31/12/2016

Rapport de TP

Métaheuristique, problème de sac à dos

Maxime DEGRES
Jean-Baptiste DURIEZ
Jordane QUINCY

Table des matières

Introduction du problème

Le but de ce mini projet est de résoudre le problème du sac à dos multidimensionnel avec des contraintes de demandes. Il consiste à remplir un sac d'objet en respectant des contraintes de demandes et des contraintes de capacités tout en maximisant le gain. Chaque objet rapporte quelque chose mais a également un poids (pour chaque dimension de poids) et un apport par rapport aux demandes. Il faut donc trouvé quels objets sont à mettre dans le sac pour avoir le meilleur gain. Nous utiliserons des heuristiques et méta-heuristiques pour résoudre ce problème.

Résolution du problème

Afin de résoudre ce problème nous avons choisi d'utiliser le langage C, puisque c'est un langage avec lequel on peut facilement avoir le contrôle sur l'allocation mémoire et donc où on peut plus facilement optimiser le code afin d'avoir les meilleurs résultats possibles. Nous aurions aussi pu utiliser le C++ mais nous sommes plus à l'aise en C et nous n'avons pas forcément besoin de notion d'objet pour ce problème.

1. Structure de données

Afin de modéliser le problème nous avons utilisé la même structure de données que celle fournie en TD pour le sac à dos multidimensionnel, nous avons juste ajouté les contraintes de demandes à la suite des contraintes de capacités dans la matrice ainsi que le nombre de contraintes de demandes.

Pour récapitulatif, le problème est donc représenté par un pointeur (mkp), pointant vers 3 entiers : n, cc et cd, représentant respectivement le nombre de variables, le nombre de contraintes de capacités et le nombre de contraintes de demandes. Ce pointeur pointe également vers la matrice d'entiers « a ». Les lignes de la matrice correspondent aux contraintes, et les colonnes aux variables. Ainsi dans la case a[3][10] on va retrouver la valeur de la $10^{ème}$ variable pour la $3^{ème}$ contrainte. En a[0][x] on retrouve l'apport de la variable x pour la fonction objectif. Enfin dans cette matrice, on différencie les contraintes de capacités des contraintes de demandes grâce à l'index. Ainsi les contraintes a[1][x] à a[mkp->cc][x] correspondent à des contraintes de capacités et celles qui suivent à des contraintes de demandes.

Pour la représentation de la solution nous avons également utilisée la représentation donnée en TD, nous avons simplement fait de « slack » un tableau à 2 dimensions pour y ajouter la slack des contraintes de demandes.

Pour récapitulatif, une solution est donc représentée par un pointeur (sol), pointant vers un entier (objValue), un tableau à 1 dimension d'entiers (x) et un tableau à 2 dimensions d'entiers (slack). objValue correspond au résultat de la fonction objectif, x correspond à ce qu'on va retrouver dans le sac. Ainsi x a pour taille le nombre de variables + 1, dans x[0] on a le nombre d'objet mis dans le sac et x[n] vaut 1 si la nième variable est dans le sac et 0 sinon. Enfin slack correspond à ce qui reste pour les contraintes. slack[0] correspond aux contraintes de capacités (taille = nombre de contraintes de

capacités + 1) et slack[1] correspond aux contraintes de demandes (taille = nombre de contraintes de demandes + 1). Dans slack[0[0] et slack[1][0] on retrouve le nombre de contraintes non respectées pour la solution en cours et dans slack[0][n] et slack[1][n'] on retrouve ce qu'il nous reste pour la nième et n'ième contraintes. Exemple pour une contrainte de poids (la 2ème contrainte), si on a déjà mis 10 alors qu'on peut mettre en tout 15, alors on va mettre 5 dans slack[0][2]. Ainsi une solution réalisable est une solution pour laquelle slack[0][0] et slack[1][0] valent toutes les deux 0.

2. Méthode de résolution

Pour résoudre ce problème, nous avons procédé par étapes, nous avons d'abord essayé d'avoir une solution réalisable puis petit à petit nous avons essayé d'améliorer notre solution grâce à différents algorithme.

a. Obtenir la solution réalisable

Afin d'obtenir cette solution réalisable nous avons dans un premier temps rempli tout le sac, afin de satisfaire toutes les contraintes de demandes. Puis nous avons retiré petit à petit des objets (tout en respectant les contraintes de demandes) jusqu'à ce que les contraintes de capacités soient elles aussi respectées. Ainsi nous avons rapidement eu une solution réalisable.

Pour choisir l'ordre des objets à retirer nous avons utilisés différents algorithme pour trier les objets en fonction d'un certains coefficients permettant d'obtenir la meilleure solution possible dès le début. Ce coefficient peut donc dépendre du poids de l'objet, de ce qu'il rapport pour la fonction objectif et de sa demande. Nous avons calculé différents coefficients afin de voir ce qu'il fonctionnait le mieux et bien que certains algorithme nous ont donnés des solutions initiales plus intéressantes, à la fin on n'obtenait pas forcément la meilleure solution. De ce fait nous avons gardé tous nos algorithmes.

Voici les différents calculs des coefficients (que vous pourrez retrouver dans le code) :

- getTableauOrdonneByCoeff : Coefficient obtenu en divisant ce que rapporte l'objet par son poids.

- getTableauOrdonneByCoeffDemandeSurPoids : Coefficient obtenu en divisant ce que rapporte l'objet en termes de demande par son poids

- getTableauOrdonneByCoeffPoidSurDemandePlusValeur : Coefficient obtenu en divisant ce que rapporte l'objet dans la fonction objectif et en demande par son poids (dans le code c'est l'inverse mais le trie après est aussi inversé ce qui revient au même)

- getTableauOrdonneByLessDemand : Coefficient obtenu en fonction de ce que rapporte l'objet en termes de demande (on enlève en premier celui qui rapporte le moins en termes de demande, dans l'optique d'enlever le plus d'objets possibles tout en satisfaisant les contraintes de demandes)

- getTableauOrdonneByIndex : Coefficient obtenu tout simplement par l'index de l'objet (on enlève en premier le premier objet de la liste, utile pour avoir une autre solution initiale mais aucune d'avoir une bonne solution initiale dès le début)
- getTableauOrdonneByReverseIndex : Coefficient obtenu tout simplement par l'index de l'objet (on enlève en premier le dernier objet de la liste, utile pour avoir une autre solution initiale mais aucune chance d'avoir une bonne solution initiale dès le début)
- getTableauOrdonneRandom : Coefficient obtenu aléatoirement (utile pour avoir une autre solution initiale mais aucune chance d'avoir une bonne solution initiale dès le début)

Une fois la solution initiale réalisable obtenue, nous avons essayé de l'améliorer.

b. Première amélioration : parcours des voisins

Afin d'améliorer la solution nous avons d'abord essayé d'enlever un objet et d'en ajouter un autre afin d'obtenir un meilleur résultat et ce jusqu'à ce qu'on ne puisse plus améliorer la solution. C'est notre parcours des voisins de la solution. Nous avions 2 possibilités, la première, faire le premier mouvement rencontré qui améliore la solution afin de ne pas avoir à parcourir tous les objets. La deuxième parcourir tous les objets afin d'avoir tous les mouvements améliorants et de réaliser le meilleur d'entre eux.

Ne sachant pas quelle façon de faire aller être la meilleure nous avons développé les 2. Au final (comme vous pourrez le voir dans la partie résultat) il n'y a pas vraiment de meilleure solution parmi ces deux-là même si a priori le parcours de tous les voisins semblent être meilleur en termes de résultat, nous avons donc laissé dans le code finale les 2 solutions et vous pouvez choisir laquelle utiliser en changeant à la ligne 667 le 3ème argument de la méthode parcoursVoisin (0 => on prend le premier mouvement améliorant rencontré, 1 => on parcours tous les voisins, par défaut nous avons mis 1. A la base c'était paramétrable avec un argument mais nous avons préféré suivre ce qu'il y avait dans le sujet par rapport aux arguments et nous l'avons donc retiré).

Grâce à cet algorithme nous avons pu améliorer notre solution mais assez rapidement, on s'est retrouvé bloqué car plus de mouvement améliorant, nous avons donc mis en place l'algorithme tabou pour continuer d'améliorer la solution.

c. Deuxième amélioration : algorithme tabou

Pour information, tout ce qui suit a été codé dans les deux méthodes citées au-dessus (celle avec le parcours de tous les voisins et celle qui prend le premier mouvement améliorant).

Afin de se débloquer une fois le moment où aucun mouvement n'améliore la solution il nous faut quelque chose nous permettant de dégrader la solution pour voir s'il n'y a pas mieux derrière, c'est donc à ça que sert notre algorithme tabou. C'est très simple, lorsqu'il n'y a plus de mouvement améliorant on va continuer en effectuant le mouvement le moins dégradant et refaire un parcours des nouveaux voisins, ainsi durant le parcours des voisins, on va sauvegarder quel mouvement n'est pas améliorant mais qui dégrade le moins. Si on a tout parcouru sans trouver de meilleurs mouvements alors on va effectuer ce mouvement sauvegardée. Le problème est qu'à l'étape d'après on va

rechercher les mouvements améliorants et il est très probables qu'on tombe sur le mouvement inverse qu'on vient d'effectuer pour dégrader la solution. Hors en faisant ce mouvement on va revenir en arrière est on ne va pas avancer puisqu'on va boucler. Pour résoudre ce problème nous avons donc mis en place une liste taboue.

Cette liste tabou va contenir des structures qui vont représenter un mouvement (donc 2 entiers représentants les indices des objets échangés) et la valeur de la fonction objectif de la solution juste avant la réalisation du mouvement dégradant. Ainsi, juste avant de dégrader la solution on va sauvegarder le mouvement inverse qu'on va effectuer pour dégrader la solution et dans les prochains parcours de voisins, même si on voit que ce mouvement améliore la solution et bien on va l'interdire pour justement éviter de boucler !

La sauvegarde de la valeur de la fonction objectif permet également d'éviter de boucler en effet on suppose que si en faisant un mouvement (différent des mouvements présents dans la liste tabou) on obtient une solution ayant la même valeur en fonction objectif qu'une des valeurs présentent dans la liste tabou alors on ne va pas faire le mouvement car en le faisant on va sûrement revenir à une solution déjà parcourue. Il est possible que ça soit 2 solutions différents avec les mêmes valeurs en fonction objectif mais pour ne pas sauvegarder toute une solution et faire à chaque fois une comparaison on a considéré que si la valeur de la fonction objectif est la même pour deux solutions alors elles sont égales et on ne veut pas revenir sur une solution déjà parcourue sinon nous allons boucler d'où cette interdiction.

Enfin au bout d'un certains moment on va pouvoir refaire les mouvements tabou qu'on avait fait il y a longtemps pour éviter au maximum d'oublier des solutions. Pour ce faire, la liste tabou se comporte comme une file FIFO (ici de taille 15), ainsi dès qu'il y a 15 mouvements tabous, au 16ème mouvement, le premier mouvement tabou ne sera plus tabou et on pourra donc le refaire etc etc.

Grâce à cet algorithme tabou nous avons pu continuer d'améliorer notre solution en prenant la meilleure solution trouvée parmi tous les parcours de voisins. L'algorithme tabou ne s'arrête jamais dû à la taille fixe de la liste taboue nous avons donc forcé l'arrêt de l'algorithme après avoir fait 3000 itérations sans améliorer la solution (si au bout de 2000 itérations on améliore alors le conteur revient à 0). Nous avons essayé avec 5000 itérations mais les résultats étaient les mêmes qu'avec 3000, pour nous ça ne sert donc à rien de faire 2000 itérations de plus.

Il ne nous reste plus qu'à utiliser tout le temps que nous avons pour trouver la meilleure solution possible.

d. Exploitation du temps imparti

Afin d'exploiter entièrement le temps qu'on nous donne en argument, une fois que notre algorithme se termine pour une solution initiale, on le refait mais avec une autre solution initiale qui pourra potentiellement nous amener à une meilleure solution. Les différentes manières d'obtenir des solutions initiales présentées juste au-dessus ont donc toutes été utilisées afin d'avoir à chaque fois une nouvelle solution initiale. A chaque exécution du programme l'ordre des solutions initiales générées est le même :

- getTableauOrdonneByCoeff
- getTableauOrdonneByCoeffDemandeSurPoids
- getTableauOrdonneByCoeffPoidSurDemandePlusValeur

- getTableauOrdonneByLessDemand
- getTableauOrdonneByIndex
- getTableauOrdonneByReverseIndex
- getTableauOrdonneRandom

Si on arrive à la fin de ce cycle on continue de générer des solutions avec getTableauOrdonneRandom puisque cette méthode nous donne à chaque fois une solution initiale différente (aléatoire).

Grâce à cela on utilise la totalité du temps donné pour trouver la meilleure des solutions en utilisant une multitude de solution initiales (la seule chose statique est donc le fait de parcourir tous les voisins ou de prendre le premier meilleur voisin).

3. Difficultés

Tout au long de ce mini-projet nous avons eu quelques difficultés, d'abord pour bien comprendre le problème et les différentes méthodes de résolution mais surtout les fuites mémoires. En effet quand on faisait beaucoup d'appels récursifs, le programme plantait. Nous avons donc dû chercher nos fuites mémoires et une fois une très grandes parties des fuites corrigées (peut-être qu'il en reste), le programme planté quand même avec un grand nombre de récursivités. Nous avons donc passé pas mal de temps pour chercher à résoudre ces problèmes et au final après des recherches on a vu qu'on était limité en nombre de récursivités possibles. Et que juste le code suivant plantait lui aussi tout seul dû à cette limitation :

```
int testRecursivite(int i, int cpt) {
   if(cpt < 500000)
      return testRecursivite(i, cpt++);
   else
      return i;
}
testRecursivite(0, 0);
```

C'est donc normal qu'au bout de beaucoup d'appels récursifs le programme plante et cela ne gênait pas pour notre programme puisqu'avec 3000 itérations pour l'algorithme tabou ça passe largement (on plante vers 10000 itérations pour l'algorithme tabou)
La solution pour résoudre ce problème serait de transformer notre fonction de parcours des voisins en fonction itérative plutôt qu'en fonction récursive. Mais nous ne l'avons pas fait car il fallait tout remodifier (alors que pour nous ça fonctionne bien) et nous n'avons pas eu le temps.

Pour information : pour lancer le programme référez-vous au fichier readmeCompilationExcecution.

Résultats

Pour tous nos résultats, nous avons utilisé un ordinateur avec 6Go de RAM, un processeur Intel Core i7 à 2,40GHz sous Windows 10.

Afin de vous montrer les meilleurs résultats possibles, pour chaque instance nous avons lancé notre programme plusieurs fois en partant de solution initiale différente, ainsi nous pourrons spécifier à chaque fois avec quelle solution initiale nous avons abouti à la meilleure de nos solutions.
Les solutions initiales ont été obtenues en remplissant le sac (voir méthode de résolution) suivant différentes manières de tri qui peuvent être retrouvées dans le code.

Ce comportement est donc différent du programme final que nous vous avons transmis puisque ce dernier utilise au maximum le temps qu'il a. Donc au sein d'une seule exécution nous utilisons plusieurs solutions initiales afin de trouver la meilleure solution possible en une exécution dans le temps imparti.

1. Instance 100Md5_1_1pos.txt

Meilleure solution : **33018** en améliorant une solution initiale basée sur « ByCoeffDemandeSurPoids » (donnant une solution initiale à 14694) et en utilisant l'algorithme tabou en parcourant tous les voisins (en prenant le premier meilleur voisin : 32961).

2. Instance 100Md5_1_2pos.txt

Meilleure solution : **29073** en améliorant une solution initiale basée sur « ByCoeffDemandeSurPoids » (donnant une solution initiale à 14681) et en utilisant l'algorithme tabou en prenant le premier meilleur voisin (même résultat en parcourant tous les voisins). Egalement obtenu avec une solution initiale aléatoire.

3. Instance 100Md5_1_5pos.txt

Meilleure solution : **22130** en améliorant une solution initiale basée sur « ByCoeffDemandeSurPoids » (donnant une solution initiale à 13206) et en utilisant l'algorithme tabou en parcourant tous les voisins (en prenant le meilleur voisin : 22051).

4. Instance 100Md5_2_1pos.txt

Meilleure solution : **30644** en améliorant une solution initiale basée sur « ByCoeff » (donnant une solution initiale à 29477) et en utilisant l'algorithme tabou en prenant le premier meilleur voisin (même résultat en parcourant tous les voisins).

5. Instance 100Md5_2_2pos.txt

Meilleure solution : **27879** en améliorant une solution initiale basée sur « ByCoeffPoidsSurDemandePlusValeur » (donnant une solution initiale à 15516) et en utilisant l'algorithme tabou en prenant le premier meilleur voisin (en parcourant tous les voisins : 27829).

6. Instance 100Md5_2_5pos.txt

Meilleure solution : **26236** en améliorant une solution initiale basée sur « ByCoeffDemandeSurPoids » (donnant une solution initiale à 16820) et en utilisant l'algorithme tabou en prenant le premier meilleur voisin (en parcourant tous les voisins : 26139).

7. Instance 250Md5_1_1pos.txt

Meilleure solution : **90085** en améliorant une solution initiale basée sur « ByCoeff » (donnant une solution initiale à 87910) et en utilisant l'algorithme tabou en prenant le premier meilleur voisin (en parcourant tous les voisins : 90073).

8. Instance 250Md5_1_2pos.txt

Meilleure solution : **79906** en améliorant une solution initiale basée sur « ByCoeffPoidsSurDemandePlusValeur » (donnant une solution initiale à 56501) et en utilisant l'algorithme tabou en parcourant tous les voisins (en prenant le meilleur voisin : 79799).

9. Instance 250Md5_1_5pos.txt

Meilleure solution : **68110** en améliorant une solution initiale basée sur « ByCoeffDemandeSurPoids » (donnant une solution initiale à 46894) et en utilisant l'algorithme tabou en parcourant tous les voisins (en prenant le meilleur voisin : 68075).

10. Instance 250Md5_2_1pos.txt

Meilleure solution : **82821** en améliorant une solution initiale basée sur « ByCoeffPoidsSurDemandePlusValeur » (donnant une solution initiale à 59622) et en utilisant l'algorithme tabou en parcourant tous les voisins (en prenant le meilleur voisin : 82664).

11. Instance 250Md5_2_2pos.txt

Meilleure solution : **76835** en améliorant une solution initiale basée sur « ByCoeff » (donnant une solution initiale à 75010) et en utilisant l'algorithme tabou en parcourant tous les voisins (en prenant le meilleur voisin : 76878).

12. Instance 250Md5_2_5pos.txt

Meilleure solution : **61651** en améliorant une solution initiale basée sur « ByCoeffPoidsSurDemandePlusValeur » (donnant une solution initiale à 37302) et en utilisant l'algorithme tabou en prenant le meilleur voisin (en parcourant tous les voisins : 61647).

13. Instance 500Md5_1_1pos.txt

Meilleure solution : **177554** en améliorant une solution initiale basée sur « ByCoeffDemandeSurPoids » (donnant une solution initiale à 82784) et en utilisant l'algorithme tabou en parcourant tous les voisins (en prenant le meilleur voisin : 177394).

14. Instance 500Md5_1_2pos.txt

Meilleure solution : **146114** en améliorant une solution initiale basée sur « ByCoeffDemandeSurPoids » (donnant une solution initiale à 81643) et en utilisant l'algorithme tabou en parcourant tous les voisins (en prenant le meilleur voisin : 146043).

15. Instance 500Md5_1_5pos.txt

Meilleure solution : **136437** en améliorant une solution initiale basée sur « ByCoeff » (donnant une solution initiale à 128898) et en utilisant l'algorithme tabou en parcourant tous les voisins (en prenant le meilleur voisin : 136415).

16. Instance 500Md5_2_1pos.txt

Meilleure solution : **180886** en améliorant une solution initiale basée sur « ByCoeff » (donnant une solution initiale à 175415) et en utilisant l'algorithme tabou en prenant le meilleur voisin (en parcourant tous les voisins : 180839).

17. Instance 500Md5_2_2pos.txt

Meilleure solution : **161938** en améliorant une solution initiale basée sur « ByCoeffDemandeSurPoids » (donnant une solution initiale à 96009) et en utilisant l'algorithme tabou en prenant le meilleur voisin (en parcourant tous les voisins : 161850).

18. Instance 500Md5_2_5pos.txt

Meilleure solution : **145044** en améliorant une solution initiale basée sur « ByCoeff » (donnant une solution initiale à 141980) et en utilisant l'algorithme tabou en parcourant tous les voisins (en prenant le meilleur voisin : 145018).

Organisation au sein de l'équipe

Au niveau de l'organisation, chaque membre du groupe a contribué à la réalisation de ce mini-projet.

Maxime DEGRES :

- Mise en place des structures de données
- Instanciation du problème via le fichier donné en argument
- Développement de la méthode pour parcourir tous les voisins
- Recherche et correction de fuites mémoires

Jean-Baptiste DURIEZ :

- Développement des algorithmes pour obtenir une solution initiale
- Développement de la méthode qui prend le premier voisin améliorant
- Mise en place de l'algorithme tabou
- Recherche et correction de fuites mémoires
- Mise en place finale de l'utilisation du temps
- Récupération et exploitation des résultats
- Réalisation du Makefile (pour la compilation)
- Rédaction du rapport

Jordane QUINCY

- Développement des algorithmes pour obtenir une solution initiale
- Développement de la méthode pour parcourir tous les voisins
- Grande recherche et correction de fuites mémoires (utilisation d'outil spécifique)
- Début mise en place de l'utilisation du temps
- Ecriture de la solution trouvée dans un fichier

Conclusion

Au final, ce mini-projet nous a permis d'apprendre l'utilisation des heuristiques et méta-heuristiques. On en sait également plus sur les fuites mémoires et sur les outils pour aider à les corriger. Enfin, si on se fie à nos résultats, on voient bien que ce n'est pas parce que notre solution initiale est très bonne qu'on obtient au final une meilleure solution en faisant le parcours des voisins avec l'algorithme tabou, à l'inverse une solution initiale moins bonne nous a souvent permis de trouver à la fin une meilleure solution ! De même le parcours de tous les voisins est souvent meilleur que de prendre le premier voisin améliorant mais de très peu au final, pour des problèmes avec un très grand nombre de variables, il vaut peut-être mieux utiliser uniquement la méthode en prenant le premier voisin améliorant qui est beaucoup moins gourmande en termes de temps d'exécution.