Rapport de projet

Implémentation de méthodes heuristiques pour le partitionnement de graphes

Paul-Emile Boutoille
Claire Pennarun
Tatiana Rocher

Bordeaux 1
Graphes et Recherche Opérationnelle

8 janvier 2013

Table des matières

1 Présentation du programme

1.1 But

Le but de notre projet est d'implémenter plusieurs algorithmes d'heuristiques pour la résolution du problème de partitionnement d'un graphe. Il faudra séparer en un nombre donné de classes de sommets avec un minimum d'arêtes interclasses et en ayant au maximum des classes de tailles équivalentes.

Ce programme permet ainsi de faire des tests de rapidité et de comparer la vitesse des algorithmes en fonction du nombre de sommets du graphe et du nombre de classes.

1.2 Mode d'emploi du programme

Le programme doit être utilisé sur des fichiers de la forme suivante :

```
# nbSommets nbAretes
4 4
# dmin dmax
1 3
# source but valeur
1 2 1
1 3 1
2 3 1
3 4 1
# sommet degre
1 2
2 2
3 3
4 1
```

La dernière partie du fichier (qui donne les degrés de chaque sommet) n'est pas nécessaire car elle n'est pas utilisée par notre programme.

Notre programme se lance avec les arguments suivants :
<fichier> <algorithme> <nbClasse> <nbLancements> <option1> <option2>

Les deux derniers arguments ne sont nécessaires que dans certains cas, mais ne font pas planter le programme si ils sont présents.

- *<fichier>* : Le chemin du fichier sur lequel on veut travailler. Chaque fichier représentant un graphe.
- *<algorithme>* : L'algorithme que l'on veut utiliser : "ex" pour l'algorithme exhaustif, "gradient" pour l'algorithme de descente de gradient, "recuit" pour l'algorithme

du recuit simulé et “tabou” pour l’algorithme tabou.

- <*nbClasse*> : Le nombre de classes voulu pour le partitionnement.
- <*nbLancements*> : Le nombre de lancements de l’algorithme choisi (avec les mêmes paramètres).
- <*option1*> : Représente le voisinage que l’on veut utiliser : “PnD” pour le voisinage Pick and Drop, “Swap” pour le voisinage swap. Cet argument n’est pas nécessaire pour lancer l’algorithme exhaustif mais obligatoire pour les autres.
- <*option2*> : Est lié à l’algorithme :
  - La température initiale pour l’algorithme du Recuit Simulé (un double)
  - La taille du tableau des mouvements tabou pour l’algorithme Tabou (int supérieur à 0).
  - Inutile pour les autres algorithmes

2 Choix généraux

2.1 Choix des algorithmes implémentés

Nous avons implémenté quatre algorithmes :
- Algorithme exhaustif
- Algorithme avec descente de gradient
- Algorithme de recuit simulé
- Algorithme Tabou

Les trois premiers algorithmes étaient obligatoires, et nous avons choisi comme quatrième algorithme l'algorithme Tabou, car la gestion d'un tableau de mouvements interdits nous paraissait intéressante. Les trois derniers algorithmes sont lancés plusieurs fois sur plusieurs solutions initiales. Nous détaillons plus loin cette fonctionnalité.

2.2 Langage de programmation

Nous avons choisi de programmer en Java, car c'est un langage que nous connaissons et qu'un langage objet nous paraissait pertinent pour gérer des objets comme des solutions ou des graphes. De plus, Java nous permet d'utiliser les fonctions liées à ses bibliothèques qui sont déjà optimisées. Nous savions donc que si un algorithme s'exécutait dans un temps supérieur à la normale, ce serait de la faute de l'algorithme et non des fonctions Java.

2.3 Structures de données

Pour stocker le graphe en mémoire, nous avons choisi d'utiliser une liste d'adjacence qui est remplie à la lecture du fichier par le programme. Cette liste est ensuite traitée par le programme pour créer un premier partitionnement (stocké dans l'objet GraphePartition), qui est la base de notre programme et sur lequel toutes les méthodes seront appliquées. Cet objet GraphePartition représente un graphe partitionné et contient notamment le numéro de classe attribué à chaque sommet, le nombre de classes prévues et l'évaluation de la solution courante.

2.4 Voisinages implémentés

Nous avons décidé d'implémenter deux voisinages, le Pick'n'Drop et le Swap. Ainsi l'utilisateur pourra choisir entre eux. Nous n'avons pas implémenté le voisinage par Sweep car il revient à effectuer plusieurs Swap d'affilée.

2.5 Solution initiale

Pour la solution initiale, chaque sommet du graphe est placé dans l'une des classes aléatoirement. Nous avons implémenté deux fonctions différentes suivant le voisinage utilisé : pour le voisinage Swap, on choisit un écart maximal pour le nombre de sommets par classe. Pour le voisinage Pick'n'Drop, la répartition des sommets dans les classes est totalement aléatoire.

2.6 Evaluation

La fonction d'évaluation se présente en deux étapes. Notre démarche à été de calculer une première évaluation simpliste (on ajoute 1 par arête interclasse) puis d'appliquer un malus correspondant au respect de l'égalité des classes.

Pour une première évaluation de x, si l'écart moyen des classes est de 13%, on ajoutera a x ces 13%. Ainsi :

1. Calcul de I1 : La première évaluation est le nombre arêtes interclasses, cette étape ne sera effectuée qu'une seule fois. *Cette évaluation sera ensuite mise à jour pour chaque sommet déplacé pour économiser du temps.*
2. Calcul de I2 : La deuxième évaluation est calculée en fonction de I1 et de e, l'écart moyen du cardinal des classes (en pourcentage du nombre de sommets). On aura I2 = (100 + I1)*(1 + e*a). Nous avons ajouté 100 pour minorer l'évaluation, et ne pas avoir I2 = I1 quand I1 = 0. Le coefficient a sera réglé pour donner plus ou moins d'importance à l'égalité des classes (un a grand fera un grand malus). *Cette évaluation sera recalculée en fonction de I1 à chaque fois que l'utilisateur la demandera.*

Exemple : Pour un graphe G de 1000 sommets ayant une solution avec trois classes de taille 320, 335, 345 et 712 arêtes interclasses (on suppose a = 1) :

– I1 : Pour 712 arêtes interclasses, la première évaluation sera I1 = 712

– I2 :

- Ecart moyen : E = ((345-335)+(345-320)+(335-320))/3 = 13.33.
- Ecart moyen en pourcentage : e = E / nbSommet = 13.33/1000 = 0.013 (1.3%)
- Calcul final : I2 = (100 + I1)*(1+e) = (100 + 712)*1.013 = 822.556

2.7 Lanceur

Ce module sert à lancer en boucle un algorithme, en utilisant le multi-threading. Quand il est instancié, il aura comme paramètres :

– Le graphe initial (sur lequel travailler)
– Le type de voisinage (PnD ou swap)
– Le nombre de classes désirées

NB : On peut noter que ces paramètres sont les mêmes que ceux pour instancier un algorithme, le lanceur en aura besoin puisque qu'il créera des algorithmes en boucle.

Ensuite, on utilisera la fonction run() du lanceur pour démarrer les tests. Cette dernière a plusieurs arguments :

- Un string représentant le nom de l'algorithme à tester
- Le nombre de lancements à effectuer
- L'argument optionnel de l'algorithme. *NB : Cet argument est optionnel pour run(), et il sera ignoré s'il doit lancer la descente de gradient (qui n'a pas besoin d'argument)*

Il lance alors n algorithmes en parallèle, et indiquera à la fin :

- Chacun des algorithmes lancé avec :
  - Leur solution initiale, et son évaluation
  - Leur temps d'exécution.
  - Leur solution finale, et son évaluation
- La meilleure solution trouvée (et son évaluation)
- L'évaluation moyenne
- Le temps total d'exécution
- Le temps moyen d'exécution d'un algorithme.

NB : Les temps relatifs à chacun des algorithmes (et par conséquent le temps moyen) sont altérés. En effet, le multi-threading rend le calcul du temps faux (entre le début de l'algorithme et sa fin, d'autres processus ont eu la main). Pour un meilleur temps moyen approximatif, nous avons divisé le temps total par le nombre d'algorithmes lancés.

2.8 Tests

Nous avons choisis de tester (principalement) le PnD et de comparer avec quelques résultats du Swap. Nous avons fait varier les nombres de sommets, les nombres de classes et effectué une analyse critique des résultats. Nos tests ont été effectués sur les machines du Cremi, ce sont des machines 4 coeurs.

3 Algorithme exhaustif - Enumération explicite

3.1 Description

Cet algorithme va examiner toutes les solutions possibles. Nous avons choisi d'effectuer un parcours en profondeur. Nous avons considéré le tableau des sommets, avec pour valeur un entier représentant leur classes.

Nous l'avons ensuite initialisée à 0, et "compté" en base n (n étant le nombre de classe). Exemple pour 2 classes et 4 sommets :
$[0, 0, 0, 0]$
$[1, 0, 0, 0]$
$[0, 1, 0, 0]$
$[1, 1, 0, 0]$
(...)

De cette manière nous sommes sûrs de tout tester, nous pouvons cependant remarquer que nous testons des solutions équivalentes : $[0, 0, 1, 1] \Leftrightarrow [1, 1, 0, 0]$.

3.2 Paramètres

Il n'est pas nécessaire de choisir un voisinage. La solution initiale est une solution avec tous les sommets dans la première classe (0).

3.3 Tests effectués

3.3.1 Graphe à 5 sommets

nb de classes	2	3	4
temps (ms)	4	16	65

3.3.2 Graphe à 10 sommets

nb de classes	2	3	4	5
temps	55ms	2s 773ms	50s 800ms	7min 54s

3.3.3 Autres graphes

Graphe à 15 sommets / 2 classes : 1s 843ms
Graphe à 20 sommets / 2 classes : 1min 12s
Graphe à 22 sommets / 2 classes : 2min 30s
Graphe à 23 sommets / 2 classes : 10min 50s

3.4 Résultats

C'est un algorithme qui devient très rapidement long, mais qui donne toujours la meilleure solution. Il est donc préférable de l'utiliser sur des petits graphes.

Il est utilisable quelque soit le nombre de classes jusqu'à 10 sommets et pour 2 classes jusqu'à 22 sommets.

4 Descente de gradient

4.1 Description

Cet algorithme part d'une solution initiale aléatoire et tourne tant qu'une solution meilleure a été trouvée dans son voisinage. On cherche à chaque tour de boucle la meilleure solution voisine. Si celle-ci est meilleure que la solution courante, elle devient solution courante. Sinon, l'algorithme s'arrête.

4.2 Paramètres

Cet algorithme dépend du type de voisinage de la solution.

4.3 Tests

classes	sommets	lancers	tps moyen	tps total	eval min	ecart moyen
2	20	20	1 ms	13 ms	140	7.95
2	50	50	1 ms	44 ms	321	21.86
2	100	100	1 ms	115 ms	1010	63.94
2	500	500	2 ms	1 s 84 ms	3522	242.06
2	1000	1000	4 ms	4 s 953 ms	4705	277.671
5	20	20	4 ms	32 ms	140	7.5
5	50	50	3 ms	64 ms	308	33.2
5	100	100	2 ms	148 ms	1641	56.16
5	500	500	3 ms	2 s 332 ms	5811	186.864
5	1000	1000	7 ms	8 s 148 ms	7718	230.805

4.4 Résultats

Nous pouvons constater que la convergence de l'algorithme est extrêmement rapide. En revanche, la qualité des résultats trouvé est moindre. C'est pourquoi il faut, pour un nombre de sommet donnée, faire beaucoup de lancement. Malgrès cela, la rapidité et la faible efficacité restent des caractéristiques de la recherche de gradient.

5 Recuit simulé

5.1 Description

Cet algorithme s'est inspiré de la physique. Il utilise une température comme paramètre. Il est composé d'une grand boucle qui fait baisser ittérativement la température, jusqu'à un certain point, et d'une petite boucle. Le rôle de la petite boucle est de chercher, selon la température, une bonne solution courante. A la température minimale, la solution trouvée doit être optimale.

5.2 Paramètres

Cet algorithme dépend du voisinage. De plus, nous laissons à l'utilisateur le choix de la température initiale, c'est un double qui doit être supérieure à la température minimale.

5.3 Choix d'implémentation

Taille du problème Dans cet algorithme nous utilisons N, la taille du problème. Nous avons décidé de prendre le nombre d'arêtes comme taille. Le nombre d'arêtes minore le nombre de sommets car nous sommes sur des graphes simples. En revanche, le nombre de sommet de minore pas le nombre d'arête. Nous pensons donc que le nombre d'arêtes est plus représentatif des problèmes appliqués aux graphes simples.

Température initiale Elle est donnée au choix à l'utilisateur. Celle-ci est un double qui doit être supérieure à la température minimale, 0.1° . Sachant que la baisse de température est assez rapide (voir suite) la température doit être assez conséquente, soit supérieure à 20°, pour que l'algorithme tourne suffisamment longtemps.

Choix condition 1 Afin de sortir de la boucle, l'algorithme doit répondre à l'une des deux conditions suivantes :

- soit on a atteint la température minimale, fixée à 0.1°
- soit la température a changé 5 fois et la solution courante n'a pas changé.

Nous pensons que ces deux conditions sont importantes car cela permet d'arrêter plus rapidement l'algorithme, et donc d'optimiser les temps d'exécution. Si l'utilisateur choisit une température extrêmement élevée par rapport au problème, la condition sur k permet tout de même une réponse rapide.

Choix condition 2 Afin de sortir de cette deuxième boucle, l'algorithme doit répondre à l'une des trois conditions suivantes :

- soit on a fait N*N tours de boucle

– soit on a tiré 10*N solutions dont l'évaluation est plus grande que la solution courante
– soit on a accepté 10*N solutions dont l'évaluation est plus grande que la solution courante

Ces trois conditions nous paraissaient nécessaires dans le cas des grands graphes. Dans les petits graphes la boucle s'arrête quasi-systématiquement après accomplissement de la première condition, dans le cas des grands graphes, l'une des deux conditions suivantes est prévalente, ce qui est du à la croissance moins rapide de la borne maximale.

Choix de la solution voisine La solution voisine doit être aléatoire. On effectue simplement un mouvement du voisinage en cours.

Choix de la fonction g La fonction g est la fonction de chute de la température. Celle-ci est simple : on multiplie la température actuelle par la raison r. Au début de l'algorithme, r est choisie au hasard entre 0.7 et 0.9 pour que l'algorithme de Métropolis puisse accepter suffisamment de solutions supérieures pour pouvoir sortir d'un minimum local.

5.4 Tests effectués

Voici quelques tests effectués. Afin de les comparer nous utilisons un test témoin, puis nous faisons varier un paramètre à la fois.

Température initiale : 20°C
Voisinage : PnD
Nombre de classes : 2

sommets	lancers	tps moyen	tps total	ecart moyen	eval min
20	20	22 ms	500 ms	2	140
50	50	130 ms	6 s 300 ms	6.26	315
100	75	430 ms	32 s	10	1025
500	100	2 s 100 ms	3 min 30 s	41	3648
1000	300	5 s 500 ms	29 min	42	4873

Changement du voisinage

Température initiale : 20°C
Voisinage : Swap
Nombre de classes : 2

sommets	lancers	tps moyen	tps total	ecart moyen	eval min
20	20	37 ms	750 ms	35	140
50	50	210 ms	10 s 500 ms	75	301
100	75	1 s 150 ms	1 min 26 s	144	1003
500	100	3 s 800 ms	6 min 25 s	290	3482

Changement de la température

Température initiale : 60°C
Voisinage : PnD
Nombre de classes : 2

sommets	lancers	tps moyen	tps total	ecart moyen	eval min
20	20	40 ms	809 ms	1.45	140
50	50	221 ms	11 s 94 ms	6.88	315
100	75	500 ms	38 s	18	1022
500	100	2 s 500 ms	4 min 15 s	43	3653

Changement du nombre de classes

Température initiale : 20°C
Nombre de sommets : 50
Voisinage : PnD

nb classes	tps moyen	tps total	ecart moyen	eval min
5	187 ms	9 s 400 ms	5	479
10	250 ms	12 s 600 ms	5	537
20	360 ms	18 s 200 ms	6	562
30	300 ms	15 s 300 ms	33	536

Température initiale : 20°
Nombre de sommets : 100
Voisinage : PnD

nb classes	tps moyen	tps total	ecart moyen	eval min
5	245 ms	18 s 400 ms	15	1655
10	332 ms	25 s	11	1874
35	1 s 250 ms	1 min 35 s	6	2039
50	3 s	3 min 20 s	16	203
75	6 s 700 ms	8 min 21 s	17	2025

5.5 Résultats

Lorsque l'on change du voisinage Pick and Drop au voisinage Swap, les solutions trouvées sont bien meilleures. Cela provoque une augmentation de l'écart moyen.

Lorsque l'on augmente la température, le temps d'exécution est à peine plus élevé. La solution minimale est à peu près la même mais l'écart entre les solutions est très faible avec un température élevée. Cela est due au fait qu'avec une plus grande température, la boucle externe dure plus longtemps.

Lorsque l'on augmente le nombre de classes, le temps d'exécution augmente très peu. Cette augmentation se ressent à partir de graphes de 100 sommets.

6 Algorithme Tabou

6.1 Description

Cet algorithme est une heuristique qui utilise un tableau de mouvements interdits (ou tabous) pour pouvoir sortir des minima locaux. On stocke les mouvements interdits plutôt que les solutions interdites par souci d'économie de la mémoire.

6.2 Paramètres

Le tableau de mouvements interdits (ou tableau tabou) est un tableau d'objets de type Mouvement. Sa taille est fixée par l'utilisateur à l'appel de l'algorithme.

Un mouvement est un objet qui contient plusieurs champs : deux champs représentant des sommets, un champ représentant une classe et un champ représentant le timestamp associé à ce mouvement. Cette représentation permet de gérer des mouvements de type Pick'n'Drop (en instanciant seulement un des deux sommets et le numéro de la classe) et des mouvements de type swap (en instanciant les champs correspondant aux deux sommets à échanger). Le timestamp de chaque mouvement, qui correspond à la durée de présence du mouvement dans le tableau des mouvements tabous, est incrémenté à chaque tour de boucle et le mouvement est retiré du tableau quand son timestamp atteint la valeur de la taille du tableau.

6.3 Tests effectués

Pour effectuer les tests, nous avons pris comme témoin un tableau tabou de taille 5 et un partitionnement en deux classes.

Taille du tableau Tabou : 5
Voisinage : PnD
Nombre de classes : 2

sommets	lancers	tps moyen	tps total	ecart moyen	eval min
20	20	31 ms	75 ms	1.2	140
50	50	49 ms	283 ms	7.74	291
100	75	3 s 27 ms	4 s 401 ms	16.49	935
500	100	10 min 55 s 624 ms	11 min 6 s 733 ms	39.78	2952

Changement de la taille du tableau Tabou :
Taille du tableau Tabou : 8
Voisinage : PnD
Nombre de classes : 2

sommets	lancers	tps moyen	tps total	ecart moyen	eval min
20	20	73 ms	131 ms	1.05	140
50	50	89 ms	365 ms	8.6	291
100	75	4 s 89 ms	4 s 856 ms	13.84	937
500	100	11 min 4 s 350 ms	11 min 15 s 518 ms	73.11	2923

Changement du nombre de classes :
Taille du tableau Tabou : 5
Voisinage : PnD
Nombre de classes : 5

sommets	lancers	tps moyen	tps total	ecart moyen	eval min
20	20	105 ms	180 ms	2.1	172
50	50	907 ms	1 s 262 ms	10.5	443
100	75	22 s 524 ms	24 s 351 ms	24.84	1519
500	100				

Changement du voisinage utilisé :
Taille du tableau Tabou : 5
Voisinage : Swap
Nombre de classes : 5

sommets	lancers	tps moyen	tps total	ecart moyen	eval min
20	20	46 ms	172 ms	12	206
50	50	8 s 401 ms	9 s 591 ms	82.92	336
100	75	6 min 58 s 772 ms	7 min 8 s 746 ms	196.68	1092
500	100				

6.4 Résultats

En augmentant la taille du tableau Tabou, le temps de calcul augmente mais le pourcentage d'augmentation tend vers zéro avec l'augmentation de la taille du problème :

- 20 sommets / 20 lancers : 108 % d'augmentation
- 50 sommets / 50 lancers : 81 % d'augmentation
- 100 sommets / 75 lancers : 35 % d'augmentation
- 500 sommets / 100 lancers : 1,3 % d'augmentation

En augmentant le nombre de classes, le temps de calcul moyen augmente fortement :

- 20 sommets / 20 lancers : 238 % d'augmentation
- 50 sommets / 50 lancers : 1751 % d'augmentation
- 100 sommets / 75 lancers : 644 % d'augmentation

Ceci est normal, puisque la taille du voisinage augmente aussi considérablement.

En changeant de voisinage (Swap au lieu de Pick'n'Drop), les temps de calcul moyens augmentent également, car la taille du voisinage est plus importante.

7 Méthode de travail

7.1 Utilisation de git

Pour travailler sur ce projet, nous avons utilisé le logiciel de gestion de versions Git et sa plateforme d'hébergement web GitHub. Nous avons travaillé en mettant en place des branches de travail, qui étaient ensuite intégrées au code principal. Nous avons également utilisé le système des "issues" de GitHub, qui nous ont permis de lister les tâches à effectuer et donc d'avoir un suivi global du projet.

8 Conclusion

Ce projet nous a permis de tester la validé et l'efficacité des différents algorithmes. La partie test étant prépondérante, nous avons pu constater l'importance des réglages, et se rendre compte des différences de temps de calcul.

Nous avons, tout au long de notre implémentation, essayé de mettre un accent sur l'efficacité de l'architecture utilisée. En limitant les calculs et en utilisant le multi-threading, nous nous sommes à plusieurs reprisés penché sur certains problèmes d'optimisation.

Au sein de ce développement, nous avons acquis une expérience conséquente dans le problème du partitionnement des graphes. Cette introduction au méta-heuristiques nous a donné un avant goût des problèmes à optimisation.