Partitionnement de graphe

Rapport de projet

Kévin Barreau Guillaume Marques

18 mars 2015

Résumé

L'objectif du projet est d'implémenter différentes méthodes de partitionnement de graphe, vu précédemment en cours, afin des les comparer. Il s'agit de méthode exhaustive, avec l'énumération, d'algorithme glouton, avec la descente de gradient, et de métaheuristiques, avec le recuit simulé, la recherche Tabou et l'algorithme génétique. Le langage de programmation pour atteindre ce but est libre (Javascript a été choisi pour ce projet). Les résultats sont présentés de deux façons : une comparaison des résultats obtenus par les différentes méthodes sur les mêmes instances de graphe, ainsi qu'une visualisation en temps réel de la meilleure solution trouvée par les méthodes.

Sommaire

1 Présentation du projet

1.1 Description du problème

À partir d'un graphe non orienté, il faut partitionner le graphe en K classes au moyen de plusieurs métaheuristiques, de telle sorte que la somme des poids entre sommets n'appartenant pas à la même classe soit minimale. De plus il faut s'assurer que les sommets du graphe soient répartis de manière (à peu prés) équitable. C'est un problème NP-complet.

Nous avons choisi d'utiliser comme notion d'équité une représentation par un seuil de tolérance dans la différence entre la taille du plus grand cluster et la taille du plus petit cluster. Par exemple, pour un graphe de 10 sommets, un partitionnement en 3 classes avec une tolérance de 2 autorise une solution de la forme $\langle \mathbf{2}, \mathbf{4}, \mathbf{4} \rangle$ ($4 - 2 = 2$, inférieur ou égal à la tolérance) mais n'autorise pas une solution de la forme $\langle \mathbf{2}, \mathbf{3}, \mathbf{5} \rangle$ ($5 - 2 = 3$, strictement supérieur à la tolérance).

Le nombre de classe et la tolérance sont paramétrables.

1.2 Méthodes de résolution

Pour résoudre ce problème, nous avons implémenté plusieurs algorithmes.

— Énumération

— Descente de gradient

— Recuit simulé

— Méthode Tabou

— Algorithme génétique

L'énumération est une méthode exhaustive, qui parcours toutes les solutions possibles pour garder le meilleur résultat. La solution finale est la solution optimale du problème.

La descente de gradient est un algorithme glouton, qui, à partir du voisinage d'une solution, se déplace vers le meilleur résultat améliorant. On ne revient donc jamais sur une solution déjà visitée. La solution finale n'est pas forcément la solution optimale du problème.

Le recuit simulé, la méthode Tabou et l'algorithme génétique sont des métaheuristiques, cherchant une solution en essayant de ne pas avoir le plus gros problème de la descente de gradient : rester bloquer dans un optimum local. La solution finale n'est pas forcément la solution optimale du problème.

1.3 Choix de mise en œuvre

Pour implémenter les différentes méthodes de partitionnement de graphe, nous avons choisi d'utiliser le langage de programmation Javascript. Ce dernier apporte de nombreux avantages comme :

— Rapidité d'écriture (langage interprété à typage dynamique)

— Accessibilité (un navigateur web suffit pour l'exécution)

— Visualisation (HTML, CSS, Canvas, WegGL...)

On peut cependant lui reprocher une lenteur relative, comparé à des langages comme C++ ou Java. Dans un problème d'optimisation comme celui du partitionnement de graphe, on cherche toujours les meilleures performances possibles. Cependant, nous avons voulu abordé ce projet comme une introduction aux différentes méthodes et à leur comparaison, et non comme une implémentation la plus optimisée qui soit.

Les structures de données utilisées sont ainsi orientées vers le besoin d'affichage des résultats, non vers le besoin de performance. Les performances de mémoire sont principalement impactées, car plusieurs représentation du graphe sont utilisées (liste de sommets et d'arêtes pour la visualisation, matrice d'adjacence pour les algorithmes).

Ce choix de langage nous permet aussi d'apporter une interface graphique riche à moindre coût. Toutes les méthodes sont ainsi paramétrables par le biais de cette interface, sans avoir besoin de modifier et de recompiler le code.

2 Implémentation des méthodes de partitionnement de graphes

Chaque méthode possède sa propre façon de fonctionner. Il faut donc être capable de réaliser des opérations identiques sur des codages de solutions différents, qui correspondent à la structure de données pour enregistrer une solution. Nous avons utilisé 2 codages : une **liste de partitions** et un **tableau**.

Les méthodes de partitionnement de graphes utilisent aussi des fonctions de voisinage. Le voisinage correspond aux solutions atteignables à partir d'une solution, en faisant un mouvement élémentaire. Nous avons choisi d'implémenter 2 mouvements élémentaires : le **swap** et le **pick'n'drop fallback swap**.

2.1 Codage des solutions

2.1.1 Liste de partitions

La liste de partitions est un tableau de tableau. Le premier tableau correspond aux classes, chaque case représentant une classe pour le partitionnement du graphe. L'index dans ce tableau correspond au numéro de la classe. Chaque case est composée d'un tableau. Ce deuxième tableau possède le numéro des sommets appartenant à la classe. L'index dans ce tableau ne correspond à rien contrairement au précédent tableau.

Prenons comme exemple un graphe de 5 sommets partitionné en 2 classes, avec les sommets 0 et 3 dans la classe 0, et les sommets 1, 2 et 4 dans la classe 1. On obtient alors la solution sous la forme $[[\mathbf{0}, \mathbf{3}], [\mathbf{1}, \mathbf{2}, \mathbf{4}]]$.

2.1.2 Tableau unique

Le tableau unique correspond,comme son nom l'indique, à un simple tableau. Sa taille est égale au nombre de sommets dans le graphe, et chaque index du tableau correspond au numéro d'un sommet. La valeur de chaque case est le numéro de classe auquel appartient le sommet ayant pour index cette case.

Prenons comme exemple un graphe de 5 sommets partitionné en 2 classes, avec les sommets 0 et 3 dans la classe 0, et les sommets 1, 2 et 4 dans la classe 1. On obtient alors la solution sous la forme $[\mathbf{0}, \mathbf{1}, \mathbf{1}, \mathbf{0}, \mathbf{1}]$.

2.2 Méthodes de voisinage

2.2.1 Swap

Le mouvement de swap consiste, à partir d'une solution, à échanger de place deux éléments de cette solution. Dans le cas du partitionnement de graphe, ce mouvement est réalisé en prenant un premier sommet dans une classe, un deuxième sommet dans une autre classe, et en les échangeant de classe.

Un exemple de swap avec un graphe de 5 sommets partitionné en 2 classes, avec les sommets 0 et 3 dans la classe 0, et les sommets 1, 2 et 4 dans la classe 1 serait :

1. Dans la classe 0, prendre le sommet 3
2. Dans la classe 1, prendre le sommet 1
3. Mettre le sommet 3 dans la classe 1
4. Mettre le sommet 1 dans la classe 0

On passe donc d'une solution $[\mathbf{0}, \mathbf{1}, \mathbf{1}, \mathbf{0}, \mathbf{1}]$ à une solution $[\mathbf{0}, \mathbf{0}, \mathbf{1}, \mathbf{1}, \mathbf{1}]$.

Le swap est un mouvement qui **conserve la structure** de la solution de départ. C'est à dire que si la solution possède 3 classes avec 5 sommets dans chacune des classes, le mouvement ne modifiera pas cette configuration. Dans le cas où la tolérance ne dépasse pas 1, ce n'est pas un problème car il n'y a qu'une configuration possible. Mais si la tolérance est supérieure à 1, le mouvement de swap ne permettra pas d'atteindre toutes les solutions possibles du problème.

La taille du voisinage avec le mouvement swap est égale à $\sum_{i=1}^{p-1} \sum_{j=i+1}^{p} k_i k_j$, avec k_i le nombre de sommets dans la classe i. Si l'on prend $k = \frac{n}{p}$, on retrouve une estimation de l'ordre de n^2.

2.2.2 Pick'n'drop fallback swap

Le mouvement de pick'n'drop consiste, à partir d'une solution, à déplacer un élément de cette solution. Dans le cas du partitionnement de graphe, ce mouvement est réalisé en prenant un sommet dans une classe et en le déplaçant dans une autre classe.

Un exemple de pick'n'drop avec un graphe de 5 sommets partitionné en 2 classes, avec les sommets 0 et 3 dans la classe 0, et les sommets 1, 2 et 4 dans la classe 1 serait :

1. Dans la classe 0, prendre le sommet 3
2. Mettre le sommet 3 dans la classe 1

On passe donc d'une solution $[\mathbf{0}, \mathbf{1}, \mathbf{1}, \mathbf{0}, \mathbf{1}]$ à une solution $[\mathbf{0}, \mathbf{1}, \mathbf{1}, \mathbf{1}, \mathbf{1}]$.

Comme on peut le voir dans l'exemple, le pick'n'drop est un mouvement qui **ne conserve pas la structure** de la solution de départ. Une solution de départ $\langle \mathbf{5}, \mathbf{5}, \mathbf{5} \rangle$ aura ainsi une configuration différente après ce mouvement de type $\langle \mathbf{4}, \mathbf{5}, \mathbf{6} \rangle$. Cela implique que la tolérance peut ne plus être respectée suite à un pick'n'drop (une tolérance de 1 dans cette exemple ne permet pas de réaliser ce mouvement).

C'est pourquoi nous avons ajouté un système de **fallback**, qui consiste à réaliser un un deuxième pick'n'drop si la contrainte de tolérance n'est pas respectée. Si l'on reprend l'exemple précédent, on obtient une nouvelle solution $[\mathbf{0}, \mathbf{1}, \mathbf{1}, \mathbf{1}, \mathbf{1}]$, qui ne satisfait pas une tolérance de 1. On réalise alors un pick'n'drop en prenant un sommet dans la classe dans laquelle nous avons ajouté un nouvel élément, et en le mettant dans l'ancienne classe de cet élément. On retombe alors sur un mouvement de **swap**. En choisissant le sommet 1, on obtient la solution $[\mathbf{0}, \mathbf{0}, \mathbf{1}, \mathbf{1}, \mathbf{1}]$, ce qui revient à avoir fait le même mouvement que dans l'exemple du swap.

La taille du voisinage avec le mouvement pick'n'drop est égale à $n\,(p-1)$, avec n le nombre de sommets dans le graphe et p le nombre de classe. Si l'on ajoute le voisinage atteignable par le fallback, on obtient alors une taille de voisinage pour ce mouvement compris entre la taille du voisinage de pick'n'drop et la taille du voisinage de swap, dépendant de la tolérance et du nombre de sommet du graphe par rapport au nombre de classe.

2.3 Méthodes de résolution

2.3.1 Énumération

2.3.2 Descente de gradient

2.3.3 Recuit simulé

2.3.4 Recherche Tabou

2.3.5 Algorithme génétique