TPs : Prog. Linéaire en Nombres Entiers avec SCIP

On rappelle qu'un programme linéaire est défini par :

- un ensemble de variables ;
- une fonction *objectif*, fonction linéaire de ces variables à maximiser ou minimiser ;
- et un ensemble de contraintes linéaires sur les valeurs que peuvent prendre ces variables.

De nombreux logiciels permettent de résoudre de tels problèmes, et il existe plusieurs formats de fichiers pour représenter des problèmes d'optimisation combinatoire afin de les résoudre avec ces logiciels. Nous en verrons deux :

- le langage LP (brièvement), qui permet essentiellement d'écrire un objectif et des contraintes linéaire ;
- le langage ZIMPL, de plus haut niveau, qui offre des constructions permettant de décrire de manière plus concise certains problèmes courants d'optimisation combinatoire.

Un exemple en langage LP

Le programme ci-contre est contenu dans le fichier exple_simple.lp sur Moodle. La syntaxe est très simple, quelques mots-clefs délimitent plusieurs sections :

\ commence une ligne de commentaires
Maximize ou Minimize annonce la fonction objectif à optimiser
Subject To annonce les contraintes, chaque contrainte doit avoir un label (ici c et c1)
Bounds annonce les restrictions sur les domaines des variables
Binary annonce la liste des variables binaires
Generals annonce la liste des autres variables entières
End marque la fin du programme

Les variables qui ne sont pas dans les listes Binary et Generals sont autorisées à avoir des valeurs non entières.

```
\ LP format example
Maximize
  3 x1 + x2 + Y676
Subject To
  c: x1 + x2 = 1
  c1: x1 + 5 x2 + 2 Y676 <= 10
Bounds
  0 <= x1 <= 5
  Y676 >= 2
Generals
  x1 Y676
Binary
  x2
End
```

Une description plus précise du format LP se trouve à l'adresse :

`http://lpsolve.sourceforge.net/5.0/CPLEX-format.htm`

Prise en main du solveur

Nous utiliserons le logiciel SCIP, disponible gratuitement à l'adresse scipopt.org. Sur les salles de TPs informatique de la FSI, la version 8 est installée sous linux / fedora, on lance le logiciel avec la commande scip.

Le programme fonctionne en mode «ligne de commande». Il est plus facile de se placer au préalable dans le répertoire où se trouvent les fichiers contenant les programmes linéaires. Par ailleurs, pour avoir un historique des commandes accessible avec les flèches durant l'interaction avec SCIP, on peut utiliser la commande rlwrap suivie du nom de l'exécutable. L'invite de commande du solveur est SCIP>

Une fois le fichier exple_simple.lp récupéré sur Moodle et le solveur lancé, vous pouvez tester les commandes suivantes :

read exple_simple.lp : lit la description d'un problème
optimize : lance la résolution ; lorsque le solveur s'arrête, ou est arrêté, on peut voir si la résolution s'est bien passée sur la ligne commençant par SCIP Status : sur notre exemple, on doit avoir *problem is solved [optimal solution found]*

display solution : affiche la solution, c'est-à-dire la valeur de la fonction à optimiser ('objective') et les valeurs affectées aux variables non nulles (donc ici x2=0), ainsi que leurs contributions à la fonction objective (obj :xx).
write solution test.sol : permet de sauver ce résultat dans un fichier
quit : termine SCIP

Exercice 1. On s'intéresse au problème du sac-à-dos. Écrire, dans un fichier avec le suffixe .lp, un programme linéaire représentant l'instance ci-contre, et le résoudre avec SCIP. Quel est la valeur du sac optimal, et quels objets contient-il ?

Capacité 20kg,

num.	0	1	2	3	4	5	6	7	8	9	10	11
poids	11	7	5	5	4	3	3	2	2	2	2	1
valeur	20	10	25	11	5	50	15	12	6	5	4	30

Le sac-à-dos en langage ZIMPL

On va maintenant utiliser un langage de plus haut niveau pour décrire les contraintes : ZIMPL est un langage propre à SCIP depuis la version 3 ; il permet des constructions beaucoup plus proches du langage de modélisation mathématique. Le manuel du langage est à l'adresse `https://zimpl.zib.de/download/zimpl.pdf`, il est aussi sur Moodle. À la lecture d'un programme ayant la terminaison .zpl, SCIP appelle un traducteur qui génère automatiquement[1] l'instance en langage LP.

Le programme ci-dessous est contenu dans le fichier sac-a-dos-12.zpl, et permet de générer l'instance ci-dessus du sac-à-dos en langage LP ; on l'exécute dans SCIP avec la commande : read sac−a−dos−12.zpl :

```
1 param capacite := 20 ;
2 do print "capacite : " , capacite ;
3 set Objets := { 0 to 11 by 1 } ;
4 param poids[Objets] := <0> 11, <1> 7, <2> 5, ... <10> 2, <11> 1 ;
5 param valeurs[Objets] := <0> 20, <1> 10, <2> 25, ... <10> 4, <11> 30 ;
6 var x[Objets] binary ;
7 maximize valeur : sum<i> in Objets: valeurs[i] * x[i];
8 subto poids : sum<i> in Objets: poids[i] * x[i] <= capacite;
```

Explications, ligne par ligne :

1. le mot réservé `param` permet de déclarer des constantes ou coefficients, ici la capacité du sac.
2. permet d'afficher la valeur de la constante `capacite`
3. déclare un *ensemble* qu'on appelle *Objets* ; les «objets» sont ici les entiers de 0 à 11, et vont être utilisés comme des indices.
4. associe un paramètre appelé «poids» à chaque objet ; par exemple, «`<0> 11` indique qu'on associe un poids de 11 à l'objet 0.
5. associe de même une valeur à chaque objet.
6. associe à chaque objet une variable binaire dont le nom commence par `x` – on verra à l'affichage de la solution que par exemple la variable associée à l'objet 0 a pour nom `x$0`.
7. déclare l'objectif, qui est ici de maximiser $\sum_{i \in \text{Objets}} \text{valeurs}[i] \times x_i$; il faut donner un nom à la fonction objectif, ici `valeur`
8. déclare une contrainte, qu'on appelle `poids` : le poids total ne doit pas dépasser la capacité.

Une fois le fichier lu dans SCIP, on peut lancer l'optimisation puis afficher la solution comme précédemment.

[1] À condition que le bon *reader* ait bien été installé ; c'est le cas avec SCIP3 sur les PC des salles de TPs informatique de la FSI.

Exercice 2 (DÉMÉNAGEURS). On rappelle que dans le problème des DÉMÉNAGEURS, on a un ensemble de n objets numérotés de 1 à n, qu'on veut ranger dans des boites qui ont toutes la même capacité entière $C > 0$. Chaque objet a une taille entière $t_i > 0$, on suppose que pour chaque i, $t_i \leq C$. On veut minimiser le nombre de boites utilisées. On sait que c'est un problème NP-complet,

Une instance qu'on pourra prendre en exemple est :

- capacité $C = 9$;
- 24 objets, de tailles respectives 6, 6, 5, 5, 5, 4, 4, 4, 4, 2, 2, 2, 2, 3, 3, 7, 7, 5, 5, 8, 8, 4, 4, 5.

On veut écrire un programme ZIMPL pour résoudre ce problème, en utilisant deux familles de variables binaires :

- $x_{ij} = 1$ si et seulement si l'objet i est mis dans la boite j ;
- $y_j = 1$ si et seulement si la boite j est utilisée.

Question 2.1. Quel borne supérieure peut-on donner pour le nombre de boites utilisées ?

Question 2.2. Écrivez le modèle mathématique modélisant ce problème avec les variables ci-dessus ; c'est-à-dire, donnez l'objectif et les contraintes :

- chaque objet doit être dans exactement une boite ;
- une boite est utilisée ($y_j = 1 > 0$) si et seulement si elle contient au moins un objet ;
- dans chaque boite, la somme des tailles des objets ne doit pas dépasser la capacité.

Question 2.3. Écrivez les commandes ZIMPL pour

- définir deux ensembles d'indices : un pour les objets, un pour les boites ;
- définir un tableau contenant les tailles des objets ;
- définir un paramètre qui est la capacité des boites ;
- définir un tableau de variables y_j ;
- définir un tableau de variables x_{ij}.
  (Remarque : si I et J sont deux ensembles d'indices, alors la commande "`var x[I*J] binary ;`" définit un tableau de variables à deux dimensions x_{ij}.)

Question 2.4. Traduisez en ZIMPL le modèle mathématique.
(Remarque : en ZIMPL, on peut écrire "`forall <i> in I: sum <j> in J: ...}`" pour indiquer qu'une contrainte, portant ici sur une somme, doit être vérifiée pour toute valeur de i.)

Question 2.5. Résolvez l'instance ci-dessus avec SCIP.

Lecture de données dans un fichier

Le fichier sac-a-dos.zpl permet de définir les paramètres de poids, valeurs et capacité pour une instance du problème du sac-à-dos à partir d'un fichier :

```
1 param fichier := "sac-a-dos-24.txt" ;
2 param capacite :=  read fichier as "1n" comment "#" use 1 ;
3 set Objets := { read fichier as "<1s>" comment "#" skip 1 } ;
4 do print "nb objets : " , card(Objets) ;
5 param poids[Objets] := read fichier as "<1s> 2n" comment "#" skip 1 ;
6 param valeurs[Objets] := read fichier as "<1s> 3n" comment "#" skip 1 ;
7 var x[Objets] binary;
8 maximize valeur : sum<i> in Objets: valeurs[i] * x[i];
9 subto poids : sum<i> in Objets: poids[i] * x[i] <= capacite;
```

*E*xplications : la nouveauté est la fonction `read` qui permet d'extraire des informations d'un fichier, à l'aide de «patrons». Le format général est :

`read` *nom de fichier* `as` *patron* [`comment` s_1] [`match` s_2] [`skip` n_1] [`use` n_2]

Le fichier est lu ligne par ligne, chaque ligne est découpée en *token* – les *tokens* sont délimités par des espaces, tabulations, virgules, points-virgules, ou « :», mais les chaînes de caractères entre «"» ne sont pas découpées. Dans le programme ci-dessus :

- `read fichier as "1n" comment "#" use 1` :
  `"1n"` indique qu'on récupère le 1er token de chaque ligne, interprété comme un entier ; `comment` «#» indique que les lignes commençant par # sont des commentaires ; et `use 1` indique qu'on ne lit qu'1 ligne
- `read fichier as "<1s>" comment "#" skip 1` :
  `"<1s>"` indique qu'on récupère le 1er token de chaque ligne, interprété comme une chaîne de caractères (dans les fichiers de données pour le sac-à-dos, c'est l'identifiant des objets), et on le met entre <> car il va dans un *set* ; et `skip1` indique qu'on ne prend pas la première ligne (normal puisqu'il y a la capacité).
- `read fichier as "<1s> 2n" comment "#" skip 1` :
  le patron `"<1s> 2n"` indique l'association entre l'identifiant qui est dans le premier token et la valeur qui est dans le second.

Exercice 3 (Sac-à-dos – suite). Les fichiers sac-a-dos-xx.txt contiennent des instances du sac-à-dos. Quelle instance a le temps de calcul le plus long ?

Exercice 4 (déménageurs, suite). On veut résoudre des instances de déménageurs décrites dans des fichiers comme `u120_00.bpa`, formatés comme suit :

- la première ligne indique le nom de l'instance ;
- la deuxième ligne a trois nombres : le premier est la capacité, le second est le nombre d'objets, le troisième n'est pas utilisé ;
- les lignes suivantes indiquent les tailles des objets, une par ligne.

Question 4.1. Écrivez un programme ZIMPL qui lit une instance de déménageurs d'un tel fichier et la résout.
Remarque : pour lire les tailles dans un tableau, on pourra utiliser les deux instructions suivantes :

```
set tmp[<i> in I] := {read file as "<1n>" skip 1+i use 1} ;
param taille[<i> in I] := ord(tmp[i],1,1);
```