Résolution de problèmes d'optimisation

1 Introduction

Jusqu'ici on a surtout vu des problèmes de *satisfaction*, par exemple de satisfaction de contraintes, où l'on cherchait une solution dans un espace de possibilités. Nous allons maintenant élargir ce cadre à des problèmes où l'on cherche non pas une solution mais la meilleure solution parmi les possibilités. On parlera d'optimisation combinatoire, ou d'optimisation sous contraintes. Il existe de nombreuses façons d'aborder ce type de problème, mais nous verrons un cadre générique de résolution qui utilise une formulation des contraintes en inégalités sur des nombres entiers, la Programmation Linéaire en nombre entiers, et des outils dédiés à ce cadre.

1.1 Programmation Linéaire en nombre entiers

La Programmation Linéaire consiste à résoudre un système d'inégalités numériques sous une contrainte à optimiser. La Programmation Linéaire en nombre entiers (Integer Linear Programming ou ILP en anglais) consiste à restreindre les variables à des valeurs entières, ce qui permet de coder un certain nombre de problèmes d'optimisation combinatoire (résoudre un système de PLNE est un problème NP-complet).
Par exemple, supposons que je veuille mettre 2 euros dans mon porte monnaie, en pièces de 50, 20, 10 et 5 centimes, avec pas plus de 1 pièce de 50c et au moins 1 pièce de chaque sorte, en minimisant le nombre total de pièces.
Je peux coder ce problème avec des variables qui correspondent aux nombres de chaque pièce, x_{50}, x_{20}, x_{10}, x_5 avec les contraintes

- au moins une de chaque : $x_5 \geq 1$, $x_{50} \geq 1$, $x_{10} \geq 1$, $x_{20} \geq 1$,
- pas plus de 1 pièce de 50 : $x_{50} \leq 1$
- total 2 euros (200 centimes) : $50 * x_{50} + 20 * x_{20} + 10 * x_{10} + 5 * x_5 = 200$

Et en minimisant $x_{50} + x_{20} + x_{10} + x_5$.
Un solveur en nombre entier est dédié à la recherche de solution optimale pour ce type de problème.

1.2 Entrées/Sorties d'un résolveur PLNE

Un résolveur peut prendre en entrée plusieurs formats de fichiers encodant un problème sous forme de PLNE.
Un format simple est le format LP, dont voici un exemple :

```
\ LP format example

Maximize
  x + y + z
Subject To
  c0: x + y = 1
  c1: x + 5 y + 2 z <= 10
Bounds
```

```
  0 <= x <= 5
  z >= 2
Generals
  x y z
End
```

Les commentaires commencent après un \, les variables entières à assigner sont listées en fin de fichier dans une section "Generals", le domaine des variables est spécifié dans une section "Bounds". La fonction objectif à optimiser est indiquée dans une section "Maximize" ou bien "Minimize", et les contraintes sont listées dans une section "Subject To". Chaque contrainte doit avoir un label (ici c0 et c1). La fin du fichier est marqué par "End".

1.3 Le solveur SCIP

Le solveur à utiliser est normalement déjà installé et peut être lancé grâce à la commande "scip". Vérifiez la version, qui doit être 3.0 ou 3.1. Si vous n'avez pas la bonne version, vous pouvez lancer à la place la commande :

`~muller/bin/scip`

En supposant que le problème à résoudre soit encodé dans un fichier nommé "test.lp" (vous pouvez essayer avec la version de la section précédente), on peut alors taper dans le solveur :

`read test.lp`

Puis

`optimize`

Si tout va bien, la sortie doit inclure le message `SCIP Status : problem is solved [optimal solution found]` et on peut alors afficher une affectation solution

`display solution`

Ce qui doit afficher

```
objective value:                                  5
x                                                 1  (obj:1)
z                                                 4  (obj:1)
```

Soit la valeur de la fonction à optimiser ('objective') et la valeur affectée aux variables non nulles (donc ici y=0), ainsi que leur contribution à la fonction objective (obj :xx).
On peut sauver ce résultat dans un fichier :

`write solution test.sol`

On peut aussi tout rassembler dans un fichier de commandes (par exemple toto), et tout lancer en une fois avec

`scip -b toto`

Si on veut imposer une limite de temps au solver, on peut modifier ce paramètre avec la commande

`set limits time <nb de secondes>`

On peut aussi stopper la résolution sur d'autres critères (bornes sur l'optimisation, mémoire occupée, ...) ; taper "set limits" sans arguments pour voir les possibilités.

2 Quelques problèmes à représenter en PLNE puis résoudre grâce à Scip

Un grand nombre de problèmes d'optimisation combinatoire peuvent être représentés plus ou moins facilement par PLNE. Nous en allons en voir quelques exemples. Pour chacun de ces problèmes, on donnera des fichiers de test (benchmarks) de difficultés de résolution variables, il faudra :

- encoder le problème en contraintes en nombres entiers (modélisation)
- traduire les formats de représentation des benchmarks pour générer les contraintes correspondantes en nombres entiers
- produire une solution avec scip, et retraduire la solution par rapport aux données de départ.

2.1 Le problème du sac à dos

Dans ce problème, qu'on appelle aussi parfois le problème du contrebandier, on doit faire un choix optimal d'objets à mettre dans un conteneur de taille limité, chaque objet ayant une valeur et un volume spécifique.
Un format d'entrée simple est par exemple :
- première ligne indiquant le volume du conteneur
- chaque ligne suivante définissant un objet, c'est-à-dire un couple volume/valeur
ex :

```
104
2 30
10 200
5 30
...
```

Les problèmes à traiter sont donnés sur le moodle : `ks_2.in` définit un exemple jouet pour tester votre modélisation. Vous pourrez tester sur les fichiers de tests plus gros (`ks*.in`).

2.2 Problèmes de partitionnement : la couverture d'ensemble

Les problèmes de partitionnement sont des problèmes omniprésents.
Imaginons que l'on veuille obtenir au moins un exemplaire de tous les morceaux différents enregistrés par un musicien en achetant le moins de disques possibles. Par exemple Jimi Hendrix a enregistré environ une soixantaine de chansons, mais sa maison de disque a sorti plus de 40 disques à son nom (Deezer en liste 41). Si chaque disque est vu comme un ensemble de morceaux, le problème consiste à trouver un sous-ensemble minimal de disques tel que l'union des morceaux qu'ils contiennent est l'ensemble total des morceaux (en supposant qu'on ne se préoccupe pas d'enregistrements différents de la même chanson). Ce problème est celui de la **couverture d'un ensemble**.
Là aussi, ce problème se représente bien en PLNE.
Vous trouverez sur moodle une instance de ce problème, dans le fichier "`frb30-15-1.msc`" au format suivant :

```
c ....
```

```
s 1 2 3 100
S 2 4 210
```

Une ligne préfixée par "c" est un commentaire. Chaque ligne préfixée par "s" définit un sous-ensemble, chaque numéro codant un objet différent

2.3 Un langage de description de plus haut niveau : ZPL

On remarque que générer les contraintes est assez rébarbatif, même automatiquement, et difficile à débugger. On va maintenant utiliser le format de plus haut niveau pour décrire les contraintes, qui est le format propre à scip depuis la version 3, à savoir ZPL, celui-ci étant beaucoup plus proche du langage de modélisation mathématique.
Admettons qu'on veuille définir une contrainte sur un ensemble de variables binaires $x_1, x_2, ..., x_6$, par exemple que seulement deux seront vraies : $\sum_{i=1}^{i=6} x_i = 2$.
On définit alors un ensemble (*set*) d'indices, et des variables binaires indicées par cet ensemble :

```
set Index:= {1 to 6}
var x[Index] binary;
```

La contrainte s'exprime directement comme :

```
subto ma_contrainte:
     sum i in Index: x[i] == 1;
```

Imaginons que le problème à optmiser est le choix de deux choses de coût différents associés aux indices 1..6. On peut définir des tables en ZPL avec le mot-clef `params`. Par exemple, supposons les coûts suivants :

```
params C[Index] := 1 10, 2 12, 3 14, 4 51, 5 23, 6 70;
```

La fonction objectif est donc $min \sum_{i \in Index} c[i] * x_i$ qu'on exprime :

```
minimize cout: sum i in Index: c[i]*x[i];
```

On peut définir des ensembles de contraintes par exemple si on a un ensemble de variables $x_{i,j}$ dépendant de deux indices, et que l'on veut exprimer une contrainte pour chaque i : $\forall i(\sum_j x_{i,j} = 1)$:

```
set Indi = {1 to 6}
set Indj = {1 to 10}
var x[Indi*Indj] binary;

# genere 6 contraintes différentes
subto une:
   forall i in Indi: sum j in Indj : x[i,j] == 1
```

On peut aussi charger des paramètres directement depuis des fichiers.
Pour plus de détails, cf `http://zimpl.zib.de/download/zimpl.pdf`. Notamment pp 12-13, pour charger des données depuis des fichiers.
A partir de maintenant, on utilisera ce format pour tous les problèmes suivants. Il sera utile de chercher dans le manuel les possibilités de la syntaxe ZPL, même si le sujet vous guidera au fur et à mesure.

2.4 Problèmes de partitionnement : le set packing

Un problème voisin (mais différent) correspond au cas où l'on a un ensemble de sous-ensembles, mais on veut cette fois sélectionner des sous-ensembles disjoints deux à deux, en gardant un nombre maximal de sous-ensembles. Vous pouvez utiliser les instances du problème précédent.

3 Variations

Voici encore quelques problèmes classiques, certains sont équivalents à des instances d'un des problèmes précédents, ou éventuellement en changeant un peu la modélisation. Identifiez ces problèmes ou bien proposez de nouvelles modélisations.
Indice : pour certains problèmes, le choix des variables à utiliser est important...

1. planification de trajets aériens : on dispose d'un ensemble T de "trajets" entre 2 villes comprenant des arrêts intermédiaires, donc composés d'un ensemble E d'étapes, on veut faire suivre à une flotte d'avions un sous-ensemble minimal des trajets de telle sorte que chaque étape soit couverte par un avion.
2. Ordonnancement de tâches : on dispose d'un ensemble de machines qui peuvent réaliser des tâches de durées variables (chaque tâche prend un temps différent selon la machine), et chaque heure de travail a un coût différent sur chacune des machines. On veut minimiser le coût total du projet, avec une durée maximale D fixée à ne pas dépasser.
   A tester avec le fichier "`jobshop.in`", pour une durée à ne pas dépasser de 12 (vous pouvez essayer d'autres valeurs).
3. Variante (plus dur) : le but est de minimiser le temps d'achèvement du projet (on veut que tout soit fini au plus tôt), sans tenir compte du coût. Quel est le coût de la solution la plus rapide ?

4 Plus d'expressivité

Le dernier problème de la section précédente a du vous montrer comment coder une fonction objectif qui doit trouver un min-max ou max-min. Nous allons voir ici d'autres possibilités de modélisation du modèle en nombre entier, là encore en ajoutant des variables judicieuses. Puis nous verrons comment écrire cela plus simplement dans le format ZPL.

4.1 Contraintes conditionnelles, disjonctions, conjonctions, valeurs différentes

On a souvent besoin d'exprimer des contraintes qui dépendent d'une condition, du type si $x \geq 4$ alors on veut respecter la contrainte $y \leq 6$. On sait par ailleurs que x et y sont compris entre 0 et 10.

En introduisant une variable binaire z qui vaut 1 si la condition est vraie, écrivez les contraintes qui forcent la contraintes résultante en fonction de z.
D'une façon similaire, on peut coder des contraintes disjonctives (c'est à dire que l'on veut une contrainte respectée ou une autre), par exemple on veut soit $x + y > 2$ soit $x - y > 3$.
En pratique, dans ZPL on peut directement coder des contraintes conditionnelles, qui réalisent le premier codage de façon transparente :

```
subto untest:
 vif (x>=4) then y <= 6 end;
```

(attention à ne pas confondre avec les if dans les expressions ...). Cette contrainte génère alors les mêmes contraintes que ci-dessus, avec les mêmes restrictions (que des variables entières, et un domaine borné).
En déduire le codage de la contrainte disjonctive... En déduire une façon de coder que deux variables entières sont différentes.
Vous pouvez maintenant lire les pages 18-19 du manuel Zimpl.
Enfin, trouvez un moyen d'exprimer avec une nouvelle variable des conjonctions de contraintes, par exemples que ($x <= 4$) et ($y >= 6$) sont vrais en même temps. Dans quels contextes cela est-il utile ? (NB : si on ajoute à une modélisation ILP les contraintes séparées $x <= 4$, $y >= 6$, elles seront bien sûr vérifiées dans la solution trouvées).

4.2 Coloriage, le retour

Vous avez maintenant tout ce qu'il faut pour résoudre le problème de trouver le nombre chromatique d'un graphe. Vous pouvez supposer qu'on a déjà un encadrement de la vraie valeur grace à un algorithme approché (par exemple si la vraie valeur est 5 on peut supposer qu'on sait déjà que l'on cherche une valeur $<$ à 10). (essayez avec les fichiers flat20_3_0.col, r250.1.col, le450.5a.col). Attention, il y a de multiples façons d'encoder ce problème, certaines plus efficaces que d'autres !

4.3 Termes non linéaires

Modélisez le problème suivant (une instance est sur le moodle) :
Une épidémie de maladie infectieuse a été observée dans un certain nombre N d'endroits. Un ensemble de M équipes de médecins doivent aller enquêter pour identifier la maladie, ce qui leur prend un certain temps t_{ij} qui dépend de l'endroit j et de l'équipe i. Chaque équipe peut enquêter au maximum à 2 endroits, mais doit alors se déplacer de l'endroit j_1 à l'endroit j_2, ce qui prend un temps $d_{j_1 j_2}$. Le but est de minimiser le temps total d'enquête pour pouvoir lancer un éventuel plan de traitement global.
Ce problème comporte un problème de maximum dans la fonction objectif (cf problème III.4), et des contraintes conditionnelles. De plus, la fonction objectif comporte des termes non linéaires ... qu'il faut alors remplacer par de nouvelles variables. L'instance de test sur moodle est nommé "`contagion.in`".

4.4 Valeurs toutes différentes : contrainte "alldiff"

Une contrainte courante et pratique qu'on utilise en satisfaction de contraintes est la contrainte qu'un ensemble de variables x_i prenne des valeurs toutes différentes ("alldiff") dans un domaine entier D. On a déjà vu comment encoder des valeurs différentes avec le problème du

coloriage. Une autre façon de l'encoder en nombres entiers est d'ajouter des variables binaires A_{ilu} qui indiquent si une variable x_i est comprise entre deux valeurs u et l. On ajoute alors juste la contrainte que chaque intervalle ne peut contenir plus de variables que de valeurs disponibles.
Combien de variables et de contraintes sont-elles ajoutées par cette méthode ?
Tester votre modèle sur le fichier de test "`alldiff.in`".

5 Comparaison avec un solveur de contraintes

Il est possible de résoudre un problème d'optimisation avec un solveur de contraintes en le redémarrant plusieurs fois avec une contrainte qui intègre la valeur à optimiser. Choisissez un des problèmes précédents pour le résoudre avec jsolver. (cf le chapitre 10 du manuel utilisateur). Pour quels types de problèmes cela peut-il être une bonne idée à votre avis ?