LENARTOWICZ Sacha
MARTIN Sébastien
TISSIER Alexandre

Projet Optimisation Stochastique

-

Document organique

Sommaire

I - Choix du langage et de la technologie

A - Langage utilisé

Ce projet autour du problème du VLS (Vélos en Libre Service) implique l'implémentation d'une interface graphique pour en faciliter l'utilisation. Deux solutions s'offraient à nous, à savoir une implémentation en Java ou en C++. Nous avons opté pour la deuxième option, afin de pouvoir bénéficier du framework Qt (également disponible pour Java) et de retrouver un langage que nous affectionnons. C'est pour nous l'occasion d'élaborer un GUI en abordant une nouvelle technologie, puisque nous n'avons jusqu'alors utiliser des bibliothèques Java pour nos projets.

B - Design de l'interface

Le framework Qt possède de nombreux avantages. Le premier est qu'il est multi-plateforme. De plus, cette technologie est aujourd'hui employée par de nombreuses entreprises, elle n'est donc pas désuète. Enfin, il semble que la mise en place de ce framework soit plus aisée que pour les bibliothèques équivalentes en Java, et autres bibliothèques C++. En effet, il n'est possible d'appliquer le modèle MVC (Model-View-Controller) que d'une façon légèrement différente : Qt intègre le contrôleur à la vue, grâce à la méthode connect() qui relie un widget à une action. De ce fait, l'architecture du code est simplifiée et permet une meilleure compréhension du modèle, tout en gardant une certaine souplesse. Nous utiliserons donc le design-pattern MV (Model-View) pour l'application demandée.

Autre avantage de Qt : la disponibilité de nombreux modules. Notre application doit permettre l'affichage d'une carte où seront placées les différentes stations. Le module QtLocation va nous permettre de facilement effectuer cette tâche. La documentation sur Qt est riche et très bien développée. De plus, il est possible de retrouver, dans la documentation en ligne, de nombreux exemples illustrant la mise en place du module. L'utilisation d'un tel outil ne nécessite pas de nouvelle instanciation, la carte est simplement paramétrée dans un fichier qml dont il faut indiquer l'emplacement. L'exemple qui nous intéresse plus particulièrement est consultable à l'adresse suivante : http://doc.qt.io/qt-5/qtlocation-places-map-example.html

Le résultat doit ressembler à ceci :

Figure 1 : exemple d'utilisation de QtLocation

C - Récupération des données

L'API est une fonctionnalité d'un site web qui permet à des programmes de récupérer directement certaines données de ce site web. L'API que nous allons utiliser est décrite sur ce lien : https://github.com/NABSA/gbfs/blob/master/gbfs.md.

Malheureusement cet API ne fonctionnait pas très bien et n'était pas simple d'utilisation. Pour régler ce problème le professeur nous a envoyé un fichier json que nous avons utilisé pour implémenter nos stations mais aussi nos scenarios.

Nous avons dû récupérer sur internet un fichier .hpp qui lisait les fichiers json pour qu'on puisse s'en servir car C++ ne dispose pas de format définit (comme Java). Voici le dossier que nous avons récupérer https://github.com/nlohmann/json. Cela permet d'avoir une classe json et de se balader dans notre fichier beaucoup plus aisément.

Grâce à cette librairie, nous pouvons alors pour notre programme créer un objet json. Il suffit alors de rediriger dans cet objet json la sortie standard du fichier contenue dans un objet ifstream (fichier >> json). L'option -std=c++11 est à rajouter pour l'exécution. Pour la récupération des données, notre objet json est un tableau contenant pour chaque indice les informations pour un élément indiqué entre crochets, ici une station. Il suffit alors d'indiquer pour cette station les noms des attributs que l'on veut récupérer dans des types String. Le type de l'objet est automatiquement détecté et est récupéré dans notre variable (de type double, int ou string). Le stockage des stations à l'exécution est alors réalisé à l'aide d'une boucle sur tout le tableau json.

II - Description des classes utilisées

Comme expliqué auparavant, nous utiliserons un design pattern proche du MVC (Model-View-Controller) appelé MV (Model-View) et adapté à l'utilisation du framework Qt.

A - Partie Modèle

Abstract Class Probleme

- solutions : vector<int>// les solutions X du problème.
- mesScenarios : vector<Scenario> // Liste de scénarios d'un problème.
- monAlgo : Algorithme* // Permet d'appeler les différents algorithme sur les scénarios dont on dispose dans la classe problème.

- abstract verfiContraintes(vector<int> solutions) : bool // Cette méthode permet de vérifier les contraintes de notre modèle mathématique. Cette fonction est donc définit dans les classes filles car cela dépendra des différents modèles.
- abstract calculerFctObj(vector<int> solutions) : float // Calcul la valeur de la fonction objective du modèle mathématique. Cela sera aussi définis dans les classes filles car cela dépend du modèle.

- getSolutions() : vector<int> // Renvoie la solution au problème.
- getMesScenarios() : vector<Scenario> // Renvoie la liste des scénarios.
- getMonAlgo() : Algorithme // Renvoie l'algorithme utilisé.
- setSolutions(vector<int>) // Modifie la liste des solutions.
- setMonAlgo(Algorithme *) // Modifie l'algorithme utilisé.
- setScenarios(vector<Scenario>) // Modifie la liste de scénario

Class VLS implements Probleme

- filename : string // Nom du fichier json à traiter.
- mesStations : vector<Station> // Liste des stations du problème VLS.

- calculerFctObj(vector<int>, int) : float // Calcul la valeur de la fonction objectif pour la solution passée en paramètre en fonction des scénarios et des stations avec la prise en compte de la penalite.
- verifContrainte(vector<int>) : bool //Vérifie le respect des contraintes pour le VLS.
- retrieveData (int) //retrieveData récupère les données sur le fichier json et initialise le scénario "moyen" à partir duquel seront créé tous les scénarios.
- getMesStations() : vector<Station>// Renvoie la liste des stations.
- setFileName(string name) : void //Modifie le nom du fichier json à traiter.

- generationScenario() : void // Cette méthode va nous permettre de créer des scénarios qui serviront pour le recuit stochastique et l'algorithme du SAA en partant du scénario moyen donné. Cette fonction génère un flux de demande aléatoire entre les stations tel que $\xi_{ij} \in [\lfloor \xi_{ij} \rfloor_{moyen}-20\%, \lfloor \xi_{ij} \rfloor_{moyen}+20\%]$. Ces demandes sont stockées dans de nouveaux objets scénario. Le scénario une fois créé, est ajouté dans la liste des scénarios.
- generationAlea(int gmin, int gmax) : int // Génère un chiffre aléatoire entre gmin et gmax
- generationAlea(int gmin, int gmax) : double // Génère un double aléatoire entre gmin et gmax

Classe Station

- nom : String // Nom de la station
- loc : Localisation // Sa localisation
- nbVelo : int // nombre de vélos
- capacite : int // capacite
- coutVelo : int // coût d'achat d'un vélo pour la station i
- coutDeficit : int // coût lié à un déficit de vélo dans la station i
- coutSurplus : int // coût liée à un surplus de vélo dans la station i

- getNom() : String // Renvoie le nom de la station.
- getLoc() : Localisation // Renvoie la localisation de la station.
- getNbVelo() : int // Renvoie le nombre de vélos.
- getCapacite() : int // Renvoie la capacité .
- getcoutVelo() : int // Renvoie le coût d'achat d'un vélo.
- getCoutDeficit() : int // Renvoie le coût lié à un déficit de vélo.
- getCoutSurplus() : int // Renvoie le coût liée à un surplus de vélo.
- setNom(String) // Modifie le nom de la station.
- setLocalisation(Localisation) // Modifie la localisation de la station.
- setNbVelo(int) // Modifie le nombre de vélos.
- setCapacite(int) // Modifie la capacité.
- setCoutVelo(int) // Modifie la coût d'achat d'un vélo.
- setCoutDeficit(int) // Modifie le coût lié à un déficit de vélo.
- setCoutSurplus(int) // Modifie le coût liée à un surplus de vélo.

Class Localisation

- x : double // Localisation en x.
- y : double // Localisation en y.

- getLocalisationX() : int // Récupère la localisation en x.
- getLocalisationY() : int // Récupère la localisation en y.
- setLocalisationX(int) // Modifie la localisation en x.
- setLocalisationY(int) // Modifie la localisation en y.

Class Scénario

- donnees: vector<int> // En fonction de la position dans le tableau et du nombre de station nous connaitrons le nombre de vélo qui va d'une station i à une station j.
- solution_scenario : vector<int> // Solution actuelle du scénario

- getDonnees() : vector<int> // Récupère le tableau de demande entre les stations.
- setDonnes(vector<int>) // Modifie les demandes entre les stations.
- getSolutionScenario() : vector<int> // Récupére la solution actuelle du scénario.
- setSolutionScenario(vector<int>) //Modifie la solution actuelle du scénario.

Abstract Class Algorithme

- monProbleme : Probleme *// Le problème mathématique que l'on veut résoudre.
- abstract executer(vector<int>) : vector<int> //La fonction executer() retourne la solution X du problème considéré selon l'algorithme choisi. Elle est définie dans les classe filles car cela va dépendre de l'algorithme que l'on va utiliser.
- getMonProbleme() : Probleme *// Renvoie le problème mathématique.
- setMonProbleme(Probleme*) // Modifie le problème mathématique.

Class CPLEX

- txtmodèle : File // Modèle mathématique remplit dans un fichier texte.
- executer(vector<int>) : vector<int> //executer() retourne le résultat de la solution exacte x après l'appel à CPLEX.

Abstract Class Recuit implements Algorithme

- temperature : float // La température pour le recuit.
- nbIterations : int // le nombre d'itérations pour le recuit.
- palierTemperature : int // La valeur dont on baissera la température à chaque passage dans la condition 1 du recuit

- abstract executer() : int[ ] //exécution du recuit soit pour le déterministe soit pour le stochastique.
- voisin(vector<int>) : vector<int> // Fonction de calcul de la solution voisine, elle calcule et vérifie les contraintes en fonction de cette solution.
- recuit(vector<int>, int) : vector<int> // Fonction qui implémente l'algorithme de recuit simulé vue en cours avec la pénalité voulu.

Class RecuitDeterministe implements Recuit

- executer(vector<int>) : vector<int> // Exécute la fonction de recuit simulé pour un seul scénario .

Abstract Class Stochastique implements Recuit

- recuitStochastique() : vector<int> // Utilise plusieurs fois la fonction recuit de la classe recuit sur les différents scénarios.
- abstract executer(vector<int>) : vector<int> // Fonction qui permettra aux classes filles d'exécuter la fonction de recuitStochastique.

Class RecuitStochastique implements Stochastique

- toutesSolutionsEgale(vector<int>, int) : bool //Verifie si toutes les solutions sont égales.
- executer(vector<int>) : vector<int> // Fonction qui exécutera la fonction de recuitStochastique de la classe StochastiqueAbs.

Class SAA implements Stochastique

- echantillonsN : vector<vector<Scenario>>// Les différents échantillons contenants les divers types de scénario.
- echantillonN2 : vector<Scenario> // L'échantillon de référence plus large.
- VmMoyenne : double // valeur moyenne de la fonction objectif.
- meilleurSolution : vector<int> // La solution Xm avec la meilleur Vm.
- executer() : void // Exécution de l'algorithme SAA**.**
- getEchantillonsN() : vector<vector<Scenario>> // Renvoie les différents échantillons.
- getEchantillonN2() : vector<Scenario> // Renvoie l'échantillon de référence plus large.
- getVmMoyenne() : double // Renvoie la valeur moyenne de la fonction objectif.

- getMeilleurSolution() : vector<int> // Renvoie la solution Xm.
- setEchantillonsN(vector<vector<Scenario>>) // Modifie les différents échantillons.
- setEchantillonN2(vector<Scenario>) // Modifie le scénario de référence.
- setVmMoyenne(double) // Modifie la valeur moyenne de la fonction objectif.
- setMeilleurSolution(vector<int>) // Modifie la solution Xm.

B - Partie Vue (et contrôleur)

Class MainWindow extends QMainWindow

- cThread : ComputationThread.

Figurerons également l'ensemble des widgets et layout à utiliser pour implémenter correctement l'interface demandée. Ces widgets et layouts sont fournis par le framework Qt.

- builtUserWindow() : void // Construit l'interface avec la fenêtre utilisateur pour le logiciel.
- changeAlgo(int algo) : void // Modifie l'attribut de classe "algo" de l'objet cThread. Cette méthode sera appelée par un slot, permettant de connecter un widget et une méthode.
- startComputation() : void // Appel la méthode run() de l'attribut cThread.
- quitApp() : void // Quitte le programme en demandant confirmation via une nouvelle fenêtre.

Remarque : l'ensemble des slots permettant de connecter les widgets à des méthodes ne figure pas ici.

Class ComputationThread extends QThread

Permet de garder l'interface réactive lors de longs calculs, susceptibles de bloquer l'interface.

- nbScenario : int // indique le nombre de scénario.
- nbEchantillon : int // indique le nombre d'échantillon.
- solutionInitiale : vector<int> //Indique la solution initiale.
- solutionFinale : vector<int> //Indique la solution finale.
- recuit : Recuit * //Indique le recuit

- changerNbScenario(int nbScenario) : void //Change le nombre de scénario.
- changerNbEchantillon(int nbEchantillon) : void //Change le nombre d'échantillons
- setRecuit (Recuit *) : void // Modifie le recuit
- setSolutionInitiale(vector<int>) : void // Modifie la solution initiale
- setSolutionFinale(vector<int>) : void // Modifie la solution finale.
- getSolutionFinale() : vector<int> // Recupere la solution finale.
- runningAlgo : void // Fait tourner l'algorithme
- run() // Appel l'algorithme associé à l'attribut algo.

C – Explications sur la conception

Lors de la conception du diagramme de classe nous avons fait certains choix qui nous permettent d'avoir un projet modulable. Nous allons donc vous expliquer nos choix.

Tout d'abord nous avons une classe abstraite Probleme qui nous permettra d'avoir des classes filles tel que notre VLS pour notre cas mais aussi si l'on souhaite on peut rajouter des classes tel que le voyageur de commerce. Dans la classe Probleme nous aurons la liste des solutions, la valeur de la fonction objective ainsi qu'un Algorithme.

Nous aurons des fonctions abstraites pour vérifier les contraintes et pour récupérer la valeur de la fonction objective. Cela nous permet d'avoir des contraintes différentes et une fonction de calcul de la fonction objective différente selon les problèmes. Pour notre problème de VLS notre fonction objective et notre vérification des contraintes seront définis par notre modèle mathématique. Si nous avons d'autre problème tel que le voyageur de commerce nous pouvons donc changer le modèle mathématique depuis la classe VoyageurDeCommerce.

Nous avons aussi décidé de faire une classe abstraite Algo qui permettra d'avoir différents algorithmes par exemple pour notre cas nous aurons comme fille une classe Recuit et une classe Cplex mais on pourrait aussi rajouter d'autre algorithme de résolution de problème. Nous ne pouvons pas faire une interface car nous allons avoir un attribut Probleme qui ne sera pas statique.

Nous avons choisi de faire la classe Recuit comme classe abstraite car nous allons avoir comme classe fille le recuit déterministe et le recuit stochastique. Nous l'avons faite abstraite car nous allons avoir une fonction executer() qui sera implémenté différemment selon les classes filles. Par exemple pour le recuit déterministe il n'appellera qu'une seule fois la méthode recuit de notre classe abstraite recuit alors qu'un recuit stochastique utilisera plusieurs fois la méthode recuit dû au nombre de scénario.

Nous avons eu la même réflexion pour le recuit stochastique et SAA c'est pour cela que nous avons une classe stochastique qui est abstraite. En effet pour le SAA nous allons utiliser plusieurs fois le recuit stochastique.

III - Diagramme de classe