Projet d'Optimisation Stochastique

Problème du voyageur de commerce

-

Document organique

Enseignant : Abdel LISSER

Groupe : 3

Membres du groupe : Adrien LAVILLONNIERE
Corentin MANSCOUR
Hien Minh NGUYEN

Rédigé le : 04/11/2018

Nombre de pages : 23

Table des matières

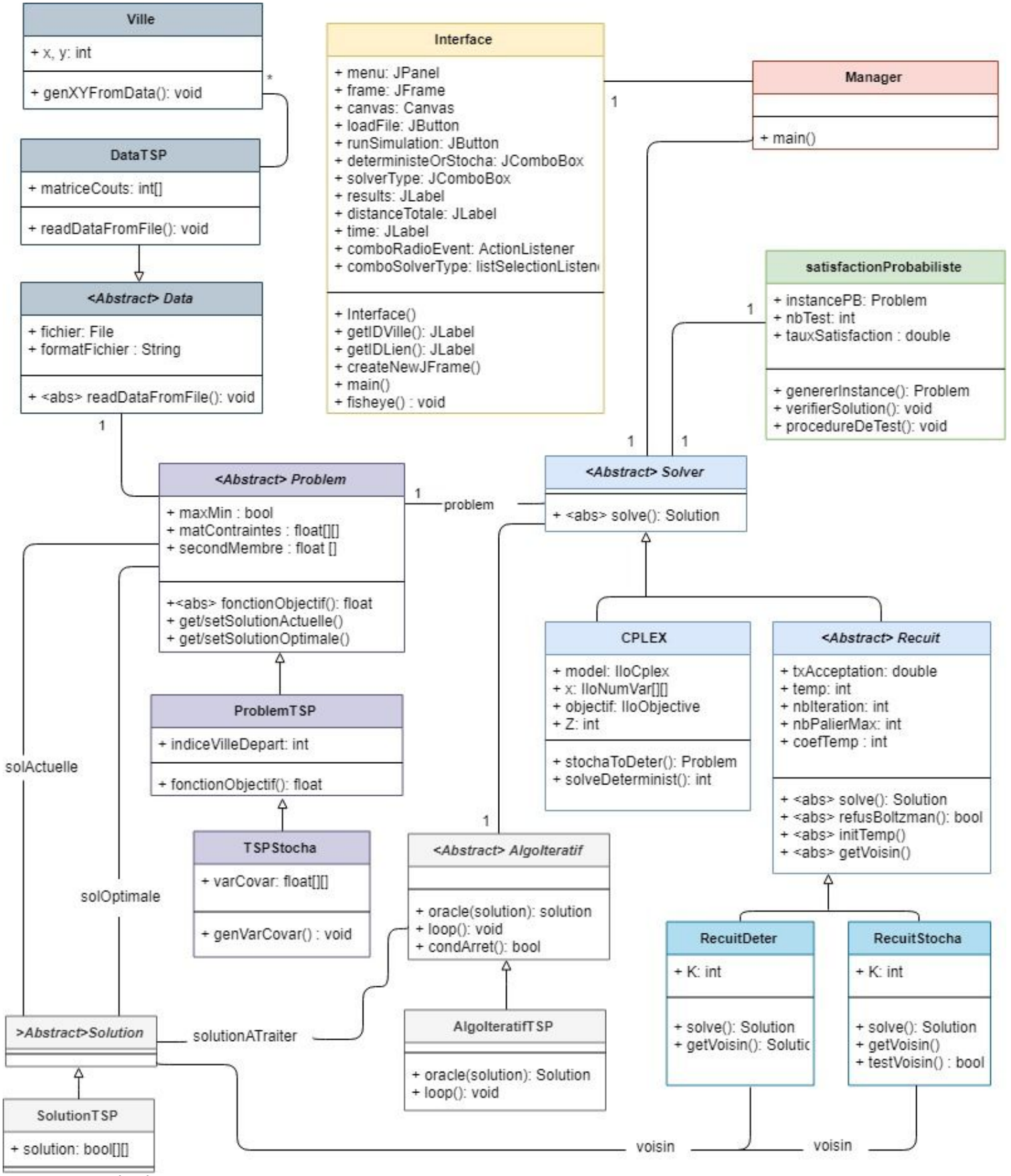
I. Introduction

Ce projet vise à proposer une solution au célèbre problème du voyageur de commerce : le voyageur doit passer par un ensemble de villes, avec la contrainte de ne passer qu'une seule fois par ville, de minimiser son temps de trajet et de retourner à la ville de départ.

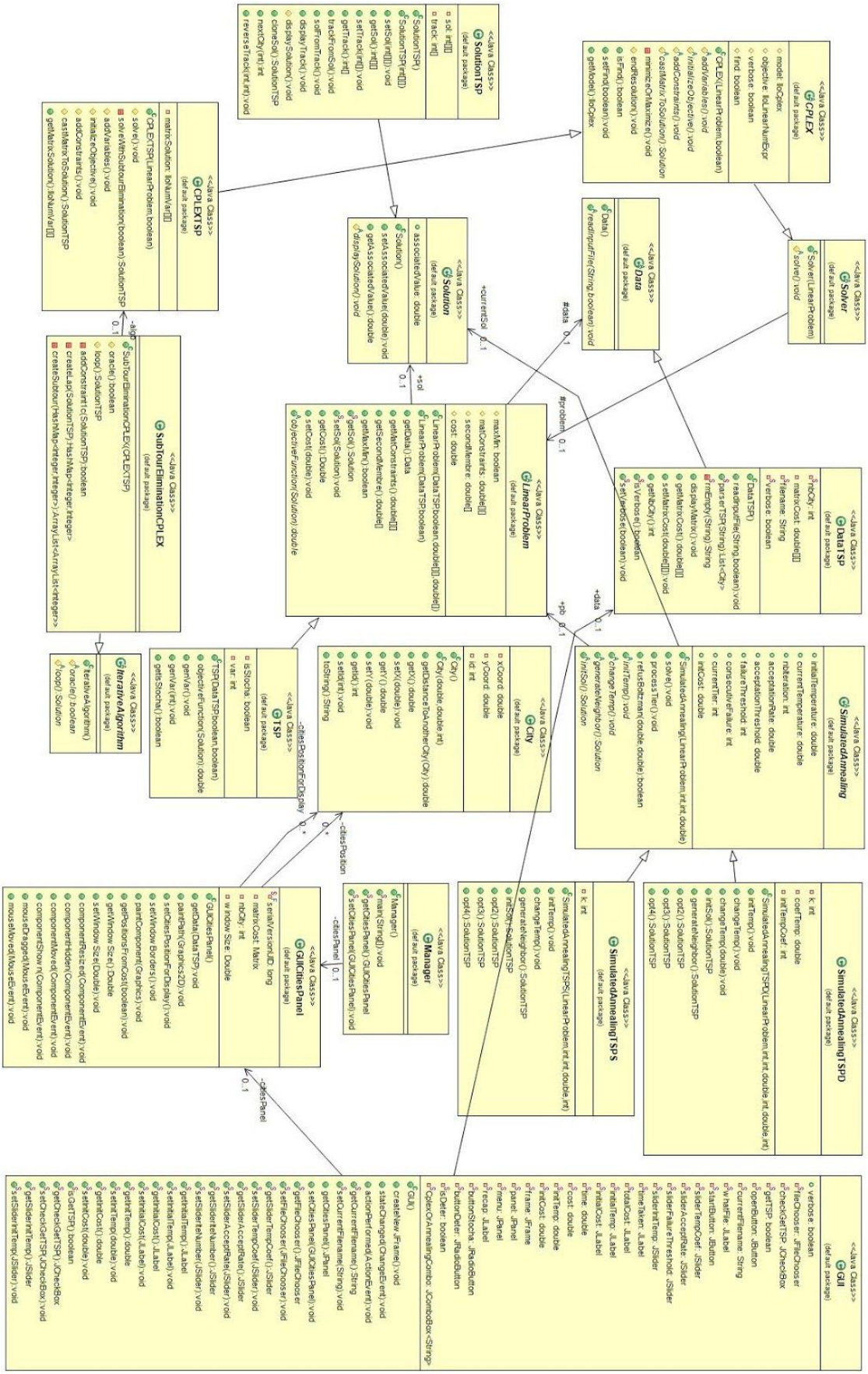
Afin de résoudre ce problème, nous développerons une application en Java, prenant en entrée des fichiers contenant un ensemble de villes et leurs données. L'application sera capable d'utiliser deux types de résolution différents : l'algorithme du recuit simulé ou bien CPLEX. L'application sera également en mesure de résoudre les versions stochastiques et déterministes du problème.

L'objectif de ce document organique sera de montrer la structure de l'application finale, via un diagramme de classe. Ce diagramme sera dans un deuxième temps détaillé sous forme de liste, avec des explications pour chaque classe, variable et méthode.

II. 1. Diagramme de classe originel



II. 2. Diagramme de classe mis à jour (généré à partir du code)

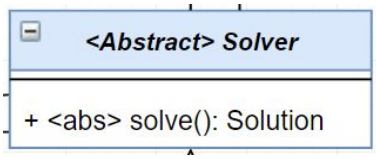


III. Explications du diagramme de classe

1) Classes génériques

1.1 Solveur générique

1.1.1 Solveur



Description : la classe abstraite du solveur générique représente les éléments communs entre les deux autres solveurs du programme : le recuit simulé et CPLEX. Grâce à la généricité, ces méthodes communes pourront facilement être implémentées dans d'autres solveurs.

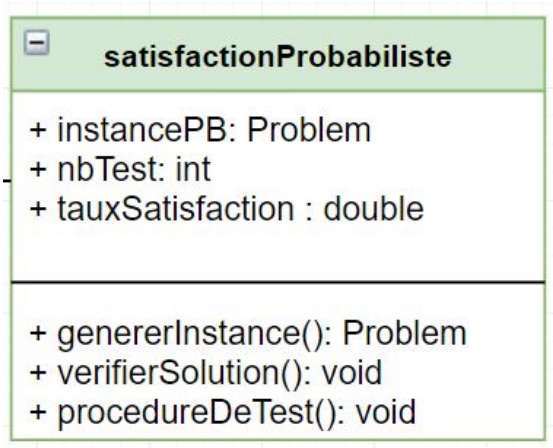
Variables :

- Problem **problem** : la classe abstraite Solveur contient une instance de la classe Problem contenant les données du problème ainsi que les solutions actuelles et optimales du programme

Méthodes :

- abstract Solution **solve()** : cette méthode lance le calcul de la solution du problème et retourne une solution à celui-ci.

1.1.2 satisfactionProbabiliste



Description : Une fois une solution à un problème stochastique proposée, cette classe va pouvoir créer des instances du problème pour tester empiriquement si la solution respecte sa contrainte en probabilité.

Variables de classe :

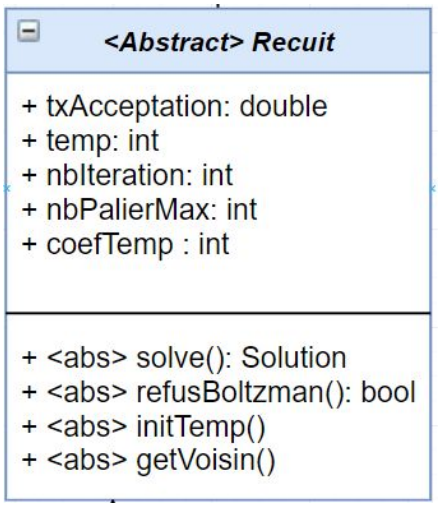
- **instancePB :** contient une instance du problème stochastique
- **nbTests :** détermine le nombre d'instances qui ont été testées
- **tauxSatisfaction :** contient le résultat de l'opération "testReussis/nbTests"

Méthodes de classe :

- **genererInstance() :** joue toutes les variables aléatoires du problème stochastique, et sauvegarde cette instance du problème dans instance PB.
- **verifierSolution() :** vérifie si le coût obtenu sur instancePB est en dessous ou en dessus du seuil de la contrainte. Incrémente nbTests, et recalcule tauxSatisfaction en fonction du résultat.
- **procedureTest(int x) :** appelle genererInstance() puis verifierSolution() un nombre X de fois.

1.2 Algorithme du Recuit simulé

1.2.1 SimulatedAnnealing (abstrait)



Description : cette classe abstraite décrit l'algorithme du recuit simulé, utilisé pour résoudre le problème. Elle contient les éléments basiques utilisés pour ajuster les paramètres généraux de l'algorithme. Elle représente les éléments communs entre les classes recuitStochastique et recuitDeterministe. Elle hérite de la classe abstraite Solveur.

Variables de classe :

- double **txAcceptation** : variable représentant le taux d'acceptation du recuit
- Problem **solutionActuelle :** solution trouvé à l'itération actuelle
- Problem **solutionOptimale** : meilleur solution trouvé à l'itération actuelle
- int **temp** : température du recuit actuelle
- int **nbIteration** : nombre d'itération par pallier
- int **nbPalierMax** : nombre de palier maximum à faire avant de considérer solutionOptimale comme la solution optimale du problème
- int **coefTemp** : le coefficient par lequel on multiplie la température après une itération

Méthodes de classe :

- **Recuit()** : constructeur de la classe, servant à créer la classe et à initialiser ses valeurs

- bool **kirkpatrick()** : méthode permettant de déterminer automatiquement si une température initiale est acceptable ou non**.** Le schéma de Kirkpatrick est décrite en détail dans le rapport technique
- bool **refusBoltzman()** : méthode calculant l'acceptation d'une solution ou non selon la distribution de Gibbs-Boltzman, tel que décrit dans le rapport technique
- abstract **initTemp()** : initialise la température

- abstract **getVoisin()** : retourne le voisin d'une solution actuelle

- float **tauxAcceptation()** : calcule le taux d'acceptation actuel en fonction du nombre de mouvements pour ce palier et du nombre fixe de mouvements par palier prédéfini

- void **solve()** : cette méthode lance le calcul de la solution problème. L'algorithme du recuit, décrit dans le document technique, se déroule comme suit.

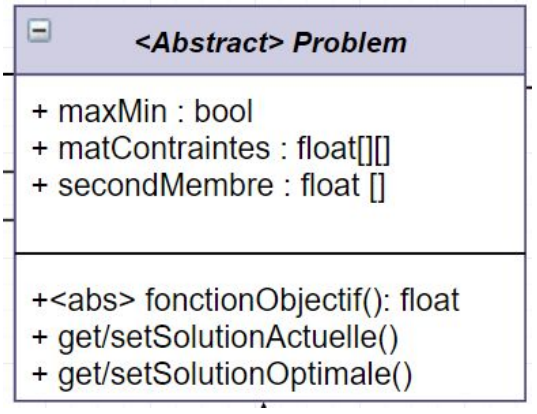
  Une solution initiale est calculée à l'aide de l'algorithme du plus proche voisin. La solution initiale est stockée dans la variable solutionOptimale. On applique ensuite la fonction objectif du problème sur cette solution afin d'obtenir son coût.

  On commence ensuite la boucle principale du programme telle que décrite dans le document technique. On calcule une solution grâce à l'algorithme du K-opt, et on compare le coût de cette solution avec la solution optimale. Si la solution actuelle est meilleure, on la garde et on l'écrit dans solutionOptimale. Sinon, on la garde ou non de manière aléatoire en fonction de la méthode *refusBoltzman()*.

  A la fin d'un palier, l'algorithme est multiplié par un coefficient arbitraire, coefTemp. L'algorithme s'arrête lorsque le taux d'acceptation sera trop faible et qu'un nombre défini d'itérations se fassent sans que rien ne change.

2.2 Programme linéaire

2.2.1 LinearProblem



Description : classe représentant un problème linéaire, sa fonction objectif minimisante ou maximisante, ses contraintes et les set/get associés.

Variables de classe :

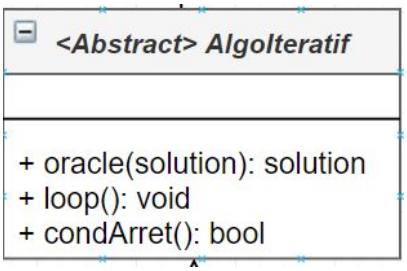
- bool **maxMin** : true si on cherche à maximiser la solution, false sinon
- double[][] **contraintes** : contraintes du problème linéaire
- double[] **secondMembre** :
- Data **data** : instance de la classe data, regroupant les informations lues à partir d'un fichier
- Solution **solutionActuelle** : variable représentant la solution actuelle du problème.
- Solution **solutionOptimale** : variable représentant la solution optimale du problème.

Méthodes de classe :

- abstract float **fonctionObjectif()** : fonction objective abstraite du problème linéaire retournant le coût d'une solution
- méthodes pour fabriquer les contraintes : matrice, second membre, inégalités
- méthodes d'affichage, getters et setters nécessaires

2.3 Algorithme itératif

2.3.1 AlgoIteratif (abstract)



Description : cette classe permet de générer et d'appliquer des contraintes à une solution du programme linéaire. Elle consiste en une boucle qui appelle le problème linéaire (notre classe Problem) afin de récupérer une solution x. Cette solution x sera ensuite transmise à une méthode appelée *oracle*, qui génèrera les contraintes et les appliquera à la solution. La boucle continue jusqu'à ce que la contrainte soit satisfaite.

Variables de classe :

- Solution **solutionATraiter** : représente une solution optimale sans la contrainte apportée par la méthode oracle.

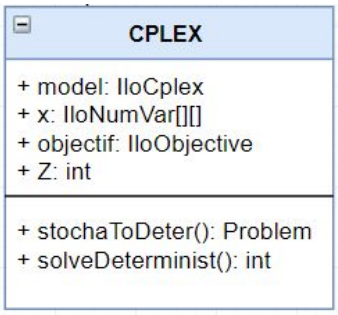
Méthodes de classe :

- void **loop()**: méthode permettant de répéter la génération de contrainte jusqu'à ce que celle-ci n'existe plus
- Solution **oracle()** : la méthode oracle vérifie si la solution à traiter contient des sous-tours, puis génère les contraintes permettant de supprimer ceux-ci. Elle renvoie en sortie une solution avec la contrainte de sous-tour appliquée.
- bool **condArret()** : permet de continuer la boucle tant que la condition d'arrêt n'est pas vérifiée
- méthodes d'affichage, getters et setters nécessaires

2.4 <Abstract> Solution

Description : classe abstraite contenant une solution au problème

2.5 CPLEX



Description : classe générique du solveur CPLEX.

Variables de classe :

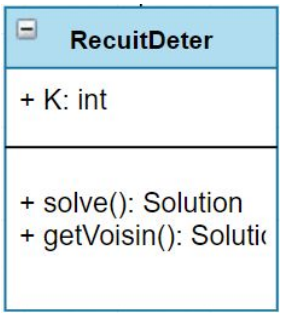
- IloCplex **model** : cette variable contient le problème à résoudre par CPLEX. Elle regroupe la fonction objectif du problème ainsi que ses contraintes
- IloNumVar **x** : contient les variables du problème linéaire
- IloObjective **objectif**: contient la fonction objective du problème linéaire
- int **Z** : constante correspondant à la valeur de la solution optimale du problème déterministe majorée de 20 à 30%.

Méthodes de classe :

- void **addVariables**() : ajoute les variables du problème linéaire au modèle
- void **addConstraints**() : ajoute les contraintes du problème linéaire au modèle
- void **initializeObjective**() : initialise la fonction objective du problème linéaire et l'ajoute au modèle
- Solution **castMatrixToSolution**() : transforme la variable contenant la solution de l'API CPLEX en une instance de Solution
- void **endResolution**() : libere l'espace memoire pris par la variable model
- getters, solveurs

2) Classes du Problème du Voyageur de Commerce

2.2. RecuitDeterministe (SimulatedAnnealingTSPD)



Description : cette classe hérite de la classe abstraite recuit simulé permettant de résoudre un problème linéaire déterministe. Elle implémente les méthodes de la classe recuit simulé.

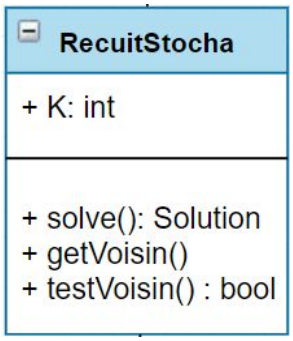
Variables de classe :

- int **K** : cet entier représente le nombre K de l'algorithme du K-Opt, soit le nombre d'arêtes à permuter à chaque itération.

Méthodes de classe :

- Solution **solve()**: calcule la solution optimale d'un problème linéaire déterministe en utilisant le recuit simulé
- Solution **getVoisin()** : retourne le voisin de la solution actuelle calculée par l'algorithme du recuit. Cette méthode se base sur l'algorithme du K-opt, tel que décrit dans le document technique, qui consiste à permuter k arêtes du graphe.
- méthodes d'affichage, getters et setters nécessaires

2.3 RecuitStochastique (SimulatedAnnealingTSPS)



Description : cette classe hérite de la classe abstraite recuit simulé permettant de résoudre un problème linéaire stochastique. Elle implémente les méthodes de la classe recuit simulé.

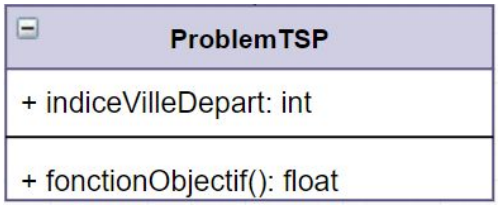
Variables de classe :

- **K** (int) : cet entier représente le nombre K de l'algorithme du K-Opt, soit le nombre d'arêtes à permuter à chaque itération.

Méthodes de classe :

- Solution **solve()**: calcule la solution optimale d'un problème linéaire stochastique en utilisant le recuit simulé
- Solution **getVoisin()** : retourne le voisin de la solution actuelle calculée par l'algorithme du recuit. Cette méthode se base sur l'algorithme du K-opt, tel que décrit dans le document technique, qui consiste à permuter k arêtes du graphe
- Solution **testVoisin()** : vérifie que le voisin généré par getVoisin() respecte la contrainte de probabilité.
- méthodes d'affichage, getters et setters nécessaires

2.4 ProblemTSP



Description : classe représentant le problème du voyageur de commerce. Elle sera remplie au fur et à mesure

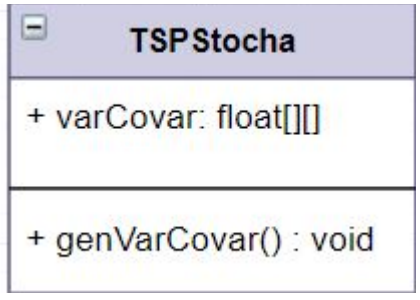
Variables de classe :

- int **indiceVilleDepart** : indique la ville de départ, permettant à l'algorithme de se placer au bon endroit dans le tableau des solutions et d'ainsi trouver sa ville voisine

Méthode de classe :

- float **fonctionObjectif()** : cette méthode représente la fonction objectif du programme et retourne le coût d'un chemin.
- **get/setSolutionActuelle()** : méthodes permettant de sauvegarder ou de lire la solution actuelle calculée par l'algorithme
- **get/setSolutionOptimale()** : méthodes permettant de sauvegarder ou de lire la solution optimale

2.5 TSPStocha



Description : classe contenant les données du problème du voyageur de commerce stochastique. Elle hérite de la classe ProblemTSP qui contient les données communes au problème du TSP stochastique et déterministe.

Variables de classe :

- float[][] **varCovar** : c'est la matrice des variance-covariance

Méthodes de classe :

- **genVarCovar()** : va générer aléatoirement la dispersion de chaque variable aléatoire autour des valeurs récupérée dans Data afin d'avoir un jeu de données stochastique. Puis à partir de ces dispersion, la fonction écrira la matrice des variances-covariances.

2.6 CPLEXTSP

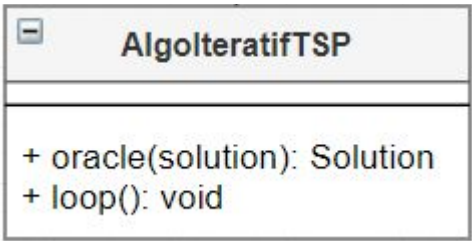
Description : classe implémentant la résolution du problème via le solveur CPLEX.
Variables de classe :

- IloNumVar[][] **matrixSolution**: matrice représentant la solution d'un probleme du voyageur de commerce. Sa valeur est de true si l'arc représenté par matrixSolution[villeI][villeJ] est retenu dans le circuit

Méthodes de classe :

- void **solve**() : résout le problème du voyageur de commerce, met à jour le coût et la solution du problème linéaire représenté par la variable problem
- SolutionTSP **solveWithSubtourElimination**() : résout le problème du voyageur de commerce en ajoutant itérativement la contrainte sur les sous-tours
- void **addVariables**() : ajoute une matrice de boolean représentant la solution du TSP au modèle
- void **initializeObjective**() : initialise la fonction objective du probleme du voyageur de commerce
- void **addConstraints**() : ajoute la contrainte "un unique arc entrant par ville", ajoute la contrainte "un unique arc sortant par ville"
- SolutionTSP **castMatrixToSolution**() : retourne une instance de SolutionTSP représentant la solution du probleme

2.7 AlgoIteratifTSP



Description : classe permettant de supprimer les sous-tours d'une solution optimale fournit par CPLEX

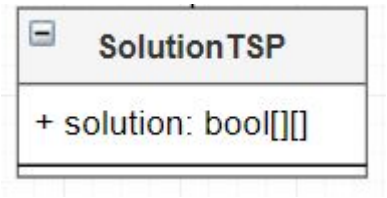
Variables de classe :

- Solution **solutionATraiter** : solution optimale d'un problème TSP obtenu après l'application de CPLEX au problème linéaire

Méthodes de classe :

- Solution **oracle(Solution)** : prends solutionATraiter en paramètre, génère une contrainte de sous-tours comme expliqué dans le document technique et l'applique. Renvoie une nouvelle solution optimale
- **loop()** : applique la méthode oracle à la variable solutionATraiter (étape 1), vérifie que la nouvelle solution optimale ne contient pas de sous-tours avec la méthode condArret() (étape 2). Si la nouvelle solution possède des sous-tours, la méthode stock cette solution dans la variable solutionATraiter et recommence depuis l'étape 1.

2.8 SolutionTSP



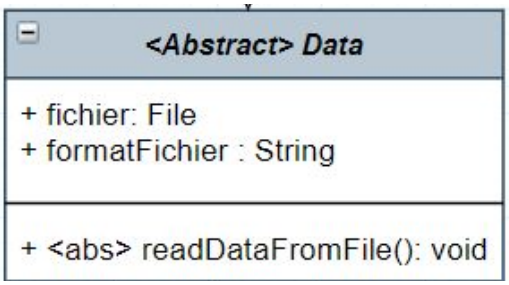
Description : cette classe contient une solution au problème du TSP sous forme d'un tableau à double entrée

Variable de classe :

- bool[][] **solution** : matrice représentant les chemins entre les villes du problème TSP. True signifie que le chemin fait partie de la solution, false que le chemin n'est pas pris en compte.

3) Classes Data

3.1 Data



Description : classe abstraite représente les données d'un problème linéaire. Elle contient également les méthodes nécessaires pour lire les fichiers contenant les données du problème.

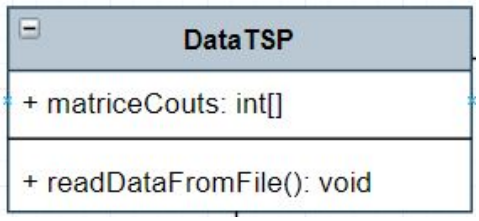
Variables de classe :

- File **fichier** : fichier contenant les données d'un problème linéaire
- String **typeFichier** : indique le type du fichier lu

Méthodes de classe :

- <abstract> void **readDataFromFile():** cette méthode permettra de lire les données d'un fichier

3.2 DataTSP



Description : classe représentant les données du problème du voyageur de commerce

Variables de classe :

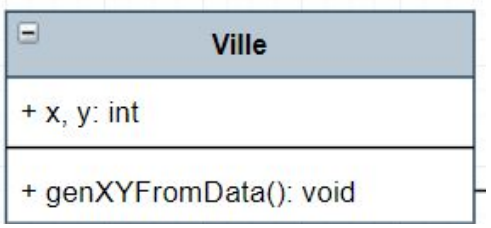
- int[] **matriceCouts** : variable stockant les coûts entre les villes, lus à partir d'un fichier

Méthodes de classe :

- void **readDataFromFile()** : méthode permettant de lire les données d'un fichier. Si le fichier regroupe des coordonnées de noeuds, alors on appelle la méthode readNodesFromFile. Sinon si le fichier se présente sous forme de matrice, alors on appelle la méthode readMatrixFromFile.

- **readNodesFromFile()** : méthode permettant de lire un fichier comportant des coordonnées de noeuds et d'écrire ces données dans la classe DataTSP
- **readMatrixFromFile()** : méthode permettant de lire un fichier comportant des coordonnées sous forme de matrice et d'écrire ces données dans la classe DataTSP

3.3 Ville



Description : cette classe contient les coordonnées d'une ville. Les villes sont stockées dans un tableau, contenu lui-même dans la classe DataTSP.

Variables de classe :

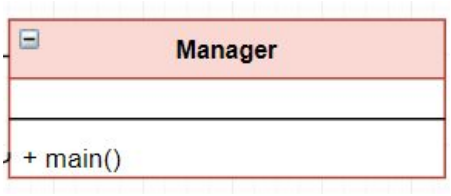
- int **x** : coordonnée x d'une ville
- int **y** : coordonnée y d'une ville

Méthodes de classe :

- void **getPositionFromCost**() : permet de générer les coordonnées des villes à partir de la matrice des coûts, grâce à une formule mathématique.
- void **setCitiesPositionForDisplay**() : permet de générer les coordonnées des villes afin de les afficher de manière correcte, dans les limites de la fenêtre du programme

4) Classes Manager

4.1 Manager



Description : cette classe gère l'intégralité de la communication entre les autres classes. Elle permet de faire le lien entre la classe Interface qui donnera en sortie les choix de l'utilisateur et la représentation graphique du problème, la classe Data qui lira les données du fichier donné par l'utilisateur et la classe Solveur.

Variables de classe :

- Interface **interface** : instance de la classe Interface
- Solver **solver** : instance de la classe Solver

Méthodes de classe :

- **main()** : le main du programme initialise et instancie toutes les classes nécessaire à son fonctionnement. Il lance la classe Interface et lui fournit une instance de la classe Data

5) Classes de l'interface utilisateur

5.1 Interface

Interface : classe de l'interface du programme utilisant la librairie Swing

- JPanel **menu** : variable contenant les éléments du menu de gauche sur la fenêtre de l'application
- JPanel **mainPanel** : panneau principal du programme
- JFrame **frame** : frame principale du programme. Elle englobe tous les autres JPanels
- Canvas **canvas** : permet à l'application de dessiner les villes et le trajet du voyageur
- JButton **loadFile** : bouton permettant de charger un fichier qui pourra être lu par l'application
- JButton **runSimulation** : bouton permettant de lancer la simulation
- JRadioButton **deterministeOrStocha** : deux boutons radio permettant de choisir entre une résolution déterministe ou stochastique du problème
- JComboBox **solverType** : une liste déroulante permettant de choisir entre une résolution via les solveurs disponibles
- JLabel **results** : le label "Résultats"

- JLabel **distanceTotale** : un label permettant de voir la distance totale parcourue par le voyageur
- JLabel **time** : un label permettant de voir le temps de calcul requis pour résoudre ce problème
- ActionListener **comboRadioEvent** : listener permettant de changer le comportement du programme en fonction du choix entre déterministe ou stochastique
- listSelectionListener **comboSolverType** : listener permettant de changer le comportement du programme en fonction du choix entre CPLEX ou l'algorithme du recuit

- **Interface()** : méthode lançant l'interface du programme
- JLabel **getIDVille()** : méthode permettant de récupérer l'ID d'une ville
- JLabel **getIDLien()** : méthode permettant de récupérer l'ID d'un lien entre deux villes
- **createNewJFrame() :** crée la fenêtre principale du programme