Université de Nantes — UFR Sciences et Techniques
Master informatique parcours “optimisation en recherche opérationnelle (ORO)”
Année académique 2018-2019

Dossier Devoirs Maison

Métaheursitiques

Marie Humbert–Ropers[1] – Arthur Gontier[2]

28 septembre 2018

Livrable du devoir maison 1 :
Heuristiques de construction et d'amélioration gloutonnes

Formulation du SPP

Le Set packing problem est un problème d'optimisation combinatoire NP-complet. Il se modélise par une fonction objectif à maximiser et qui se définit par la somme de variables de décisions binaires multipliées par leurs coûts respectifs. Chaque contrainte concerne un sous-ensemble de variables et ne permet qu'à une seule de ces variables d'être mise à un. Cela empêche donc plusieurs variables de décisions d'être utilisées en même temps. L'exemple d'application le plus connu de ce problème est le problème du chef cuisinier qui cherche à maximiser le nombre de recettes à préparer, en fonction des ingrédients disponibles. Les variables correspondent aux recettes possibles qui ont besoin de plusieurs ingrédients et les ingrédients (contraintes) ne peuvent être utilisés que pour une seule recette. Les coûts liés aux variables de décisions correspondent à la popularité de la recette. De cette manière, on souhaite faire le plus et les meilleures recettes possibles avec les ingrédients à disposition.

Le Set Packing Problem peut-être aussi illustré par la situation suivante : Après un dernier cours conclut par un bon goûter avec ses étudiants, un enseignant s'attelle à la rédaction d'un contrôle pour eux. Dans ce but, il a précédemment créé une banque d'exercices et il se décide à piocher dans ceux-ci. On note que chaque exercice est composé de différents types de questions. Son objectif est de choisir les exercices pour le contrôle en optimisant la difficulté et en maximisant la diversité de questions. Certains exercices lui semble plus importants, et pédagogiquement plus intéressant et complexe que d'autres, donc des coûts plus élevés leurs sont affublés. Comme il souhaite interroger ses élèves sur un champ de connaissances et de compétences le plus large possible, il estime qu'il est inutile d'avoir des questions redondantes entre deux exercices. Il pose donc comme contrainte qu'un type de question n'apparaisse au maximum qu'une seule fois dans le contrôle. Ainsi, si plusieurs exercices ont une question en commun, seul l'un d'entre eux pourra être sélectionné.

Modélisation JuMP (ou GMP) du SPP

Présentation de la modélisation via JuMP du SPP.

Les variables bianires modélisent la décision de prendre ou non un exercice dans le contrôle.

```
1  @variable(m,x[1:nb] >= 0,Bin)
```

L'objectif est de maximiser le nombre de questions tout en prenant en compte la difficulté de ces questions.

```
1  @objective(m,Max,sum(x[i]*C[i] for i in 1:nb))
```

Les exercices avec des questions similaires sont à l'origine d' une contrainte : la somme des variables représentant ces exercices doit être infèrieure ou égale à 1.

```
1  @constraint(m,Ingredients[icontr = 1:nb_constr], sum(x[ivars]*A[icontr,ivars] for ivars in 1:
       nb) <= 1)
```

Et enfin, on résoud le problème ainsi formulé avec le solveur GLPK MIP

Instances numériques de SPP

Le tableau suivant rassemble 10 instances avec leur nombre de variables et de contraintes :

nom de l'instance	nombre de contraintes	nombres de variables
pb1000rnd0300	5000	1000
pb1000rnd0700	1000	1000
pb100rnd0500	100	100
pb2000rnd0700	2000	2000
pb200rnd0100	1000	200
pb200rnd0300	1000	200
pb200rnd0700	200	200
pb200rnd0900	200	200
pb500rnd0700	500	500
pb500rnd0900	500	500

Heuristique de construction appliquée au SPP

Nous avons essayé plusieurs implémentations de l'algorithme du glouton pour améliorer le compromis entre la qualité du résultat et sa rapidité d'exécution. Un premier glouton ajoute à la solution les premières variables auxquelles il accède et qui sont compatibles avec les contraintes. La deuxième possibilité est de faire en sorte que l'algorithme choisisse de manière judicieuse les variables ajoutées à la solution. Pour ce faire, le ratio entre le coût d'une variable et le nombre d'apparitions dans les contraintes est calculé. En calculant ce ratio, les variables apparaissant peu dans les contraintes ou avec un grand coût sont favorisées.

Afin d'optimiser les temps d'exécutions, nous avons pris en compte différentes structures de données, ainsi que plusieurs façons de les manipuler. Nous avons implémenté les gloutons avec les méthodes suivantes afin de les comparer :

- Le premier se base sur une matrice et supprime les colonnes des variables utilisées et les lignes des contraintes satisfaites. Cette heuristique est très lente, à cause de la compléxité de l'opération de suppression. Elle est donc peu intéressante.
- Le deuxième commence par une copie de la matrice puis plutôt que de supprimer les lignes ou colonnes, il les met à 0 pour qu'elle ne soient plus prises en compte dans le calcul des ratio. Malgrès cela, le calcul des poids dans la matrice reste très lourd.
- Le dernier se base sur des vecteurs de vecteurs. On en utilise deux : un qui est d'abord indexé par les variables et dont chaque case contient un vecteur des indices des contraintes dans lesquelles la variable associée apparaît. Et un deuxième qui est indexé par les contraintes et qui, de manière symétrique, contient les indices des variables qui apparaissent dans la contrainte. De cette manière, on se rapproche du concept de matrice creuse, ce qui permet aux recherches d'informations d'être plus rapides. Comme dit précédemment, avant d'ajouter une nouvelle variable à la solution, on recherche la variable qui à le plus grand coût et qui apparaît dans le moins de contraintes pour la favoriser. Ceci ralentit très légèrement l'heuristique mais permet d'avoir une solution initiale plus intéressante pour les heuristiques d'améliorations.

Exemple 1 *L'algorithme le plus rapide est celui manipulant les vecteurs de vecteurs, que nous présentons en exemple ci-dessous :*

La première démarche est d'initialiser un vecteur solution avec des 0. Dans un second temps, on calcule dans un vecteur les ratios des coûts sur le nombre de contraintes où une variable est présente. Le choix des variables se fait selon ce ratio et dès que les contraintes le permettent la variable est à ajouter à la solution. Dans l'exemple du fichier didactic.dat, la variable avec le plus grand ratio est la variable 6. Toutes les variables dans les mêmes contraintes que la variable 6 ne sont plus prises en comptes et sont mises à 0. Les ratios sont recalculés : la variable 7 devient la plus intéressante et de la même manière, la variable 7 est mise à 1 dans la solution, le nombre de contraintes et de variables non utilisées diminuent. Le tri est refait. 4 est la prochaine variable. Après la 4, il ne reste plus de contraintes non fixée. Une solution de base est trouvée.

Heuristique d'amélioration appliquée au SPP

Dans l'heuristique d'amélioration, nous utilisons le mouvement de kp-échange sur deux voisinages. Nous avons là aussi implémenté plusieurs approches du k-p échange. Avec les résultats des heuristiques de construction (présentés dans la partie suivante), nous avons conclu que, pour la rapidité d'exécution, il était préférable d'utiliser les vecteurs de vecteurs comme structures de données pour les approches suivantes :

— Notre première approche enchaîne tous les voisinages 2-1 puis tous les 1-1 et enfin les 0-1. Elle choisit les variables entrantes dans un tableau trié en fonction des coûts des variables et elle commence par les coûts les plus importants. De cette manière, on obtient un algorithme de recherche locale en simple descente qui favorise les variables avec les coûts réduits les plus intéressants.
De petites modifications permettent de transformer cet algorithme en plus profonde descente. Il suffit de rechercher dans tous les voisinages en gardant la solution la plus intéressante au lieu de s'arrêter au premier voisin améliorant. Comme avec les algorithmes gloutons construisant une solution admissible, la structure de données que l'on choisit d'utiliser influence beaucoup la qualité des résultats. En effet, une première idée d'implémentation de l'heuristique de recherche locale parcourait toutes les variables pour trouver les combinaisons d'un k-p échange, alors qu'il suffit de vérifier l'espace des contraintes affectées par la variable que l'on souhaite mettre à un. Et il ne faut considérer que les variables qui sont liées à cet espace pour choisir celles que l'on veut retirer. D'où le choix des vecteurs de vecteurs qui permettent une vérification plus rapide.

— Une deuxième version pourrait être de choisir en fonction de la situation différents k et p. De cette manière, il serait possible d'obtenir une solution plus efficace que l'enchainement brutal présenté ci-dessus. Néanmoins, nos implémentations de ce genre d'algorithme n'ont pas été très concluantes, et sont plus lentes que la première ci-dessus.

Exemple 2 *Prenons l'exemple du fichier de test de taille 9. Si l'on part d'une solution où seules les variables 2 et 4 sont à 1. Afin d'améliorer la solution, l'algorithme envisage la possibilité du 2-1 échange. En utlisant les ratios triés, il y a une tentative de 2-1 mais il n'y a pas de solution améliorante. Ensuite, on passe au 1-1 échange : la variable 6 devient candidate pour être ajoutée à la solution en enlevant 2. Après cela, aucune autre variable ne permet un 1-1 échange améliorant. Puis, on poursuit par un 0-1 échange qui permet de rajouter 7 à la solution. Enfin, l'algorithme ne peut plus améliorer la solution et s'arrête, il a trouvé un optimum local.*

Expérimentation numérique

L'ensemble des programmes ont été réalisé en julia 0.6.4 et les tests ont été effectués avec un pc acer sous ubuntu18 avec les caractéristiques suivantes :

— RAM : 4 Gio
— CPU : Intel® Core™ i3-7130U CPU @ 2.70GHz × 4
— GPU : Intel® HD Graphics 620 (Kaby Lake GT2)

Pour les différentes versions de l'algorithme glouton, nous obtenons les résultats suivants :

type de glouton	simple		trié		supr matrice		manip matrice		manip vecteurs	
nom de l'instance	temps /s	z	temps /s	z	temps /s	z	temps /s	z	temps /s	z
pb1000rnd0300	0.008775	351	0.010590	504	1.850464	507	0.130910	507	0.002170	507
pb1000rnd0700	0.003924	1229	0.005638	1878	1.733921	2050	0.091845	2050	0.006371	2050
pb100rnd0500	0.000021	533	0.000038	623	0.010806	621	0.000270	621	0.000148	621
pb2000rnd0700	0.019139	866	0.026417	1310	6.245873	1524	0.259427	1524	0.013657	1524
pb200rnd0100	0.000324	215	0.000251	348	0.042346	351	0.007225	351	0.000250	351
pb200rnd0300	0.003471	424	0.000508	616	0.093298	682	0.003796	682	0.000434	682
pb200rnd0700	0.000061	702	0.000158	822	0.026194	945	0.001321	945	0.000659	945
pb200rnd0900	0.000092	1015	0.000123	1267	0.033477	1279	0.001330	1279	0.000489	1279
pb500rnd0700	0.002245	667	0.000716	936	0.145437	1043	0.006458	1043	0.001288	1043
pb500rnd0900	0.001329	1527	0.004547	2032	0.282291	2163	0.011376	2163	0.002403	2163

On note par la suite gl1, le glouton le plus simple et gl2, le glouton sur des variables favorisées par leurs poids.

Pour les différentes versions des algorithmes de recherches locales, nous obtenons les résultats suivants :

type de descente	simple/gl1		profonde/gl1		profonde/gl2		profonde eco/gl2	
nom de l'instance	temps /s	z	temps /s	z	temps /s	z	temps /s	z
pb1000rnd0300	0.069109	528	0.04641	416	0.025537	525	0.054189	525
pb1000rnd0700	0.157854	2077	0.020181	1692	0.021901	2082	0.012117	2098
pb100rnd0500	0.002202	621	0.002334	596	0.001129	626	0.000928	626
pb2000rnd0700	0.303181	1515	0.067378	1141	0.049690	1552	0.025904	1562
pb200rnd0100	0.002710	375	0.001966	292	0.004903	359	0.001333	366
pb200rnd0300	0.010585	677	0.002225	551	0.002173	689	0.001336	689
pb200rnd0700	0.004580	937	0.004385	849	0.001720	955	0.001014	955
pb200rnd0900	0.015460	1276	0.005575	1180	0.002598	1309	0.001561	1318
pb500rnd0700	0.014237	1008	0.014051	792	0.004580	1065	0.002500	1103
pb500rnd0900	0.078460	2143	0.010796	1880	0.006794	2181	0.002367	2181

Les algorithmes de recherches locales en profonde descente sont plus représentés puisque plusieurs essais de simple descente avec un tri donnent de relativement bons temps uniquement sur certaines instances mais de très mauvais temps sur d'autres. Nous avons donc ensuite préféré la constance des temps et la qualité des résultats de la plus profonde descente.

Discussion

— Après l'observation de nos résultats, nous remarquons que dès que l'instance augmente en taille, à partir de 100 variables, la résolution exacte prend au moins plusieurs minutes, pour la plupart des instances (cf : resultats-simplex.txt). La différence de temps pour aboutir à une solution, optimale ou non, entre les deux méthodes est très creusée (moins d'une seconde pour les heuristiques contre plusieurs minutes minimum pour des résolutions exactes sur de grandes instances). Les solutions trouvées ne sont pas optimales mais approche l'optimalité, notamment avec la plus profonde descente. Les heuristiques paraissent donc compétitives pour proposer des résultats rapides pour les problèmes de grandes tailles.

— Le recours aux métaheuristiques semble donc très prometteur car les temps de calculs sont très intéressants. Néanmoins, ces heuristiques comportent plusieurs défaults. En premier lieu, ce n'est intéressant que si l'on peut faire un compromis sur l'optimalité. Ces solutions ne sont pas exactes, mais en plus, il n'y a pas de garantie quant à leurs qualités. L'argumentation pour la validité et la comparaison des différentes heuristiques nous semble difficilement démontrable et il existe souvent une forme des données défavorable à l'heuristique choisie.

— On remarque aussi que le glouton intelligent (qui prend en compte le ratio coût/(nombre d'apparition dans les contraintes)) donne une bien meilleure solution initiale que le basique. A tel point que sur plusieurs instances, les heuristiques d'améliorations n'arrivent pas à améliorer cette solution. Il est probable que ce glouton trouve directement un optimum local dont les descentes ne peuvent pas sortir. Une heuristique ayant la capacité de sortir des optimums locaux serait plus adaptée, tels que le Tabu Search, Recuit Simulé...

— Les (méta)heuristiques sont prometteuses puisque pour une démarche gloutonne, il est possible de trouver des solutions très rapidement. Il est donc possible de s'imaginer que des méthodes plus réfléchies ou bien la combinaison de plusieurs méthodes adaptées à la forme des données puissent permettre de se rapprocher de l'optimum globale.