisateur.
- L'ensemble des demandes doivent être traitées en un temps minime.

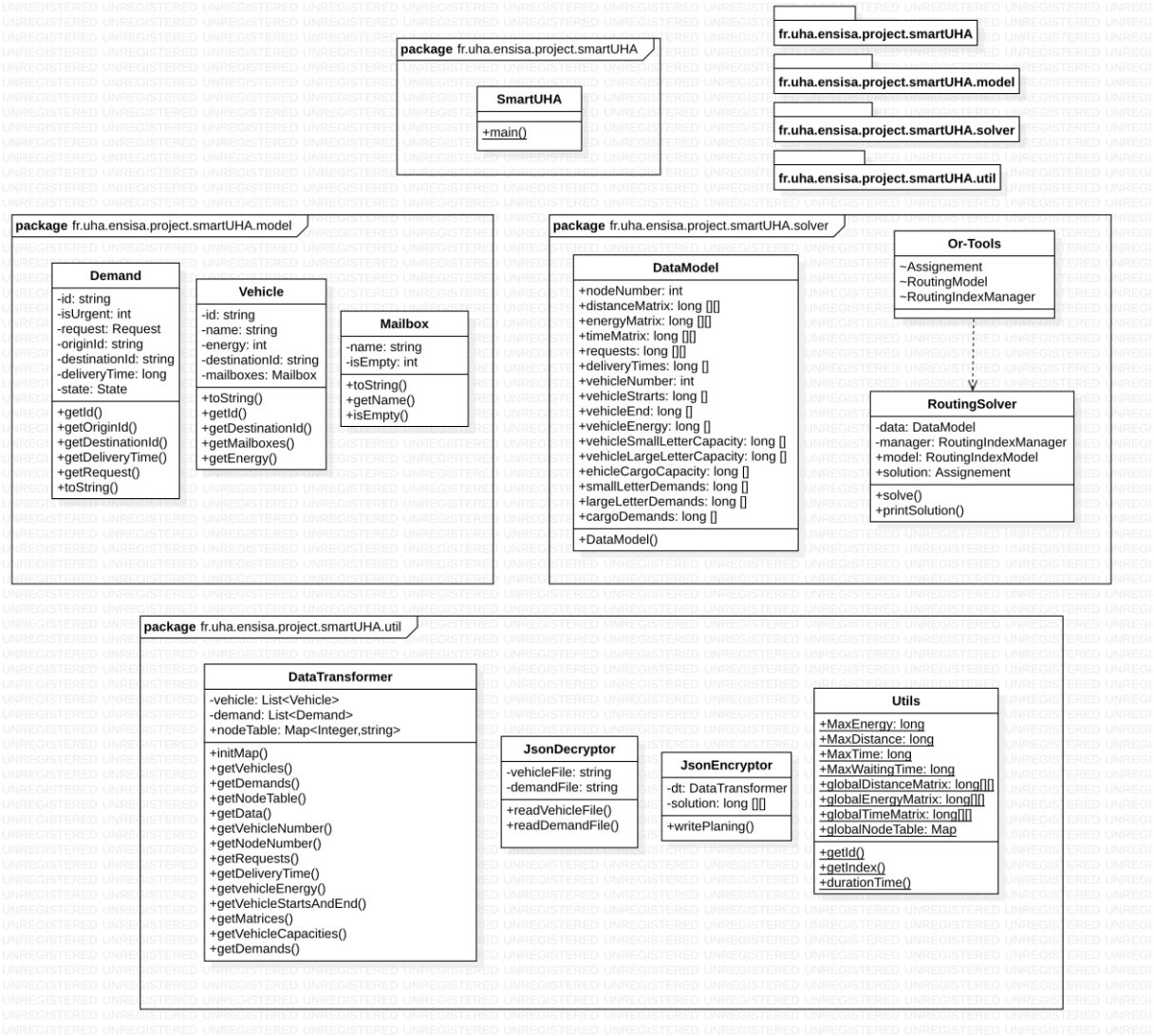
5. Processus de résolution

- Structure hiérarchique du programme :



Organisation des différents packages et leurs contenus sous Eclipse.

- Diagramme UML du projet



5.1. Récupération des données

Les informations sur les véhicules et les demandes nous sont envoyées, depuis l'API client, à l'aide de deux fichier JSON. Le premier fichier nous renseigne sur les demandes en cours (point de chargement, de déchargement, heure de chargement et type de colis envoyé). Le second nous renvoie l'état des différents véhicules disponible (position, autonomie, capacité).

Quant aux données de distance, de temps et d'énergie requise entre les points, nous sommes censés les recevoir par les intervenants qui s'occupent du pilotage des véhicules. Cependant, nous n'avons pas plus d'indication sur la façon dont cela doit être fait. A la place, nous avons donc instancier une matrice de distance, de temps ainsi qu'une matrice d'énergie en adéquation avec celles utilisées par l'autre groupe qui travaille sur l'API utilisateur.

Technique :
La lecture/écriture des données se fait dans le package *ortools.routing.utils.*
La classe s'occupant de lire les fichiers JSON est *JsonDecryptor*. Elle prend comme paramètres deux chaines de caractères qui correspondent aux fichiers à décrypter.
Elle comporte deux méthodes *readVehicleFile* et *readDemandFile* qui renvoient respectivement une liste de *Vehicle* et une liste de *Demand*.

5.2. Transformation des données

5.2.1. Classes principales

DataTransformer : A partir des données lues dans la classe *JsonDecryptor*, il faut transformer ces données pour qu'elle soit utilisable dans le modèle du solveur. C'est le but de la classe *DataTransformer* qui prend comme paramètres une liste de véhicules et une liste de demandes. Elle appelle la méthode *getData(DataModel)* qui consiste à initialiser les attributs de *DataModel*.

DataModel : Cette classe va contenir toutes les variables (données) nécessaires au solveur pour établir une planification.

Utils : Elle peut être considérée comme une classe annexe, principalement chargée de conserver les données utiles à l'étude telles que les distances entre chaque dépôt du campus de l'Illberg, l'énergie ainsi que le temps nécessaire pour y parvenir.

RoutingSolver : C'est la classe qui contient l'algorithme majeur de notre projet, elle va être chargée de récupérer les données contenues dans la classe DataModel, et essayer de trouver une solution optimale de planification en établissant diverses contraintes réalisables sur les données reçues.

5.2.1. Classes secondaires

Les classes qui vont suivre ont été créées dans un souci de modélisation des données du problème, qui seront entre autres utile pour le DataTransformer.

Demand : Elle permet de modéliser une demande provenant d'un client en y associant toutes les informations utiles reçus dans le fichier demand.json.

Mailbox : : Elle permet de modéliser un compartiment du véhicule autonome, notamment par son nom et son contenu (vide ou occupé) ;

Vehicle : Elle permet de modéliser un véhicule en y associant tous ses attributs, notamment son énergie, sa capacité, sa disponibilité, son identifiant… reçus dans le fichier vehicle.json.

JsonEncryptor : Classe contenant une méthode writePlaning, qui va permettre une fois une solution trouvée de pouvoir la restituer à l'interface client sous forme d'un document Json qui sera rendu lisible par l'interface client.

Main : C'est la classe principale permettant de lancer le processus de résolution.

5.3. Énonciation des contraintes

Maintenant que nous avons toutes les données utiles à l'écriture des contraintes, il est temps d'instancier la classe *RoutingSolver* ayant pour attribut un jeu de données *DataModel*.
Le processus d'écriture des contraintes est presque le même pour chacune des contraintes à quelques détails près lorsque l'on utilise les bons outils fournis par la bibliothèque de routage de Or-Tools.

- L'ajout des contraintes portant sur la distance à parcourir un véhicule, permettant d'y associer un coup entre les différents trajets du campus, qui sera cumulé par chaque véhicule lors de son déplacement.
- L'ajout des contraintes portant sur l'énergie (autonomie) de chaque véhicule, définit par l'évaluation à chaque instant du niveau d'énergie du véhicule afin de s'assure si la tâche est réalisable ou non.
- L'ajout des contraintes portant sur le temps de chargement et de livraison, devant être respecter pour le véhicule chargée de ladite commission.
- L'ajout des contraintes portant sur la capacité du véhicule, caractérisé par une charge maximale et le fait de prendre le plus de colis possible.

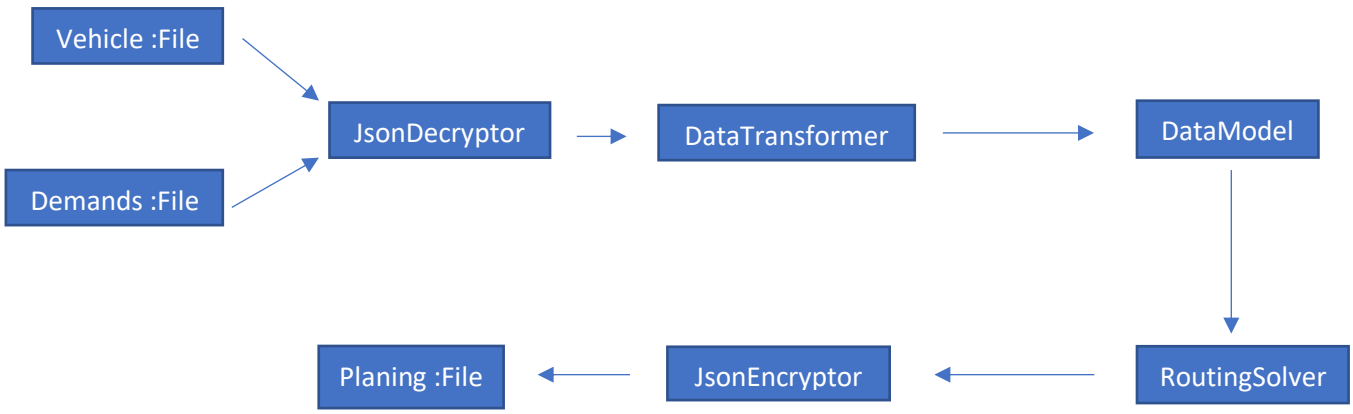
5.4. Processus de traitement

Deux fichiers au format json sont reçus :

- Demand.json : il contient toutes les demandes formulées par les clients et devant être traitées.
- Vehicle.json : il contient tous les véhicules disponibles et leurs différentes caractéristique, nécessaires à la planification.

Après le traitement des données et la résolution des contraintes, une solution sera écrite dans un autre fichier au format json :

- planing.json : il contiendra quant à lui les itinéraires associés à chaque véhicule afin de satisfaire au mieux les demandes des clients.



5.5. Idées pour continuer le projet

- Trouver un moyen de gérer les demandes urgentes lorsqu'il y'en a plusieurs.
- Établir une solution radicale ou non en cas de non satisfaction de toutes les demandes.
- Lire directement les données topologiques stockées dans une base de données, en communiquant avec l'autre système en temps réel, de même que pour les échanges de fichier entre les deux systèmes.
- Pouvoir établir une matrice de l'énergie à fournir par les véhicules entre les différents points du site, ce qui permettra de supprimer la matrice des distances, car il n'existe pas réellement des contraintes à formuler sur les distances à parcourir…

Lien du projet existant : https://github.com/AnisMessaoudi/SolverSmartUHA