Spatially Balanced Latin Square

Modélisation et résolution du problème SBLS
avec la Programmation par Contraintes

Daniel Carriba Nosrati

UE Introduction à la Programmation par Contraintes
Master 1 Informatique
Université Côte d'Azur

Semestre 1 2025/2026

Sommaire

1 Introduction

Ce projet a pour but de modéliser le problème SBLS avec la Programmation par Contraintes, et d'essayer de le résoudre pour le plus grand n possible.

1.1 Présentation du problème SBLS

Le problème SBLS (Spatially Balanced Latin Square), en français "Carré Latin Spatialement Équilibré", est un problème où on dispose d'un carré de taille $n \times n$ (pour un n donné) et qui possède les mêmes contraintes que le problème du Carré Latin, soit pour chaque ligne et colonne il y a exactement une seule occurence de i $\forall i \in [0, n-1]$, autrement dit tout les éléments sont tous différents un à un au sein d'une ligne et d'une colonne, pour toute ligne et colonne du carré $n \times n$.

Le problème SBLS (ou Carré Latin Spatialement Équilibré) possède une contrainte supplémentaire comparé au problème du Carré Latin. Cette contrainte supplémentaire contraint que la somme des distances entre chaque paire de nombres i et j est égale, $\forall i \in [0, n-1]$, $\forall j \in [0, n-1]$.

1.2 Présentation du projet

Dans ce projet, le problème SBLS à été modélisé avec la Programmation par Contraintes grâce à la librairie Java Choco-solver. Plusieurs méthodes de résolution différentes ont étés implémentées, qui seront comparé à une méthode dite "simple".

Dans ce rapport on présentera la méthode dite "simple" ainsi que les autres méthodes de résolution qui ont étés implémentées dans notre programme. De plus on présentera pour tout n les différentes statistiques de résolution des différentes méthodes implémentées, comme le temps de résolution ou encore le nombre de noeuds utilisés.

1.3 Matériel utilisé

Les statistiques calculées par notre programme et présentées dans ce rapport ont été réalisées sur la machine suivante :

- Apple MacBook Pro (14 pouces, 2021)
- CPU : Apple M1 Pro, 8 cœurs CPU (dont 6 hautes performances et 2 à haute efficacité énergétique)
- GPU : Apple M1 Pro, 14 cœurs GPU
- RAM : 16 Go de mémoire unifiée
- OS : macOS Sequoia 15.7.1

Java version 21.0.8 distribué par Oracle ainsi que Choco-solver version 4.10.14 ont été utilisés.

2 Modélisation

Pour pouvoir implémenter et résoudre le problème SBLS avec Choco-solver, on a choisit de modéliser le problème SBLS avec les variables, les domaines et les contraintes suivantes.

2.1 Variables

Les variables de notre problème seront représentés par $Variables$ de type `IntVar[][]`, soit une matrice d'éléments de type `IntVar`.

$Variables$ est définit mathématiquement de la manière suivante :

$$Variables = \{v_{i,j}, \forall i \in [0, n-1], \forall j \in [0, n-1]\}$$

où $v_{i,j}$ représente la valeur au sein de la cellule (i, j), i.e. la cellule de la ligne $i \in [0, n-1]$ et de la colonne $j \in [0, n-1]$ du carré $n \times n$.

Dans notre programme $Variables$ est définit de la manière suivante :

```
private IntVar[][] variables;
variables = model.intVarMatrix(n, n, 0, n-1);
```

2.2 Domaines

Les domaines des variables est définit mathématiquement de la manière suivante:

$$Domaines = \{D_{i,j}, \forall i \in [0, n-1], \forall j \in [0, n-1]\}$$

avec

$$D_{i,j} = \{0, \ldots, n-1\}, \forall i \in [0, n-1], \forall j \in [0, n-1]$$

où $D_{i,j}$ est le domaine de la variable $v_{i,j}$, $\forall i \in [0, n-1], \forall elt \in [0, n-1]$

Cela signifie que toute cellule (i, j) du carré $n \times n$ peut prendre pour valeur entre 0 et $n - 1$.

2.3 Contraintes

2.3.1 Contrainte sur les lignes

La première contrainte du problème SBLS proscrit que il y a exactement une seule occurence de chaque élément elt au sein d'une ligne i du carré $n \times n$ $\forall i \in [0, n-1]$.

Dans notre modélisation, la variable $v_{i,j}$ représente la valeur de la cellule (i, j) du carré $n \times n$.

Ainsi, cette première contrainte est mathématiquement défini de la manière suivante :

$$\forall i \in [0, n-1], v_{i,0} \neq v_{i,1} \neq \ldots \neq v_{i,n-1}$$

ce qui signifie que pour toute ligne i du carré $n \times n$ tout $elt \in [0, n-1]$ est à une position différente.

Dans notre programme cette contrainte à été implémentée de la manière suivante :

```
for (int i = 0; i < n; i++) {
    model.allDifferent(variables[i]).post();
}
```

2.3.2 Contrainte sur les colonnes

La seconde contrainte du problème SBLS proscrit que il y a exactement une seule occurence de chaque élément elt au sein d'une colonne j du carré $n \times n$ $\forall j \in [0, n-1]$.

Dans notre modélisation, la variable $v_{i,j}$ représente la valeur de la cellule (i, j) du carré $n \times n$.

Ainsi, cette seconde contrainte est mathématiquement défini de la manière suivante :

$$\forall j \in [0, n-1], v_{0,j} \neq v_{1,j} \neq \ldots \neq v_{n-1,j}$$

ce qui signifie que pour toute colonne j du carré $n \times n$ tout $elt \in [0, n-1]$ est à une position différente.

Dans notre programme cette contrainte à été implémentée de la manière suivante :

```
for (int j = 0; j < n; j++) {
    IntVar[] column = new IntVar[n];

    for (int i = 0; i < n; i++) {
        column[i] = variables[i][j];
    }

    model.allDifferent(column).post();
}
```

2.3.3 Contrainte sur l'équilibre spatiale

La troisième et dernière contrainte du problème SBLS est la contrainte sur l'équilibre spatiale. Pour respecter cette contrainte il faut que toutes les sommes des distances entre deux éléments $elt1$ et $elt2$ soit égales, pour tout couple d'éléments $(elt1, elt2)$ $\forall elt1 \in [0, n-1], \forall elt2 \in [0, n-1$.

Ainsi, cette contrainte est mathématiquement défini de la manière suivante :

$$\begin{aligned} EnsembleSommes = & \left\{ \forall elt1 \in [0, n-1], \forall elt2 \in [0, n-1], \sum_{i=0}^{n-1} (distance(elt1, elt2) \text{ au sein de la ligne } i) \right\} \\ & \bigcup \left\{ \forall elt1 \in [0, n-1], \forall elt2 \in [0, n-1], \sum_{j=0}^{n-1} (distance(elt1, elt2) \text{ au sein de la ligne } j) \right\} \end{aligned}$$

$$\forall sum_k \in EnsemblesSommes, sum_0 = sum_1 = ...$$

Dans notre programme cette contrainte à été implémentée en suivant le pseudo-code suivant :

```
sommesDistances = {}
pour elt1 dans Elements :
    pour elt2 dans Elements :
        sommeDistanceLigne = 0
        pour toute ligne i du carre n*n :
            sommeDistanceLigne += Distance(elt1, elt2)
        sommeDistanceColonne = 0
        pour toute colonne j du carre n*n :
            sommeDistanceColonne += Distance(elt1, elt2)
```

```
        nouvelleContrainte(sommeDistanceLigne = sommeDistanceColonne)
        sommesDistances.ajouter(sommeDistanceLigne)
nouvelleContrainte(sommesDistances.tousEgaux)
```

3 Méthodes de résolution

3.1 Méthode simple

3.2 Méthode avancée

4 Résultats

4.1 Méthode simple

	n=2	n=3	n=4	n=5	n=6	n=7
Solution Found	Yes	Yes	No	Yes	Yes	?
Building time (in s)	0,041	0,055	0,071	0,122	0,203	?
Resolution Time (in s)	0,022	0,033	0,518	0,228	39,315	?
Nodes	2	7	485	76	10944	?
Nodes per second	89,6	215	936,4	333,9	278,4	?
Backtracks	0	0	971	55	21671	?
Backjumps	0	0	0	0	0	?
Fails	0	0	486	29	10842	?
Restarts	0	0	0	0	0	?

Table 1: Résultats d'exécutions de la méthode de résolution simple

Après plus de 30 minutes, la méthode de résolution simple n'avait pas produit de résultats pour n=7

4.2 Méthode avancée

5 Conclusion