UNIVERSITÉ NATIONALE DU VIETNAM – UNV

INSTITUT FRANCOPHONE INTERNATIONALE - IFI

VIỆN QUỐC TẾ PHÁP NGỮ
Institut Francophone International

Cours de Programmation par contrainte

Enseignant : Pham Quang Dung

Master 1

Rapport de Projet

TP réalisé par :
Ginel DORLEON
Aout 2016

Plan

I- Analyse du problème

En recherche opérationnelle et en optimisation combinatoire, le ***bin packing*** est un problème algorithmique. Il s'agit de ranger des objets avec un nombre minimum de boîtes. Le problème classique se définit en une dimension, mais il existe de nombreuses variantes en deux ou trois dimensions.

Le problème de *bin packing* peut être appliqué à un grand nombre de secteurs industriels ou informatiques.

Pour la version classique en une dimension:

◆ Rangement de fichiers sur un support informatique;
◆ Découpe de câbles;
◆ Remplissage de camions ou de containers avec comme seule contrainte le poids ou le volume des articles.

Pour la version en deux dimensions:

◆ Découpe de matière première;
◆ Placement de boîtes sur une palette (sans superposition de boîtes);
◆ Placement dans un entrepôt (sans superposition de boîtes).

Pour la version en trois dimensions:

◆ Rangement d'objets physiques dans des boîtes, un entrepôt, des camions, etc. (avec superposition de boîtes, de palette, etc.).

II Formulation du problème- Modélisation Mathématique

Dans le problème classique, les données sont :

➢un nombre infini de boîtes de taille $\boldsymbol{C}$;
➢une liste 1, 2, . . . , n d'articles i de taille $\boldsymbol{ci}$

On cherche à trouver le rangement valide pour tous ces articles qui minimise le nombre de boîtes utilisées. Pour qu'un rangement soit valide, la somme des tailles des articles affectés à une boîte doit être inférieure ou égale à.

Pour décrire une solution, on peut utiliser un codage binaire pour indiquer dans quelle boîte chaque objet est rangé.

- La variable ***xij*** vaudra 1 si l'article ***i*** est rangé dans la boîte ***j*** , et 0 sinon.
- La variable binaire ***y j*** est égale à 1 si la boîte ***j*** est utilisée, 0 sinon.

On cherche donc à minimiser le nombre de boîtes utilisées

Min $\sum$ yj ;

Sous les contraintes :

$\sum$ Cixij ≤ Cyj, j=1, …., n

$\sum$ Cixij ≤ Cyj, i=1, …., n

xij ∈{0,1}, i = 1, ……., n, j = 1,..n

yj ∈ {0,1}, j =, 1, ……, n,

La première inégalité signifie qu'on ne peut dépasser la taille d'une boîte pour un rangement. À noter que la partie droite de l'inégalité *oblige* à prendre la valeur dès qu'un article est rangé dans la boîte. La deuxième inégalité impose à tous les objets d'être rangés dans une boîte et une seule. Toute solution pour laquelle la famille d'équations précédente est vérifiée est dite *réalisable*.

La modélisation décrite plus haut a été proposée par Leonid Kantorovich en 1960. Outre l'approche de Leonid Kantorovich, il existe d'autres formulations linéaires pour ce problème, sous forme d'un problème de flot maximum dans un graphe.

Une autre manière de définir les contraintes est la suivante

a- Contraintes

2 rectangles ne se superposent jamais

En ce qui concerne la contrainte sur la superposition, on a 4 cas:
$\forall i, j \in \{1, ..., N\}$ i on a:

- $o_i = 0$ et $o_j = 0 \Rightarrow$
$$\begin{cases} \mathrm{x}_i + w_i \leq x_j \\ \text{ou} \\ x_j + w_j \leq x_i \\ \text{ou} \\ y_j + h_j \leq y_i \\ \text{ou} \\ y_i + h_i \leq y_j \end{cases}$$

Ici plus haut, c'est le premier cas sur les contraintes de superposition, c'est très important dans le traitement de BinPacking 2D, cela aide à gagner beaucoup en terme d'espace d'utilisation des bins.

Voir plus bas la définition des autres contraintes pour la non superposition des rectangles.

- $o_i = 0$ et $o_j = 1 \Rightarrow \begin{cases} \mathrm{x}_i + w_i \leq x_j \\ \text{ou} \\ x_j + h_j \leq x_i \\ \text{ou} \\ y_j + h_j \leq y_i \\ \text{ou} \\ y_i + h_i \leq y_j \end{cases}$
- $o_i = 1$ et $o_j = 0 \Rightarrow \begin{cases} \mathrm{x}_i + h_i \leq x_j \\ \text{ou} \\ x_j + w_j \leq x_i \\ \text{ou} \\ y_j + h_j \leq y_i \\ \text{ou} \\ y_i + w_i \leq y_j \end{cases}$
- $o_i = 1$ et $o_j = 1 \Rightarrow \begin{cases} \mathrm{x}_i + h_i \leq x_j \\ \text{ou} \\ x_j + h_j \leq x_i \\ \text{ou} \\ y_j + w_j \leq y_i \\ \text{ou} \\ y_i + w_i \leq y_j \end{cases}$

Le rectangle doit toujours se situer dans le conteneur

En ce qui concerne les contraintes sur la position du rectangle dans la grille, on distingue toujours 2 cas: $\forall i \in \{1, ..., N\}$

- $o_i = 0 \Rightarrow \begin{cases} \mathrm{x}_i + w_i \leq W \\ \text{et} \\ y_i + h_i \leq H \end{cases}$

- $o_i = 1 \Rightarrow \begin{cases} \mathrm{x}_i + h_i \leq W \\ \text{et} \\ y_i + w_i \leq H \end{cases}$

Domaine des variables

$\forall i \in \{1, ..., N\}$

$Domaine(x_i) = \{0, ..., W\}$

$Domaine(x_i) = \{0, ..., H\}$

$Domaine(o_i) = \{0, 1\}$

III- IMPLÉMENTATION

Nous implémentons le problème en utilisant les librairies CHOCO et CBLS.

a- Implémentation avec CHOCO

```
IntegerVariable[] x = new IntegerVariable[n];
IntegerVariable[] y = new IntegerVariable[n];
IntegerVariable[] o = new IntegerVariable[n];
for (int i = 0; i < n; i++) {
x[i] = Choco.makeIntVar("x[" + i + "]", 0, W);
y[i] = Choco.makeIntVar("y[" + i + "]", 0, H);
o[i] = Choco.makeIntVar("o[" + i + "]", 0, 1);
}
for (int i = 0; i < n; i++) {
// constraint on inside the grid
m.addConstraint(Choco.implies(Choco.eq(o[i], 0),
Choco.and(Choco.leq(Choco.plus(x[i], w[i]), W),

Choco.leq(Choco.plus(y[i], h[i]), H))));

m.addConstraint(Choco.implies(Choco.eq(o[i], 1),
Choco.and(Choco.leq(Choco.plus(x[i], h[i]), W),

Choco.leq(Choco.plus(y[i], w[i]), H))));
}
for (int i = 0; i < n - 1; i++) {
for (int j = i + 1; j < n; j++) {
// o[i] = 0, o[j] = 0
// constraint on overlap
m.addConstraint(Choco.implies(
Choco.and(Choco.eq(o[i], 0), Choco.eq(o[j], 0)),
Choco.or(Choco.leq(Choco.plus(x[i], w[i]), x[j]),
Choco.leq(Choco.plus(x[j], w[j]), x[i]),
Choco.leq(Choco.plus(y[j], h[j]), y[i]),
Choco.leq(Choco.plus(y[i], h[i]), y[j])))));
// o[i] = 0, o[j] = 1
m.addConstraint(Choco.implies(
Choco.and(Choco.eq(o[i], 0), Choco.eq(o[j], 1)),
Choco.or(Choco.leq(Choco.plus(x[i], w[i]), x[j]),
Choco.leq(Choco.plus(x[j], h[j]), x[i]),
Choco.leq(Choco.plus(y[j], w[j]), y[i]),
Choco.leq(Choco.plus(y[i], h[i]), y[j])))));
```

```
// o[i] = 1, o[j] = 0
// constraint on overlap
m.addConstraint(Choco.implies(
Choco.and(Choco.eq(o[i], 1), Choco.eq(o[j], 0)),
Choco.or(Choco.leq(Choco.plus(x[i], h[i]), x[j]),
Choco.leq(Choco.plus(x[j], w[j]), x[i]),
Choco.leq(Choco.plus(y[j], h[j]), y[i]),
Choco.leq(Choco.plus(y[i], w[i]), y[j]))));

// o[i] = 1, o[j] = 1
// constraint on overlap
m.addConstraint(Choco.implies(
Choco.and(Choco.eq(o[i], 1), Choco.eq(o[j], 1)),
Choco.or(Choco.leq(Choco.plus(x[i], h[i]), x[j]),
Choco.leq(Choco.plus(x[j], h[j]), x[i]),
Choco.leq(Choco.plus(y[j], w[j]), y[i]),
Choco.leq(Choco.plus(y[i], w[i]), y[j]))));
```

b- Implémentation avec CBLS

```
        y = new VarIntLS[n];
        x = new VarIntLS[n];
        o = new VarIntLS[n];
        for (int i = 0; i < n; i++) {
        x[i] = new VarIntLS(ls, 0, W);
        o[i] = new VarIntLS(ls, 0, 1);
        y[i] = new VarIntLS(ls, 0, H);
        }
        for (int i = 0; i < n; i++) {
        // constraint on inside the grid
        IConstraint[] c5 = new IConstraint[2];
        c5[0] = new LessOrEqual(new FuncPlus(x[i], w[i]), W);
        c5[1] = new LessOrEqual(new FuncPlus(y[i], h[i]), H);
        S.post(new Implicate(new IsEqual(o[i], 0), new AND(c5)));

        IConstraint[] c6 = new IConstraint[2];
        c6[0] = new LessOrEqual(new FuncPlus(x[i], h[i]), W);
        c6[1] = new LessOrEqual(new FuncPlus(y[i], w[i]), H);
        S.post(new Implicate(new IsEqual(o[i], 1), new AND(c6)));
        }
        for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {

        // o[i] = 0, o[j] = 0
        IConstraint[] c1 = new IConstraint[4];
        c1[0] = new LessOrEqual(new FuncPlus(x[i], w[i]), x[j]);
        c1[1] = new LessOrEqual(new FuncPlus(x[j], w[j]), x[i]);
        c1[2] = new LessOrEqual(new FuncPlus(y[j], h[j]), y[i]);
        c1[3] = new LessOrEqual(new FuncPlus(y[i], h[i]), y[j]);
        S.post(new Implicate(new AND(new IsEqual(o[i], 0), new IsEqual(
        o[j], 0)), new OR(c1)));

        // o[i] = 1, o[j] = 0
        IConstraint[] c2 = new IConstraint[4];
        c2[0] = new LessOrEqual(new FuncPlus(x[i], h[i]), x[j]);
        c2[1] = new LessOrEqual(new FuncPlus(x[j], w[j]), x[i]);
        c2[2] = new LessOrEqual(new FuncPlus(y[j], h[j]), y[i]);
        c2[3] = new LessOrEqual(new FuncPlus(y[i], w[i]), y[j]);
        S.post(new Implicate(new AND(new IsEqual(o[i], 1), new IsEqual(
        o[j], 0)), new OR(c2)));
```

```
// o[i] = 1, o[j] = 0
IConstraint[] c2 = new IConstraint[4];
c2[0] = new LessOrEqual(new FuncPlus(x[i], h[i]), x[j]);
c2[1] = new LessOrEqual(new FuncPlus(x[j], w[j]), x[i]);
c2[2] = new LessOrEqual(new FuncPlus(y[j], h[j]), y[i]);
c2[3] = new LessOrEqual(new FuncPlus(y[i], w[i]), y[j]);
S.post(new Implicate(new AND(new IsEqual(o[i], 1), new IsEqual(
o[j], 0)), new OR(c2)));

// o[i] = 0, o[j] = 1
IConstraint[] c3 = new IConstraint[4];
c3[0] = new LessOrEqual(new FuncPlus(x[i], w[i]), x[j]);
c3[1] = new LessOrEqual(new FuncPlus(x[j], h[j]), x[i]);
c3[2] = new LessOrEqual(new FuncPlus(y[j], w[j]), y[i]);
c3[3] = new LessOrEqual(new FuncPlus(y[i], h[i]), y[j]);
S.post(new Implicate(new AND(new IsEqual(o[i], 0), new IsEqual(
o[j], 1)), new OR(c3)));

// o[i] = 1, o[j] = 1
IConstraint[] c4 = new IConstraint[4];
c4[0] = new LessOrEqual(new FuncPlus(x[i], h[i]), x[j]);
c4[1] = new LessOrEqual(new FuncPlus(x[j], h[j]), x[i]);
c4[2] = new LessOrEqual(new FuncPlus(y[j], w[j]), y[i]);
c4[3] = new LessOrEqual(new FuncPlus(y[i], w[i]), y[j]);
S.post(new Implicate(new AND(new IsEqual(o[i], 1), new IsEqual(
o[j], 1)), new OR(c4)));
}
```

IV - Expérimentation et Présentation des solutions

Ici on présente des tests sur les données fournies avec l'énoncé du problème .On considère des intervalles de temps différents pour OPENCBLS et CHOCO. Le programmes a été exécuté sur une machine TOSHIBA, de processeur AMD E1-2100 APU with Radeon(TM) HD Graphics × 2

Instant considéré : t=200s

Entrée	Réussi avec CHOCO	Réussi avec CBLS
bin-packing-2D-W10-H7-I6	Oui	
bin-packing-2D-W10-H8-I6	Oui	
bin-packing-2D-W19-H16-I21	Non	
bin-packing-2D-W19-H17-I21	Oui	
bin-packing-2D-W19-H18-I21	Oui	
bin-packing-2D-W19-H20-I21	Oui	
bin-packing-2D-W19-H19-I21	Oui	

Instant considéré : t=10s

Entrée	Réussi avec CHOCO	Réussi avec CBLS
bin-packing-2D-W10-H7-I6	Oui	
bin-packing-2D-W10-H8-I6	Oui	
bin-packing-2D-W19-H16-I21	Non	
bin-packing-2D-W19-H17-I21	Oui	
bin-packing-2D-W19-H18-I21	Oui	

bin-packing-2D-W19-H20-I21	Oui	
bin-packing-2D-W19-H19-I21	Oui	

V- CONCLUSION

Ainsi, par ces méthodes, on voit comment on peur réussir a résoudre le problème de binpacking 2D. Pour faciliter la tâche, nous avions utilisé deux librairies java: OpenCBLS et Choco. Nous avions donné le modèle du problème c'est à dire: les variables, le domaine de chaque variable et les contraintes entre les variables ensuite les programmes sont fournis en annexe, faits en Choco et en OpenCBLS. Ensuite, nous avons présenté une étude comparative entre les deux librairies sera ensuite faite avec le temps de recherche comme paramètre.