Rapport de projet informatique, programmation orientée objet.

Méthode d'optimisation : *Simulated Annealing* ou le "recuit simulé".

Thibault Barnouin

5 mars 2020

1 Le projet

Un algorithme d'optimisation cherche à déterminer un jeu de paramètres qui minimise (ou maximise) une fonction ou un modèle donné. Il existe plusieurs méthodes d'optimisation. Une méthode de type *force brute* teste toute les solutions possible pour en déterminer la meilleure. Ce type de méthode n'est cependant plus utilisable lorsque l'espace des solutions devient trop grand, voire même continu ou infini. Une méthode heuristique cherche une solution réalisable, pas nécessairement la plus optimale, pour un problème d'optimisation difficile.

La méthode de "recuit simulé" est une méthode heuristique qui tire son inspiration de pratiques issues de la thermodynamique : un système qui refroidit lentement peut atteindre un niveau d'énergie plus bas, et donc être plus stable. Cette méthode s'appuie ainsi sur un refroidissement lent, autorisant des temps suffisamment longs pour que les solutions se redistribuent au fur et à mesure que leur énergie diminue. Elle permet d'explorer un voisinage d'une solution sans pour autant se bloquer dans des zones de minimum (ou maximum) local de la fonction ou du modèle étudié et ainsi atteindre un extremum global.

Le but du projet est de mettre au profit la programmation orientée objet en *C++* pour coder un algorithme d'optimisation de type "recuit simulé" et l'appliquer à l'approximation de données expérimentales par des modèles et à la résolution du problème du Voyageur de Commerce.

2 Le code

2.1 Pertinence de la programmation orientée objet pour la réalisation de ce projet

Ce projet implique l'application d'un algorithme général, l'algorithme de "recuit simulé". Chaque modèle à optimiser par l'algorithme général doit répondre aux questions suivantes : Qu'est-ce qu'une *solution* ? Qu'est-ce qui est considéré comme une *solution voisine* ? Quel est le *coût* d'une solution ?

L'utilisation de classes avec des méthodes spécifiques répondant à chacune de ces questions facilite l'implémentation du programme.

2.2 Les classes

Dans un premier temps, je crée une classe Data pour la gestion et l'export des données étudiées et générées par des modèles. Une classe abstraite CostFunction permet la définition par héritage de fonctions définissant les coûts associés au différents modèles (on peut penser au moindre carré χ^2 pour l'approximation de fonctions, ou à la distance pour le problème du voyageur de commerce). Les modèles à optimiser sont définis en héritage d'une classe abstraite Model qui a comme attributs des objets de type Data pour les données expérimentales et du modèle (composition) et un pointeur vers une CostFunction associée au modèle (aggrégation). Chaque modèle hérite de ces attributs et y ajoute un jeu de paramètres. Il définit aussi les méthodes donnant le *coût* de la solution et la *solution voisine*. On peut ainsi définir les modèles pour une approximation linéaire, polynomiale ou pour trouver un chemin dans le problème du Voyageur de Commerce. Enfin, je crée une classe abstraite OptimizationMethod qui prend en attribut un pointeur vers un Model à optimiser (aggrégation) et définit les méthodes d'exécution et d'export de l'algorithme d'optimisation. Les différentes méthodes d'optimisation sont définies par héritage et ont chacune en attributs leur différents paramètres.

Cette architecture de classes est résumée dans le diagramme de classes (voir Figure 1).

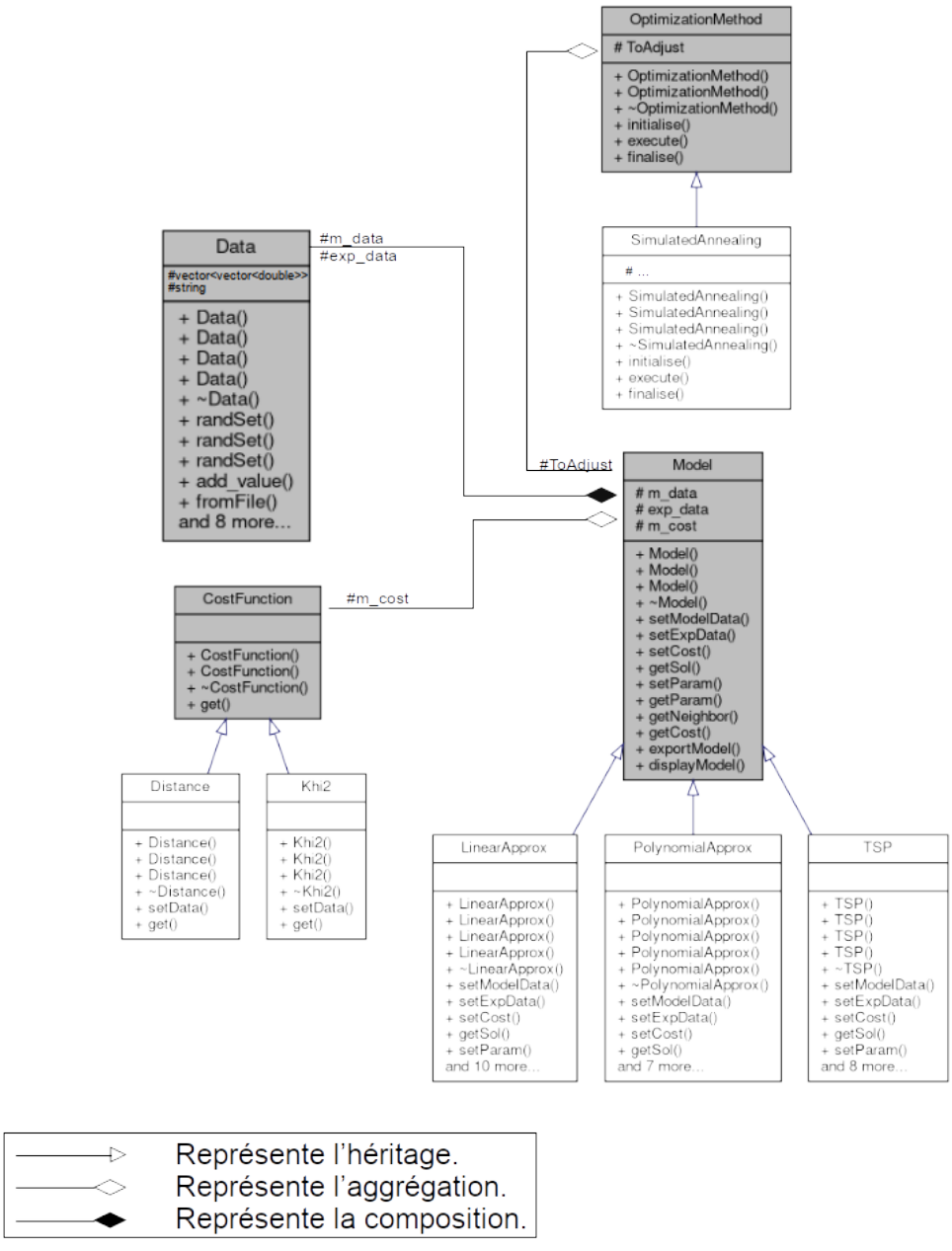

FIGURE 1 – Diagramme de classes.

3 Applications de l'algorithme d'optimisation

3.1 Approximation de données avec un modèle

Afin de tester le fonctionnement des classes et méthodes et pour une première utilisation de l'algorithme d'optimisation, je génère des données arbitraires suivant un modèle linéaire ou polynomial auquel j'ajoute un bruit aléatoire et j'utilise l'algorithme d'optimisation pour retrouver les paramètres du modèle qui a généré ces données. J'obtiens ainsi les sorties de la Figure 2.

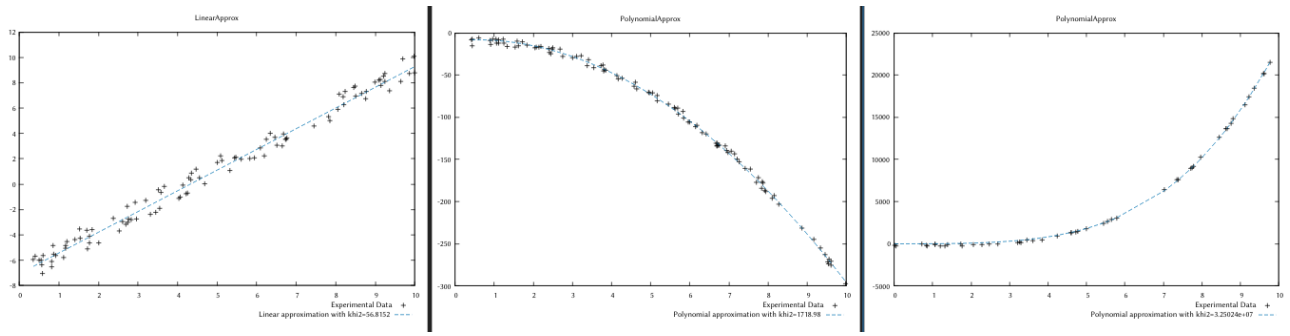

FIGURE 2 – Approximation de fonctions à l'aide de l'algorithme de recuit simulé.
A gauche une fonction linéaire, au milieu une fonction polynomiale de degré 2, à droite une fonction polynomiale de degré 4, toutes avec du bruit aléatoire.

3.2 Résolution du problème du Voyageur de Commerce

Pour résoudre le problème du Voyageur de Commerce, je génère un ensemble de N villes distribuées dans un carré de côté 1. La première *solution* est un chemin passant par les villes dans leur ordre de génération. Je définis ensuite de plusieurs façons la *solution voisine* et je cherche celle qui est la plus efficace pour répondre au problème à l'aide de l'algorithme de "recuit simulé". On peut voir des résultats générés avec la même graine mais obtenus avec des définitions de voisinage différentes en Figure 3.

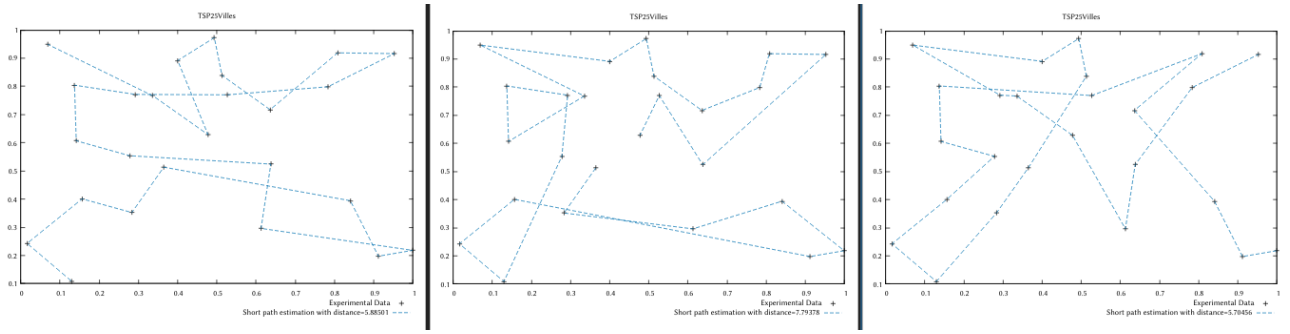

FIGURE 3 – Tentative de résolution du problème du Voyageur de Commerce en modifiant la définition de *solution voisine*.
A gauche : deux villes sont échangées dans l'ordre initial. Au milieu : une ville est placée à la fin du chemin. A droite : un sous-ensemble du chemin est mélangé.

L'obtention d'une solution est très sensible aux paramètres entrés dans la méthode de "recuit simulé". Je cherche donc à affiner les paramètres pour les adapter au problème étudié et ainsi obtenir une "bonne solution" en un temps raisonnable. Je teste ces paramètres sur une graine de génération fixe et j'obtiens les résultats en Figure 4.

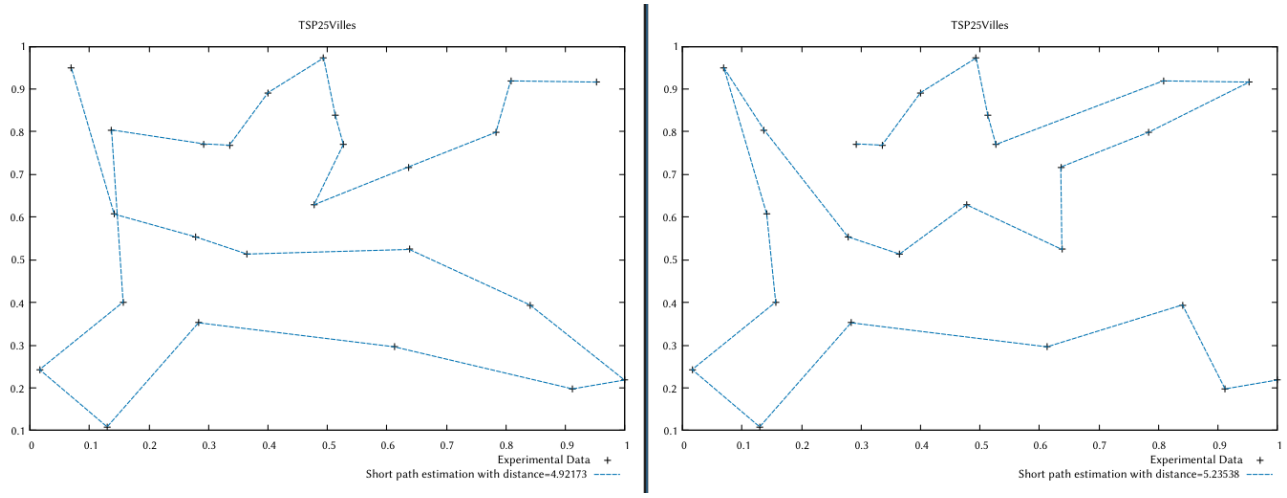

FIGURE 4 – Tentative de résolution du problème du Voyageur de Commerce en modifiant les paramètres de l'algorithme d'optimisation.

4 Bilan du projet

4.1 Les résultats

Le programme ainsi codé et compilé permet d'obtenir rapidement de bons résultats pour l'approximation de données expérimentales avec des fonctions linéaires ou polynomiales. Je suis parvenu à trouver de bonnes définitions de voisinage de solution et de bons paramètres de l'algorithme de "recuit simulé" pour résoudre le problème du Voyageur de Commerce pour une trentaine de villes. Il m'est plus difficile cependant de trouver les bons paramètres pour un grand nombre de villes et les solutions sont obtenues plus lentement et ne sont pas les plus optimales que je puisse espérer.

4.2 Pour aller plus loin

Afin d'améliorer la résolution de problème par l'optimisation il serait intéressant de trouver de nouvelles définitions de *solution voisine* ainsi que les paramètres d'optimisation associés à chaque modèle et à son voisinage.

J'ai pensé les classes de façons à ce que le code soit facilement modulable. En effet, il est simple d'ajouter des méthodes d'optimisation, modèles ou fonctions de coût en héritage des classes abstraites qui s'articulent toujours de la même façon. Je pourrais donc reprendre ce code dans le futur pour l'améliorer et le diversifier.