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. Deux méthodes de résolution différentes ont étés implémentées, une première dite "simple" ainsi qu'une seconde dite "avancée".

Dans ce rapport on présentera les deux méthodes de résolution qui ont étés implémentées dans notre programme. Ensuite 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. Finalement on va comparer les résultats des deux méthodes.

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

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 (i.e. l'élément) 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 (i.e. l'élément) 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 :

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

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

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

Dans ce projet nous avons implémentée deux méthodes de résolutions différentes.

3.1 Méthode simple

La première méthode de résolution est la méthode simple. La méthode simple à été implémentée par la classe `SimpleSpatiallyBalancedLatinSquare` qui étend la classe abstraite `SpatiallyBalancedLatinSquare`.

La classe abstraite `SpatiallyBalancedLatinSquare` contient la modélisation, avec les variables et les contraintes, du problème SBLS.

Pour la résolution de problème, la méthode de résolution simple (`SimpleSpatiallyBalancedLatinSquare`) ne précise pas de stratégie de recherche, et précise comme algorithme de filtrage pour les contraintes `alldifferent` sur les lignes du carré $n \times n$ et `alldifferent` sur les colonnes du carré $n \times n$ l'algorithme de filtrage par défaut (`DEFAULT`). L'algorithme de filtrage par défaut utilisé par Choco-solver est un compromis entre `BC` et `AC_ZHANG`.

3.2 Méthode avancée

La seconde méthode de résolution est la méthode avancée. La méthode avancée à été implémentée par la classe `AdvancedSpatiallyBalancedLatinSquare` qui étend la classe abstraite `SpatiallyBalancedLatinSquare`.

La classe abstraite `SpatiallyBalancedLatinSquare` contient la modélisation, avec les variables et les contraintes, du problème SBLS.

Pour la résolution de problème, la méthode de résolution avancée (`AdvancedSpatiallyBalancedLatinSquare`) précise une stratégie de recherche, et précise comme algorithme de filtrage pour les contraintes `alldifferent` sur les lignes du carré $n \times n$ et `alldifferent` sur les colonnes du carré $n \times n$ l'algorithme de filtrage `AC`. L'algorithme de filtrage `AC` utilisé par Choco-solver est l'algorithme Arc-Consistency de Zhang, une amélioration de l'algorithme Arc-Consistency de Régin.

La méthode de résolution avancée précise une stratégie de recherche. La stratégie de recherche utilisée par la méthode de résolution avancée est la stratégie `domOverWDegSearch`.

Cette stratégie de recherche sélectionne les variables selon le rapport entre la taille du domaine et le degré pondéré des contraintes. Cette stratégie s'adapte dynamiquement à la difficulté des contraintes pendant la recherche, ce qui permet une exploration plus efficace de l'espace de recherche pour des problèmes fortement contraints comme SBLS.

En effet `domOverWDegSearch` s'est avérée la plus efficace, après plusieurs essais avec différents autres stratégies de recherche.

De plus, la méthode avancée applique une contrainte pour casser la symétrie diagonale : `variables[0][i] = variables[i][0]` pour tout $i \in [0, n-1]$. Cette contrainte force l'égalité entre la première ligne et la première colonne du carré, réduisant significativement l'espace de recherche sans éliminer les solutions valides du problème SBLS.

4 Résultats

4.1 Méthode simple

Après l'exécution de la méthode de résolution simple sur plusieurs n différents, on a obtenus les statistiques suivantes :

	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	0,206	?
Resolution Time (in s)	0,022	0,033	0,518	0,228	41,653	?
Nodes	2	7	485	76	10944	?
Nodes per second	89,6	215	936,4	333,9	262,7	?
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

Pour des valeurs de n supérieures où égales à 7, la méthode de résolution simple n'a pas produite de résultats, après plus de 30 minutes de temps passé.

4.2 Méthode avancée

	n=2	n=3	n=4	n=5	n=6	n=7
Solution Found	Yes	Yes	No	Yes	Yes	?
Building time (in s)	0,047	0,059	0,079	0,129	0,207	?
Resolution Time (in s)	0,015	0,025	0,158	0,071	2,708	?
Nodes	2	3	72	9	1191	?
Nodes per second	137,7	122,0	456,0	127,1	439,8	?
Backtracks	0	0	145	0	2352	?
Backjumps	0	0	0	0	0	?
Fails	0	0	73	0	1180	?
Restarts	0	0	0	0	0	?

Table 2: Résultats d'exécutions de la méthode de résolution avancée

Pour des valeurs de n supérieures où égales à 7, la méthode de résolution avancée n'a pas produite de résultats, après plus de 30 minutes de temps passé.

4.3 Analyse des résultats

Comme on peut le voir sur Table 1 et Table 2, la méthode de résolution avancée est plus efficace que la méthode de résolution simple.

En effet la méthode de résolution avancée résout en un temps de résolution inférieur ainsi que avec un nombre de nœuds et un nombre de fails et de backtracks inférieur, comparé à la méthode de résolution simple.

Par exemple, la méthode simple a trouvée pour n=6 une solution en environ 41 secondes, avec 10 944 nœuds, 10 842 fails et 21 671 backtracks. La méthode avancée a trouvée pour n=6

une solution en environ 2,7 secondes, avec 1 191 nœuds, 1 180 fails et 2352 backtracks, ce qui est considérablement inférieur à la méthode simple.

Cependant, à partir de la valeur 7 pour n, ni la méthode simple et ni la méthode avancée ne peuvent donner des résultats en un temps de résolution raisonnable.

5 Conclusion

Pour conclure, nous avons réussi à modéliser le problème SBLS (Spatially Balanced Latin Square) et nous avons réussi à le résoudre, pour certaines valeurs de n, grâce à la librairie Java Choco-solver.

Pour résoudre le problème SBLS, nous avons implémenté deux méthodes de résolutions, une simple et une avancée.

Comparée à la méthode simple, la méthode de résolution avancée précise des algorithmes de filtrage pour les contraintes sur les lignes et les colonnes, utilise une stratégie de recherche, et casse la symétrie du carré $n \times n$ avec une contrainte supplémentaire.

Ainsi, la méthode de résolution avancée est plus efficace que la méthode de résolution simple, comme on a pu voir dans la section 4.

Cependant, ni la méthode simple et ni la méthode avancée ne peuvent donner des résultats pour des valeurs de n supérieures ou égales à 7, en un temps de résolution raisonnable.