Fiche Projet Méta-Heuristiques

mmc
marc-michel dot corsini at u-bordeaux dot fr

29 octobre 2021

Le travail demandé représente une charge étudiant de l'ordre de 20h. Le projet étant réalisé par groupe de deux. Le travail à remettre est un rapport au format **pdf** et un code avec une page texte expliquant sa mise en œuvre.

1 Description

À partir d'un graphe non orienté, éventuellement pondéré dans $\mathbb{R}$. Il faudra partitionner le graphe en p classes, de telle sorte que la somme des poids entre sommets n'appartenant pas à la même classe soit minimale. De plus il faudra s'assurer que les sommets du graphe sont répartis de manière **(à peu prés)** équitable dans les p classes.
Vous avez toute latitude pour définir, le nombre de classes et la notion d'« à peu prés équitable ». L'objectif du travail est de pouvoir comparer différentes approches de partitionnement, en fonction de la qualité de la solution trouvée, du temps de calcul, de la limite d'utilisation des méthodes en fonction de la taille du problème et éventuellement d'autres critères.
Pour mener à bien votre comparaison, il faudra implémenter les méthodes simples permettant de trouver la solution optimale afin de garantir que les méta-heuristiques implémentées soient cohérentes.

- Un algorithme d'énumération pouvant être explicite / implicite. Cet version doit fournir l'optimum.
- Un algorithme glouton du `gradient` **sans optimisation**, c'est-à-dire que l'algorithme parcourt **tout** le voisinage d'une solution pour trouver l'optimum local.
- La première métaheuristique sera le `recuit simulé`.
- La seconde métaheuristique sera au choix :
  1. Méthode tabou **sans aspiration** ;
  2. Réseau de Hopdield, si vous vous restreignez à la résolution en 2 classes ;
  3. Algorithme génétique simple ;
  4. Agents ;
  5. Méthode GRASP.
- Enfin, de manière facultative, vous pourrez introduire des heuristiques dans la descente de gradient ou la méthode tabou.

Vos algorithmes doivent être testés et validés, et doivent pouvoir traiter **tous** les exemples de graphes accessibles sur le site **Box.com** via le lien sur la page **marcmichelcorsini/MIMSE**.

1.1 Structure fichiers fournis

Les fichiers de graphes à traiter sont basés sur une syntaxe simple, un commentaire débute par un dièse (#). Les quatre premières valeurs sont : le nombre de sommets du graphe, le nombre d'arêtes du graphe, le degré minimum du graphe et enfin le degré maximum du graphe. Puis viennent pour chaque ligne 2 ou 3 valeurs, qui donnent les extrémités d'une arête et son poids. Si le poids est manquant, il vaut 1. Les sommets sont fournis dans l'ordre croissant. Une fois que le nombre d'arêtes est atteint, le dernier bloc est constitué de 2 nombres, le premier est un sommet, le second est le degré

du sommet dans le graphe.
Pour les plus anciens fichiers, les 4 permières valeurs sont sur 2 lignes et tous les poids sont à 1. Il y a donc (hors les lignes commentaires) 2 + card(E) + card(V) lignes dans un fichier.

2 Programmation

L'environnement de programmation est laissé à votre entière discrétion, il suffit qu'il soit accessible sur n'importe quelle plateforme et libre. Vous ne pouvez pas utiliser `matlab`, mais vous pouvez choisir `scipy`. Soit vous fournissez un makefile, soit vous devez préciser la méthode pour pouvoir obtenir l'exécutable. Si vous avez programmé dans un langage, précisez la version, afin de me faciliter la tâche.

3 Document écrit

Le document a fournir n'est pas un rapport de programmation mais une étude de performances de différentes méthodes pour résoudre un problème particulier. Il est nécessaire de présenter le problème à résoudre, les choix et les raisons de vos choix pour les paramètres des méthodes de résolution.
En particulier, le nombre de classes est-il fixe ou bien variable, quelle définition de l'« à peu prés équité », quel(s) voisinage(s) avez-vous utilisé(s).
Les méta-heuristiques servent à résoudre des problèmes de grandes tailles, ne les appliquer que sur des petites instances d'un problème n'a pas de sens.
Votre document doit faire état de comparaisons entre les différentes approches que vous avez mis en œuvre. Ces comparaisons portent sur la qualité des solutions, les temps d'exécution de vos implémentations, ce qui veut dire que vous devez spécifier le matériel que vous avez utilisé (taille mémoire, système d'exploitation) mais aussi que vous ayez conscience que le choix des structures de données pour représenter un graphe et une solution ont un impact considérable sur les temps de calcul.

4 fiches Jalon

Les jalons sont facultatifs, ils ont pour but de vous aidez dans l'organisation de votre travail. Ils mettent en évidence certains éléments clefs pour la construction de votre rapport.

4.1 Jalon 01, exploratoire

Puisqu'on doit résoudre un problème d'optimisation sur un graphe, il y a un certain nombre de choses à régler

1. Définir la structure de stockage du graphe
2. Définir la fonction à optimiser
3. Connaissant le graphe et une solution, renvoyer le nombre d'arêtes inter-classes
4. Connaissant le graphe et une solution, renvoyer la valeur de la fonction à optimiser
5. Définir ce qu'est un mouvement élémentaire pour un voisinage donné.

4.2 Jalon 02, énumération

La fonction objectif étant choisie.

- être capable de faire un calcul direct à partir d'une solution
- être capable de faire un calcul indirect à partir d'une solution et d'un mouvement élémentaire
- implémenter une méthode énumérative qui cherche la solution optimale dans l'espace de recherche.

4.2.1 Énumération

Énumérer c'est compter. Il suffit donc d'attribuer un entier à chaque partition possible du graphe. Si on considère un graphe à n sommets, que l'on veut partitionner en p classes, il existe p^n solutions possibles. Une idée simple consiste donc à noter les solutions sur n valeurs en base k.
Même dans ce cas, il est possible, a priori de déterminer une borne maximale strictement inférieur à p^n du fait que le numéro de la classe à laquelle un sommet est affecté n'a aucune incidence sur la valeur de la fonction à optimiser. Par exemple, pour un graphe à 4 sommets en 2 classes, la configuration $c_0c_1c_2c_3$ est identique à la configuration $\overline{c_0}\,\overline{c_1}\,\overline{c_2}\,\overline{c_3}$.
Néanmoins, l'espace de recherche est défini en fonction des contraintes du problème. Si toute solution est une solution réalisable, il faudra énumérer les p^n configurations. Par contre si une solution n'est pas forcément réalisable, il est souhaitable d'adjoindre un mécanisme permettant de ne pas explorer des solutions inutiles.

Generate and test
Cet algorithme est très simple à mettre en place

```
Pour i de 1 à p^n faire
   construire la ième solution
   Si la solution est réalisable Alors
      Evaluer
      Si meilleure que la dernière meilleure Alors Sauvegarder Fsi
   Fsi
Fpour
Renvoyer la dernière sauvegarde
```

Si vous avez réduit l'espace de recherche à l'espace des solutions réalisables, dont vous avez calculé la taille N, le principe est exactement le même. Il faut explorer les N solutions pour découvrir l'optimum global. Du point de vue de l'espace de recherche, il s'agit d'une énumération implicite, puisqu'on ne regarde qu'une partie des solutions.

Tests
L'algorithme d'énumération n'est applicable que sur les petites instances (moins de 30 sommets). Le nombre de solutions à évaluer, même sur l'espace de recherche réalisable est assez colossal. Faites des graphiques reportant le temps en fonction du nombre de sommets et du nombre de partitions.

4.3 Jalon03, gradient

L'algorithme de descente de gradient est dépendant des conditions initiales. Cet algorithme trouve l'optimum local de la fonction à optimiser accessible depuis la configuration initiale.
Le principe de l'algorithme de descente est le suivant :

```
Soit C_0 la configuration initiale
Soit m le mouvement élémentaire
Soit V(C) le voisinage de C

i <- 0
fini <- faux
TantQue pas fini Faire
   Calculer fopt(C_i)
   Calculer V(C_i) = { C tel que C = m(C_i) }
   Pour chaque C de V(C_i) Faire
      Calculer fopt(C)
   FPour
   Soit C_{i+1} de V(C_i) tel que fopt(C_{i+1}) = min(fopt(C))
   Si f(C_{i+1}) < f(C_i) Alors i <- i+1
   Sinon fini <- vrai
   Fsi
FTantQue
Renvoyer C_i
```

Les objectifs sont :

- Etant donnée une configuration, être capable d'accéder à chaque configuration du voisinage.
- Mettre en place l'algorithme du gradient pour une configuration initiale.
- Mettre en place le tirage aléatoire de la configuration initiale.
- Faire une boucle permettant d'obtenir le meilleur optimum local, le pire, la valeur moyenne, la médiane, l'écart-type, le nombre de fois où le meilleur optimum local a été trouvé .... Bref toute statistique que vous jugerez utile pour les comparaisons entre les différents algorithmes implémentés.

```
Input : G, n (nombre de sommets)
int f(n) // renvoie un nombre qui depend de n

Pour i de 1 a f(n) faire
 Choisir une configuration initiale
 Faire tourner l'algorithme du gradient
Fait
Renvoyer les statistiques
```

Tests Sur les petites instances, la méthode du gradient doit être capable de trouver l'optimum global. Comparez les temps d'exécution avec ceux de l'énumération.

4.4 Jalon04, recuit simulé

Les objectifs sont :

- Étant donnée une configuration, être capable d'extraire aléatoirement une configuration de son voisinage.
- Être capable de calculer de manière automatique les paramètres utiles pour le Recuit Simulé tels que :
  - Temperature initiale : pour cela vous devez estimer la qualité de la configuration initiale par rapport à son voisinage.
  - Longueur de la chaîne de Markov, en théorie cette chaîne doit être infinie pour aboutir à l'équilibre thermodynamique à température constante. En pratique, il faut qu'elle soit suffisamment longue.
  - Définition du critère d'arrêt « système figé » : Température minimale atteinte, amélioration epsilonesque, durée maximale atteinte, ...
  - Mettre en place un chronomètre afin de mesurer la performance des différents algorithmes
  - Mettre en place une sortie des résultats afin de pouvoir analyser les performances dans votre rapport

  Au départ il est plus simple de fournir une température initiale arbitraire, et un nombre d'itérations maximum pour la boucle externe. Pour la boucle interne, on choisira un nombre d'itérations suffisant pour espérer trouver un état stable. Ce nombre d'itérations ne peut, en aucun cas être inférieur à

$$\min(\text{card}(V), \text{card}(E))$$

  sous peine de perdre totalement l'intérêt de l'approche par recuit.

Tests Sur les petites instances, le recuit doit être capable de trouver l'optimum global. Sur les grosses instances, la taille de la boucle interne (température constante) a un impact significatif sur la qualité des solutions obtenues. Comparez les temps d'exécutions avec ceux de la méthode du gradient.

5 Jalon heuristiques gradient

Une fois réalisée la seconde méta-heuristique, on pourra s'intéresser aux heuristiques possibles pour améliorer la performance de l'algorithme de gradient

1. Au lieu de chercher l'optimum d'un voisinage, on pourra étudier l'impact de la recherche d'une kème solution améliorante.

```
input C_0 la configuration initiale
input k le nombre de configurations améliorantes
V(C) le voisinage de C

i <- 0
fini <- faux
TantQue pas fini Faire
  Calculer fopt(C_i)
  j <- 0
  TantQue j < k et V(C_i) non vide Faire
       Soit C ∈ V(C_i)
       Si fopt(C) < fopt(C_i) Alors
          j <- j + 1
          Sauvegarder C
       Fsi
   FTantQue
   Si Sauvegarde existe Alors
      C_{i+1} <- Sauvegarde
      i <- i+1
   Sinon
      fini <- vrai
   FSi
FTantQue
Renvoyer C_i
```

2. La taille d'un voisinage pouvant être très importante, il est possible de découper la voisinage en b blocs de taille similaire. On effectue une recherche de la meilleure solution dans le bloc considéré

```
Soit C_0 la configuration initiale
Soit m le mouvement élémentaire
Soit V_k(C) le kème bloc du voisinage de C
Soit b le nombre de blocs du voisinage V

i <- 0
fini <- faux
k <- 0
TantQue pas fini Faire
   Calculer fopt(C_i)
   Calculer V_k(C_i) = { C tel que C = m(C_i) }
   Pour chaque C de V_k(C_i) Faire
      Calculer fopt(C)
   FPour
   Soit C_{i+1} de V_k(C_i) tel que fopt(C_{i+1}) = min(fopt(C))
   Si f(C_{i+1}) < f(C_i) Alors
      i <- i+1
      k <- (k + 1) mod b
   Sinon fini <- vrai
   Fsi
FTantQue
Renvoyer C_i
```

3. On pourra, au sein de la seconde heuritique, introduire la première heuristique. C'est-à-dire qu'au lieu de chercher la meilleure solution d'un bloc du voisinage, on pourra s'intéresser à la kème configuration améliorante du bloc.

Il est entendu que les gains en temps risquent fortement d'être contrebalancés par la perte de qualité des solutions obtenues.

Tests Sur les petites instances, la descente de gradient avec heuristique doit être capable de trouver l'optimum global. Sur toutes les instances, elle doit être capable de trouver plus rapidement une réponse que ne le fait la descente de gradient sans heuristique. Comparez les temps d'exécution et la qualité des solutions avec la méthode de gradient et les 2 méta-heuristiques.

6 Limiter les temps d'exécution

Comme vous allez vite vous en rendre compte, les différents algorithmes sont gourmands en temps, il peut être souhaitable de mettre en place un système qui limite les calculs à une durée particulière.

7 Jalon analyse

La dernière étape consiste à analyser les résultats, une analyse purement descriptive est bien entendu la première étape. Cependant, il existe de nombreux moyens statistiques permettant de répondre à la question « Quel est le meilleur algorithme pour résoudre le problème **P** ? ».