Université de Nantes - UFR Sciences et Techniques

Année 2017-2018 — M1 ORO - M1 DS

Métaheursitiques

Livrable DM1 (distanciel)

Romain BERNARD

Formulation du SPP

Le SPP ou Set Packing Problem est un problème linéaire en maximisation s'écrivant :

$$\begin{aligned} \max \quad & \sum_{i=1}^{n} (C_i * x_i) \\ \text{s.c.} \quad & \sum_{i=1}^{n} t_{i,j} * x_i \leq 1 \quad \forall j \in J \\ & x_i \in \quad \{0,1\} \quad \forall i \in I \end{aligned}$$

Ce type de problème peut être applicable dans la réalité à la réalisation de produits finis à partir de sets de ressources de bases limités tout en maximisant le profit.
Imaginons un menuisier qui a besoin de certains sets de planches pour réaliser des meubles. Un même set peut servir pour la construction de différents meubles et un meuble est réalisable grâce à au moins un des sets. Si le menuisier veut optimiser le profit généré par ses ressources limitées, on est face à un SPP.

Modélisation JuMP (ou GMP) du SPP

```
# Setting an ip model of SPP
function setSPP(solverSelected, C, A)
  m, n = size(A)
  ip = Model(solver=solverSelected)
  @variable(ip, x[1:n], Bin)
  @objective(ip, Max, dot(C, x))
  @constraint(ip , cte[i=1:m], sum(A[i,j] * x[j] for j=1:n) <= 1)
  return ip, x
end
```

Dans un premier temps, on assigne à une variable notre modèle et son solveur grâce au constructeur Model. Ensuite, l'instruction @variable permet de déclarer nos variables grâce à la notation x[1 :n] qui permet de faire se répéter l'instruction sans écrire de boucle itérant les variables et répétant l'instruction. C'est aussi à ce niveau qu'on définit les variables comme étant binaires ("Bin" dans le code)

L'instruction @objective permet de déclarer la fonction objectif qui est en maximisation. On déclare les valeurs de la fonction via la fonction dot qui multiplie les vecteurs C et x valeur par valeur (une notation équivalente aurait été C .* x).

Enfin, l'instruction @constraint défini nos contraintes. Pour ce faire, on exécute l'instruction pour chaque contrainte récupérée de notre fichier (de 1 à m) puis on forme notre contrainte en multipliant le coefficient trouvé dans la matrice de contraintes à la variable, donnant nos contraintes de la forme $\sum_{i=1}^{n} x_i \leq 1$ à l'aide de la matrice donnée par le fichier.

Instances numériques de SPP

Nom	Nb Variables	Nb contraintes	Densité	Max-Uns	Optimum
didactic	9	7	n/a	n/a	30
pb_100rnd0100	100	500	2.00	2	372
pb_100rnd0300	100	500	2.96	4	203
pb_200rnd0100	200	1000	1.49	4	416
pb_200rnd0600	200	1000	2.49	8	14
pb_200rnd0900	200	200	1.00	2	1324
pb_200rnd1600	200	600	1.00	2	79
pb_500rnd0100	500	2500	1.23	10	323
pb_500rnd1000	500	500	0.70	5	179
pb_1000rnd0100	1000	5000	2.60	50	67
pb_2000rnd0100	2000	10000	2.54	100	40

Heuristique de construction appliquée au SPP

Pour construire une solution de base à notre problème de base, on récupère l'instance à étudier dans le fichier. Ensuite, on utilise la formule d'utilité $\frac{C_i}{Count_i}$ où C est le coefficient de i dans la fonction objectif et Count le nombre de fois où i est présent dans les contraintes. Effectivement, moins une variable est présente dans les contraintes et plus elle rapporte d'argent, plus elle est intéressante.

On trie ensuite les variables selon leur utilité puis, si elles ne font pas partie d'une contrainte déjà saturée, on les ajoute à notre solution et ce, jusqu'à avoir étudié toute les variables. Quand on sature une condition, on va ajouter les variables concernées par cette condition à un ensemble permettant de connaître les variables inexploitables
Une fois ceci fait, on dispose de notre solution de base que l'on va chercher à améliorer à l'aide d'une heuristique de recherche locale.

Heuristique d'amélioration appliquée au SPP

A partir d'une solution x0, on va récupérer l'ensemble des solutions voisines à x0. C'est à dire l'ensemble des solutions atteignables via un k-p exchange. Dans notre cas, on va s'intéresser aux permutations 1-1 qui donnent un ensemble de nouvelles solutions intéressantes tout en restant de taille raisonnable.
Pour ce faire, on sélectionne tour à tour (via deux boucles) quelle valeur passer à 0 et quelle autre valeur passer à 1. Afin d'optimiser la génération des voisins, on passe au plus tôt les valeurs déjà à 0 (resp. à 1) quant on choisit quelle valeur passer à 0 (resp. à 1). Ensuite, on vérifie que le voisin généré ne viole aucune contrainte. Encore une fois, par soucis d'optimisation dès qu'une contrainte est violée, on quitte la boucle et on jette ce voisin.
Au cours de la construction de voisins, on ne conserve que la solution améliorant notre solution de base, et si un nouveau voisin est meilleur, il remplace le voisin stocké. Dans le cadre de ce DM, on effectue deux améliorations de la solution par k-p exchange avant d'estimer être satisfait de la solution.

Expérimentation numérique

La machine utilisée pour les tests dispose d'un SSD et d'un processeur intel core i5 à 3.5 GHz. Z représente la valeur optimale, Z_n représente la solution obtenue via les heuristiques après n exécution de la recherche locale.
Pour justifier l'utilisation d'heuristiques, j'ai aussi lancé une résolution via le solveur complet MIP de GLPK, certaines instances prenaient cependant énormément de temps et n'ont donc pas été jusqu'au bout. ces dernières sont alors notées ℵ

Nom	Z	Z_0	Z_2	Tps GLPK	Tps heuristique
didactic	30	30	30	0.000227s	0.1348s
pb_100rnd0100	372	342	343	1.4464s	0.0672s
pb_100rnd0300	203	193	194	0.5468s	0.0671s
pb_200rnd0100	416	351	354	87.18 min	0.6910s
pb_200rnd0600	14	11	11	85.587s	0.2214s
pb_200rnd0900	1324	1078	1114	0.0017s	0.4716s
pb_200rnd1600	79	70	70	8.42 h	0.9336s
pb_500rnd0100	323	267	267	ℵ	5.699s
pb_500rnd1000	179	151	151	ℵ	12.81s
pb_1000rnd0100	67	49	49	6 min	7.2756s
pb_2000rnd0100	40	37	37	61.7 min	50.68s

Discussion

Réponse aux questions suivantes :

— au regard des temps de résolution requis par le solveur MIP (GLPK) pour obtenir une solution optimale à l'instance considérée, l'usage d'une métaheuristique se justifie-t-il ?

Le tableau résumant l'expérimentation indique clairement que l'usage d'heuristiques est nécessaires à la résolution de problèmes à échelle réelle. Effectivement, le temps de résolution des solveurs complets grandit rapidement avec le nombre de contraintes et certaines instances pourtant petites peuvent faire tourner le solveur de nombreuses heures là où, malgré sa relative simplicité, notre heuristique donne des résultats pour la majorité acceptable tout en ne dépassant pas l'ordre de la minute pour résoudre un problème.

— avec pour référence la solution optimale, quelle est la qualité des solutions obtenues avec l'heuristique de construction et l'heuristique d'amélioration ?
Sur le plan des temps de résolution, quel est le rapport entre le temps consommé par le solveur MIP et vos heuristiques ?

Les solutions obtenues sont dans la majorité plutôt proches de l'optimum, malgré que certaines solutions soient un peu éloignées de l'optimum, comme les instances pb_500 mais dont le temps de résolution complet dépasse les 24h. On remarque cependant avec l'instance pb_200rnd0900 que dans certains cas particuliers, le solveur complet en plus de donner l'optimum est encore plus rapide que l'heuristique, mais ceci n'est pas une tendance générale.

— Le recours aux (méta)heuristiques apparaît-il prometteur ?
Entrevoyez-vous des pistes d'amélioration à apporter aux (méta)heuristiques ?

Le recours au (méta)heuristiques est plus que prometteur au vue des résultats sur des ensembles de "petite" taille (comparé à des problèmes à échelle réelle). Sur de tout petits ensembles, les solveurs sont assez optimisés pour surpasser les heuristiques mises en place en rapidité (dans la majorité des cas, car certaines instances comme pb_200rnd1600 font tourner le solveur très longtemps).
On remarque cependant que notre heuristique peut rester coincée dans des minima locaux. On pourrait donc l'améliorer en augmentant la portée du voisinage (simple mais rapidement coûteux) ou utiliser des mécanismes afin de s'échapper de ces minima locaux en acceptant de plus mauvaises solutions par moment.