Rapport projet : programmation par contraintes

# 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.

# 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.

OR-Tools, quant à lui, est un outil qui permet l’optimisation de résolution de problème. Écrit en C++ et utilisable dans quatre language (C++, Java, C# et Python), 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.

# 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.

# SolverSmartUHA

## 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.

## 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.

# Processus de résolution

## 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 et de temps en adéquation avec celle utilisé 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*.

## Transformation des données

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*, la classe qui contient toutes les variables nécessaires au solveur.

## Enonciation 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*.

## Résolution du problème