Projet 2015-2016

MPRO-ECMA

Auteurs :
Paulin Jacquot
Roxane Delpeyrat

March 6, 2016

Contents

1 Modèle mathématique

1.1 Ex.1 : écriture d'un programme linéaire

1.1 On utilise des variables x_{ij} pour $(i, j) \in M$, valant 1 SSI la maille (i, j) est selectionnée. Le programme s'écrit alors :

$$max \sum_{(ij)\in M} x_{ij} \tag{1}$$

$$s.c. \ \frac{\sum H_{ij}^p C_{ij}^p x_{ij}}{\sum C_{ij}^p x_{ij}} + \frac{\sum H_{ij}^a C_{ij}^a x_{ij}}{\sum C_{ij}^a x_{ij}} \geq 2 \tag{2}$$

$$x_{ij} \in \{0, 1\} \tag{3}$$

1.2 Pour linéariser la contrainte fractionnaire (2), on utilise les variables :

$$y = \frac{1}{\sum C_{ij}^p x_{ij}} \qquad z = \frac{1}{\sum C_{ij}^a x_{ij}}$$

ce qui donne les contraintes quadratiques :

$$\sum C_{ij}^p y \cdot x_{ij} = 1 \qquad \sum C_{ij}^a z \cdot x_{ij} = 1 \tag{4}$$

On linéarise ensuite ces contraintes. Pour cela, introduisons les quantités :

$$M^p = \frac{1}{\underset{(i,j)\in M}{min} C_{ij}^p} \qquad M^a = \frac{1}{\underset{(i,j)\in M}{min} C_{ij}^a}$$

bien définies car les coefficients C^p et C^a sont strictement positifs pour tout $(ij) \in M$.

En posant $u_{ij} = x_{ij} \cdot y$ et $v_{ij} = x_{ij} \cdot z$, les contraintes (4) sont alors équivalentes à :

$$\sum_{(i,j)\in M} C^p_{ij} \cdot u_{ij} = 1 \qquad \sum_{(i,j)\in M} C^a_{ij} \cdot v_{ij} = 1 \tag{5}$$

$$u_{ij} \leq x_{ij} \cdot M^p \qquad v_{ij} \leq x_{ij} \cdot M^a, \qquad \forall (i,j) \in M \tag{6}$$

$$u_{ij} \leq y \qquad v_{ij} \leq z, \qquad \forall (i,j) \in M \tag{7}$$

$$u_{ij} \geq (x_{ij} - 1) \cdot M^p + y \qquad v_{ij} \geq (x_{ij} - 1) \cdot M^a + z, \qquad \forall (i,j) \in M \tag{8}$$

$$u_{ij} \geq 0 \qquad v_{ij} \geq 0, \qquad \forall (i,j) \in M \tag{9}$$

La contrainte (2) se réécrit également de façon linéaire :

$$\sum H^p_{ij} C^p_{ij} u_{ij} + \sum H^a_{ij} C^a_{ij} v_{ij} \geq 2 \tag{10}$$

Le programme linéaire s'obtient avec les contraintes (10) et 5, 6,7,8,9 et la même fonction objectif :

$$max \sum_{(ij)\in M} x_{ij}$$

1.2 Ex.2 : modélisation de la connexité

Définissons, pour chaque $h \in [|0, n^2|]$, les variables binaires $l_{ijh} \forall (i,j) \in M$. On modélise alors la connexité comme le suggère l'énoncé : il existe une et une seule maille "racine" de hauteur $h = 0$. Ensuite, chaque maille (ij) sélectionnée se voit attribuer une hauteur h (et alors $l_{ijh} = 1$) et une maille est selectionnée avec hauteur $h > 0$ si une de ses voisines est selectionnée avec hauteur $h - 1$.

Pour toute maille selectionnée (i,j), il existe donc un chemin empruntant des mailles selectionnées jusqu'à la maille racine de hauteur 0. La solution est donc étoilée par rapport à cette maille racine, donc connexe.

Les contraintes s'écrivent donc de la manière suivante :

$$\sum_{(ij)\in M} l_{ij0} = 1 \text{ (une et une seule racine)} \quad (11)$$

$$\sum_{h=0}^{n^2} l_{ijh} = x_{ij},\ \forall (i,j) \in M \quad (12)$$

$$l_{ijh+1} \leq l_{i-1jh} + l_{i+1jh} + l_{ij-1h} + l_{ij+1h}, \forall (i,j) \in M,\ h \in [|0, n^2-1|] \quad (13)$$

$$ł_{ijh} \in \{0,1\},\ \forall (i,j) \in M, \forall h \in [|0, n^2|] \quad (14)$$

Cela représentant un très grand nombre de variables et de contraintes (en $\mathcal{O}(n^4)$), on pourra ajouter les contraintes au fur et à mesure, seulement si elles sont violées.

2 Résolution directe

2.1 Résolution frontale et borne

Nous avons implémenté le modèle précédent en utilisant l'API Cplex C++. Le modèle permet la résolution exacte des plus petites instances (5x8), mais le nombre de variables est trop élevé pour les instances plus grandes.

Dans une heuristique plus évoluée, nous effectuons une première résolution du problème sans contraintes de connexités. La valeur optimale obtenue $M*$ donne ensuite une très bonne borne sur la hauteur maximale de l'arbre de connexité décrit dans la partie précédente (la borne triviale était de l'ordre de $n \times m$). La hauteur de l'arbre de peut en effet dépasser :

$$h_{max} = \frac{1}{2} M * +1$$

Cela permet ainsi de générer significativement moins de variables binaires correspondant aux contraintes de connexité.

Avec ce modèle, nous arrivons à résoudre quelques unes des instances 10x12, mais cette méthode n'est pas efficace pour les instances ayant une solution non connexe de valeur élevée (comme l'instance 10_12_1.*dat*).

Instance	$M*$ (non conn.)	Res connexe	CPU time (s)
$projet581$	24	24	0.356659
$projet582$	4	4	0.184414
$projet583$	30	30	0.644796
$projet584$	18	18	1.04376
$projet585$	40	40	0.08267
$projet586$	23	23	0.219315
$projet587$	30	30	13.5549
$projet588$	28	28	14.762
$projet589$	23	23	13.4261
$projet5810$	40	40	0.077614

Total simulation time : 44.3763s.

Figure 1: *Résolution directe avec arbre de connexité, sans Callback*

Le fait de connaître la valeur optimale $M*$ du problème relaché sans les contraintes de connexité nous donne aussi une borne supérieure de la solution du problème initiale. Cette borne fournit en pratique une aide très précieuse à Cplex qui n'est pas capable de génerer la coupe $\sum x_{i,j} \leq M*$.

Par exemple, la résolution de l'instance *projet 5 8 1* prend 13 secondes avec cette coupe, mais plus de 30 minutes sans cette coupe.

Le tableau suivant donne les résultats obtenus par cette méthode sur les premières instances :

2.2 Ajouts successifs des contraintes de connexité

Afin de résoudre plus rapidement le problème, nous avons tenté, comme dans la résolution typique du voyageur de commerce, d'ajouter les contraintes de connexités décrites dans la partie 1 au fur et à mesure de la résolution du problème, en vérifiant le problème de "séparation" à chaque résolution.

Cela se fait simplement en utilisant la méthode *ILOLAZYCONSTRAINT-CALLBACK* disponible dans l'API Cplex. Cependant, en pratique, cela n'amène pas à une résolution beaucoup plus rapide : beaucoup de contraintes doivent être ajoutées pour obtenir une solution connexe, et finalement, pour certaines instances, la résolution est plus longue que si l'on considère toutes les contraintes de connexité dès le départ.

Cette méthode ne permet donc pas de résoudre les instances plus larges que 10x12.

2.3 Minimisation des bords

Nous avons également implémenté une méthode heuristique de minimisation des bords de la solution obtenue.

La méthode consiste à introduire $(n+1)\cdot m+n\cdot(m+1)$ variables binaires supplémentaires $e^h_{i,j}$ et $e^v_{i,j}$ correspondant aux arêtes horizontales et verticales de chaque case du damier. La variable $e_{i,j}$ vaut 1 SSI elle définit un "bord" de la solution, ce qui se traduit par les contraintes:

$$e^h_{i,j} \geq x_{i,j} - x_{i-1,j} \tag{15}$$
$$e^h_{i,j} \geq x_{i-1,j} - x_{i,j} \tag{16}$$
$$e^v_{i,j} \geq x_{i,j} - x_{i,j-1} \tag{17}$$
$$e^v_{i,j} \geq x_{i,j-1} - x_{i,j} \tag{18}$$

Comme une case sélectionnée dans la solution “rajoute” au plus deux bords, on utilise maintenant la fonction objectif :

$$\sum_{i,j} x_{i,j} - \alpha \sum_{i,j} \left(e^h_{i,j} + e^v_{i,j}\right)$$

En pratique, nous avons pris $\alpha = 0.49$. Cette méthode permet une résolution assez rapide et optimale sur les premières instances, mais ne permet toujours pas d'obtenir la résolution exacte des instances 10x12 en général.

Les résultats pour les premières instances sont regroupés dans le tableau suivant (figure **??**).

Instance	$M*$ (non conn.)	Res connexe	CPU time (s)
$projet581$	24	24	0.889101
$projet582$	4	4	0.695555
$projet583$	30	30	0.785902
$projet584$	18	18	0.602633
$projet585$	40	40	0.013569
$projet586$	23	23	0.741241
$projet587$	30	30	7.70596
$projet588$	28	28	0.436938
$projet589$	23	23	1.18512
$projet5810$	40	40	0.01264
$Totalsimulationtime : 13.0807s.$			

Figure 2: *Résolution directe avec minimisation des bords.*

3 Résolution par recuit simulé

3.1 Présentation de l'algorithme

Le recuit simulé est une métaheuristique assez courante mais efficace même si la paramétrisation peut se révéler difficile.
Dans l'implémentation de l'algorithme, nous avons choisi de n'examiner que les solutions connexes car construire une solution connexe est difficile. Les fonctions de voisinages sont donc implémentées pour ne générer que des solutions connexes.
Par contre, on accepte de considérer les solutions ne respectant la contrainte fractionnaire liée au relief (voir la description du problème dans la première partie contrainte 2).

Voici l'algorithme du recuit simulé:

Algorithm 1: *Algorithme du recuit*

```
Soit x une solution initiale.
$x_{max} \leftarrow x$
$T \leftarrow T_{init}$
for k allant de 0 à $K_{max}$ do
    for r allant de 0 à $R_{max}$ do
        Générer une solution x' dans le voisinage de la solution courante x
        de manière aléatoire
        Reconstruire une solution admissible à partir de x'.
        if $f(x') > f(x)$ then
            $xgetsx'$
            Soit f la fonction objectif.
            if $f(x') > f(x_{max})$ then
                $x_{max} \leftarrow x'$
            end if
        else
            On tire q un nombre aléatoire.
            $e \leftarrow \exp \frac{f(x')-f(x)}{T}$
            if $q < e$ then
                $x \leftarrow x'$
            end if
        end if
    end for
    $T \leftarrow T * \phi$
    if aucun changement de maximum à cette itération then
        break
    end if
end for
```

3.2 Description des voisinages utilisés

En plus du choix des paramètres, le voisinage utilisé influence l'efficacité de l'algorithme. La première idée a été de créer un voisinage constitué de toutes les solutions différant de 1 par rapport à la solution initiale et de l'évaluer si la solution obtenue était connexe. Cette solution n'est pas idéale dans le cas où il y a peu de cases dont la valeur puisse être changée tout en gardant la connexité de la solution.
La fonction de voisinage implémentée commence donc par voir si on peut ajouter à la solution une case dont la somme des coefficients $H - a[i][j] + H_p[i][j] >= 2$. Si oui, la solution est générée en ajoutant toutes les cases

répondant à ce critère. Si ce n'est pas le cas, l'algorithme tire au hasard une des cases parmi celles à 1 dans la solution d'origine et les voisines de celles-ci. Si la solution obtenue en changeant la valeur de la case tirée au sort est connexe alors la fonction de voisinage renvoie la solution obtenue. Sinon elle tire au sort une autre case.

3.3 Résultats obtenus

L'algorithme est assez efficace et trouve une bonne solution en moins de quelques secondes pour les petites instances et en moins de 2 minutes pour les plus grandes même s'il ne trouve pas toujours le maximum.