Université Libre de Bruxelles

Département d'Informatique

INFO-F302 Informatique fondamentale

Projet

Satisfactions de Contraintes et Utilisation de l'Outil *Choco solver*

Rapport

Dany Simone Efila – defilaef@ulb.ac.be
Olivier Pirson – opi@opimedia.be

26 mai 2017

Toutes les sources sur Bitbucket :
https://bitbucket.org/OPiMedia/chocochess/

Introduction

1 Problèmes d'échecs

Soit l'instance du problème $I = (n, k_1, k_2, k_3)$, avec $n \in \mathbb{N}_*$ et $k_1, k_2, k_3 \in \mathbb{N}$.
Il s'agit d'un échiquier de taille $n \times n$ qui doit contenir
- k_1 tours
- k_2 fous
- k_3 cavaliers.

Soit $N = \{1, 2, 3, \ldots, n\}$.

1.1 Question 1. Problème d'indépendance

Formulation CSP (Constraint Satisfaction Problem) du problème d'indépendance (aucune pièce ne peut en menacer une autre).

Variables

L'ensemble des variables correspond aux n^2 cases de l'échiquier :

$$X = \{x_{1,1},\ x_{1,2},\ x_{1,3},\ \ldots,\ x_{1,n},\ .\\ x_{2,1},\ x_{2,2},\ x_{2,3},\ \ldots,\ x_{2,n},\\ x_{3,1},\ x_{3,2},\ x_{3,3},\ \ldots,\ x_{3,n},\\ \ldots\\ x_{n,1},\ x_{n,2},\ x_{n,3},\ \ldots,\ x_{n,n}\}$$

Une variable $x_{i,j}$ correspondant à la case de la ligne i et de la colonne j de l'échiquier.

Domaines

$\forall i, j \in N$: le domaine associé à la variable $x_{i,j}$ est $D_{i,j} = \{0, 1, 2, 3\}$.
Toutes les variables ayant le même domaine nous le noterons D.
Pour chacune des cases :
- 0 représente une case vide
- 1 une case occupée par une tour
- 2 par un fou
- 3 par un cavalier.

Contraintes

- Contrainte assurant le bon nombre de tours :
  Le nombre de 1 sur l'échiquier vaut k_1.

$$c_{\#\text{tours}} = \left((x_{1,1}, x_{1,2}, x_{1,3}, \ldots, x_{n,n}), \left\{ (p_{1,1}, p_{1,2}, p_{1,3}, \ldots, p_{n,n}) \in D^{n^2} \middle| \sum_{\substack{i,j \in N \\ p_{i,j} = \mathbf{1}}} 1 = k_1 \right\} \right)$$

- Contrainte assurant le bon nombre de fous :
  Le nombre de 2 sur l'échiquier vaut k_2.

$$c_{\#\text{fous}} = \left((x_{1,1}, x_{1,2}, x_{1,3}, \ldots, x_{n,n}), \left\{ (p_{1,1}, p_{1,2}, p_{1,3}, \ldots, p_{n,n}) \in D^{n^2} \middle| \sum_{\substack{i,j \in N \\ p_{i,j} = \mathbf{2}}} 1 = k_2 \right\} \right)$$

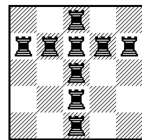
- Contrainte assurant le bon nombre de cavaliers :
  Le nombre de 3 sur l'échiquier vaut k_3.

$$c_{\#\text{cavaliers}} = \left((x_{1,1}, x_{1,2}, x_{1,3}, \ldots, x_{n,n}), \left\{ (p_{1,1}, p_{1,2}, p_{1,3}, \ldots, p_{n,n}) \in D^{n^2} \middle| \sum_{\substack{i,j \in N \\ p_{i,j} = \mathbf{3}}} 1 = k_3 \right\} \right)$$

Le problème d'indépendance impose que si une case est occupée par une pièce, alors chacune des cases menacées par cette pièce doit être vide.

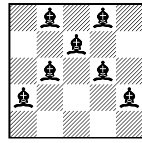
Remarquons que pour ce problème il n'est pas nécessaire de prendre en compte le fait que certaines pièces puissent en bloquer d'autre. En effet, si une pièce bloque la portée d'une autre c'est que l'instance considérée ne remplit pas le problème d'indépendance. Autrement dit ce problème (avec les vraies règles de mouvements du jeux d'échecs) reste strictement équivalent si on permet à chaque pièce de traverser les autres.

— Contrainte assurant qu'aucune tour ne menace les autres pièces [1] :
Si une tour occupe une case, les autres cases de la ligne sont vides, ainsi que les autres cases de la colonne.



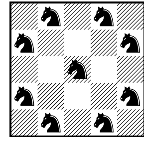
$$c_{\text{tours}} = \Big((x_{1,1}, x_{1,2}, x_{1,3}, \ldots, x_{n,n}), \Big\{(p_{1,1}, p_{1,2}, p_{1,3}, \ldots, p_{n,n}) \in D^{n^2} \Big| (p_{i,j} = \mathbf{1}) \longrightarrow \left(\bigwedge_{\substack{j' \in N \\ j' \neq j}} p_{i,j'} = 0 \right) \wedge \left(\bigwedge_{\substack{i' \in N \\ i' \neq i}} p_{i',j} = 0 \right) \Big\}\Big)$$

— Contrainte assurant qu'aucun fou ne menace les autres pièces :
Si un fou occupe une case, les autres cases des deux diagonales passant par cette case sont vides.



$$c_{\text{fous}} = \Big((x_{1,1}, x_{1,2}, x_{1,3}, \ldots, x_{n,n}), \Big\{(p_{1,1}, p_{1,2}, p_{1,3}, \ldots, p_{n,n}) \in D^{n^2} \Big| (p_{i,j} = \mathbf{2}) \longrightarrow \left(\bigwedge_{\substack{k \in -N \cup N \\ 1 \leq i+k \leq n \\ 1 \leq j+k \leq n}} p_{i+k,j+k} = 0 \right) \wedge \left(\bigwedge_{\substack{k \in -N \cup N \\ 1 \leq i+k \leq n \\ 1 \leq j-k \leq n}} p_{i+k,j-k} = 0 \right) \Big\}\Big)$$

— Contrainte assurant qu'aucun cavalier ne menace les autres pièces :
Si un cavalier occupe une case, les 8 cases possibles pour un mouvement de cavaliers sont vides.



$$c_{\text{cavaliers}} = \Big((x_{1,1}, x_{1,2}, x_{1,3}, \ldots, x_{n,n}), \Big\{(p_{1,1}, p_{1,2}, p_{1,3}, \ldots, p_{n,n}) \in D^{n^2} \Big| (p_{i,j} = \mathbf{3}) \longrightarrow \left(\bigwedge_{\substack{k \in \{-1,1\} \\ l \in \{-2,2\} \\ 1 \leq i+k \leq n \\ 1 \leq j+l \leq n}} p_{i+k,j+l} = 0 \right) \wedge \left(\bigwedge_{\substack{k \in \{-1,1\} \\ l \in \{-2,2\} \\ 1 \leq i+l \leq n \\ 1 \leq j+k \leq n}} p_{i+l,j+k} = 0 \right) \Big\}\Big)$$

Ce qui donne l'ensemble de contraintes
$C_1 = \{c_{\#\text{tours}}\} \cup \{c_{\#\text{fous}}\} \cup \{c_{\#\text{cavaliers}}\} \cup \{c_{\text{tours}}\} \cup \{c_{\text{fous}}\} \cup \{c_{\text{cavaliers}}\}$

1.2 Question 2. Problème de domination

Formulation CSP du problème de domination (chaque case est soit occupée, soit menacée par une pièce).

Les **variables** et leurs **domaines** associés sont exactement les mêmes que pour le problème précédent.

Contraintes

Les contraintes imposant le nombre de pièces aussi restent exactement les mêmes : $\{c_{\#\text{tours}}\}$, $\{c_{\#\text{fous}}\}$ et $\{c_{\#\text{cavaliers}}\}$.

1. Les éventuelles autres tours comprises.

Ce problème impose que si une case est vide alors il doit exister une pièce qui la menace.

Cela nous oblige maintenant à prendre en considération le fait que certaine pièce peuvent bloquer l'action d'autre pièce. C'est-à-dire que pour qu'une pièce en menace une autre, les cases sur son chemin doivent être vide.

Si une case est vide alors soit une tour doit être sur la même ligne ou colonne sans autre pièce s'interposant, soit un fou doit être sur une même diagonale sans autre pièce s'interposant, ou soit un cavalier doit être sur une des 8 cases possibles la menaçant [2].

$$
\begin{aligned}
c_{\text{menace}} = \big(&(x_{1,1}, x_{1,2}, x_{1,3}, \dots, x_{n,n}),\\
&\big\{(p_{1,1}, p_{1,2}, p_{1,3}, \dots, p_{n,n}) \in D^{n^2} \,\big|\\
&(p_{i,j} = 0) \longrightarrow \bigvee_{\substack{1 \leq j' \leq n \\ j' \neq j}} (p_{i,j'} = \mathbf{1}) \wedge \left(\bigwedge_{\min(j,j') < j'' < \max(j,j')} p_{i,j''} = 0 \right)\\
&\qquad \vee \bigvee_{\substack{1 \leq i' \leq n \\ i' \neq i}} (p_{i',j} = \mathbf{1}) \wedge \left(\bigwedge_{\min(i,i') < i'' < \max(i,i')} p_{i'',j} = 0 \right)\\
&\qquad \vee \bigvee_{\substack{k \in -N \cup N \\ 1 \leq i+k \leq n \\ 1 \leq j+k \leq n}} (p_{i+k,j+k} = \mathbf{2}) \wedge \left(\bigwedge_{\substack{\min(i,i+k) < i' < \max(i,i+k) \\ \min(j,j+k) < j' < \max(j,j+k)}} p_{i',j'} = 0 \right)\\
&\qquad \vee \bigvee_{\substack{k \in -N \cup N \\ 1 \leq i+k \leq n \\ 1 \leq j-k \leq n}} (p_{i+k,j-k} = \mathbf{2}) \wedge \left(\bigwedge_{\substack{\min(i,i+k) < i' < \max(i,i+k) \\ \min(j,j-k) < j' < \max(j,j-k)}} p_{i',j'} = 0 \right)\\
&\qquad \vee \left(\bigvee_{\substack{k \in \{-1,1\} \\ l \in \{-2,2\} \\ 1 \leq i+k \leq n \\ 1 \leq j+l \leq n}} p_{i+k,j+l} = \mathbf{3} \right)\\
&\qquad \vee \left(\bigvee_{\substack{k \in \{-1,1\} \\ l \in \{-2,2\} \\ 1 \leq i+l \leq n \\ 1 \leq j+k \leq n}} p_{i+l,j+k} = \mathbf{3} \right) \big\}\big)
\end{aligned}
$$

Ce qui donne l'ensemble de contraintes
$C_2 = \{c_{\#\text{tours}}\} \cup \{c_{\#\text{fous}}\} \cup \{c_{\#\text{cavaliers}}\} \cup \{c_{\text{menace}}\}$

1.3 Question 3. Implémentation *ChocoChess*

Le répertoire `src/doc/html/` contient la JavaDoc [3] de l'implémentation.

En annexe page 11 des diagrammes de classes synthétisent les attributs et méthodes de chaque classes.

La classe principale qui constitue le programme est `ChocoChess`.

2. Les cavaliers ne sont jamais bloqués par d'autres pièces.
3. Si ce document PDF est préservé dans la hiérarchie des répertoires fournie alors ce genre de lien permet d'ouvrir directement la documentation HTML.

C'est également cette classe qui contient la méthode principale du projet `modelAndSolve()`, qui initialise *Choco solver* avec les paramètres des problèmes, lance la résolution et enfin traite les résultats.

Toujours de cette classe les méthodes `setConstraintsIndependence()` et `setConstraintsDomination()` initialise les contraintes spécifiques aux deux problèmes.

Faire attention à ce que les divers indices dans le programme commencent à 0 et non pas à 1 comme dans les formulations mathématiques.

1.3.1 Question bonus. Généralisation

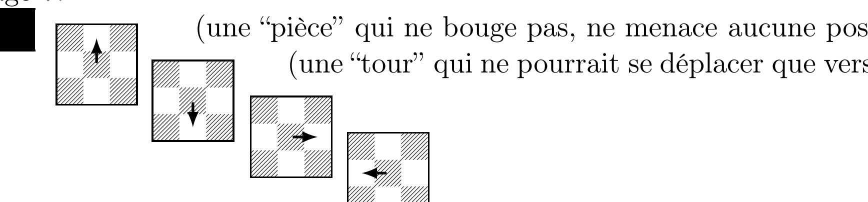
Conformément à la question bonus, la gestion des pièces a été généralisée. Une méthode `add()` appelle les méthodes correspondantes à chaque pièce. Méthodes qui définissent pour chaque case, pour chaque pièce, l'ensemble `DominatedPositions` des positions que cette pièce menace potentiellement [4]. À chacune de ces positions menacées `DominatedPosition` est associé l'ensemble des positions qui doivent être vides pour ne pas bloquer son action sur cette position.

La méthode `setConstraintsIndependence()` utilise uniquement ce premier ensemble pour construire les contraintes à passer au solveur.

La méthode `setConstraintsDomination()` utilise les deux ensembles.

Cette généralisation a permis d'ajouter les pièces suivantes : [5]
— reine (se définissant facilement comme l'union d'une tour et d'un fou)
— roi (juste 8 positions, sans problème de blocage)

On été également ajouté 5 pièces correspondant aux éléments du problème de surveillance de musée développé page 7.
— obstacle (une "pièce" qui ne bouge pas, ne menace aucune position)
— "tour" nord (une "tour" qui ne pourrait se déplacer que vers le haut)
— "tour" sud
— "tour" est
— "tour" ouest



1.3.2 Exécution

Le programme a été testé avec la version 1.8 de Java et la version 4.0.4 de *Choco solver*, sous *Debian Jessie* et *Window$ 10*.

Le programme compilé se trouve dans l'archive JAR `ChocoChess.jar`. Il est dépendant de l'archive JAR `choco-solver-4.0.4-with-dependencies.jar` qui doit se trouver dans le répertoire `lib/`.

Un `Makefile` est fourni permettant de le recompiler à partir des sources disponibles dans le répertoire `src/`.

4 scripts shell sont également fournis servant de raccourci pour appeler le programme sur les divers problèmes.

Un résumé des options [6] tels qu'affiché par la commande
\>>> `java -jar ChocoChess.jar -help`

```
Usage: java -jar ChocoChess.jar [options]
  -[i|d]: Independence problem (by default) or Domination problem
  -m n: number of chessboard lines (same of columns by default)
  -n n: number of chessboard column (8 by default)
```

4. En l'absence d'autres pièces qui pourraient bloquer son action.
5. Les seules pièces restantes du jeu d'échecs, les pions, ont un comportement dépendant notamment de l'orientation de l'échiquier. Nous les avons ignorés.
6. Dont celles définies pour les problèmes développés dans les pages suivantes.

```
-minmax: minimize number of not null pieces for Independence problem
         / maximize number of not null pieces for Domination problem
         (instead find exact number of pieces)

-t n: number of rooks   (Tours)     (0 by default)
-f n: number of bishops (Fous)      (0 by default)
-c n: number of knights (Cavaliers) (0 by default)
-r n: number of queens  (Reines)    (0 by default)
-k n: number of Kings   (rois)      (0 by default)

-o n: number of obstacles (0 by default)
-tn n: number of North-"rooks" ("Tours" Nord)  (0 by default)
-ts n: number of South-"rooks" ("Tours" Sud)   (0 by default)
-te n: number of East-"rooks"  ("Tours" Est)   (0 by default)
-to n: number of West-"rooks"  ("Tours" Ouest) (0 by default)

-load FILENAME: load initial configuration in chess or museum format
-load -: load initial configuration from standard input stream
-museum: museum format to load file and display solution(s)

-all: find all solutions instead one solution (only if not -minmax)
-help: display this message to standard error stream and quit
-latex: display also solution(s) in LaTeX format for the chessboard package
-symmetrics-optimizations: enable some symmetrics optimizations
                           (requires m = n and no initial configuration)
-verbose: display some informations to standard error stream

-debug-pos: display dominated positions to standard error stream
```

L'option `-all` affiche successivement toutes les solutions possibles. En son absence le programme s'arrête dès une première solution trouvée, comme demandé.

L'option `-symmetrics-optimizations` impose une pièce dans le quart supérieur gauche de l'échiquier. Cela réduit le nombre de solutions possibles, éliminant certaines solutions identiques à une symétrie près. Ce qui peut accélérer la recherche.

Exemples d'utilisation sur les exemples de l'énoncé.

1.3.3 Exemples – Problème d'indépendance

Exemple 1 de la page 1 de l'énoncé : [7]

```
>>> java -jar ChocoChess.jar -i -n 3 -t 1 -f 1 -c 1
```
ou
```
>>> ./ChocoChess.sh -n 3 -t 1 -f 1 -c 1
```

```
C F *
* * *
* * T
```

Exemple de la page 2 de l'énoncé :

```
>>> java -jar ChocoChess.jar -i -n 4 -t 2 -f 1 -c 2
```

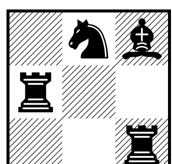
```
T * * *
* T * *
* * C *
* * F C
```

"Vous ne passerez pas !" (Gandalf)

(Remarquons que cette solution est aussi une solution pour le problème de domination.)

7. La solution différente présentée dans l'énoncé est la solution n°13 dans la liste de toutes les solutions obtenues avec l'option `-all`.

1.3.4 Exemple – Problème de domination



Exemple 2 de la page 1 de l'énoncé : [8]

```
>>> java -jar ChocoChess.jar -d -n 3 -t 2 -f 1 -c 1
```

ou

```
>>> ./domination.sh -n 3 -t 2 -f 1 -c 1
```

```
* C F
T * *
* * T
```

1.4 Question 4. Minimisation du nombre de cavaliers nécessaires pour dominer

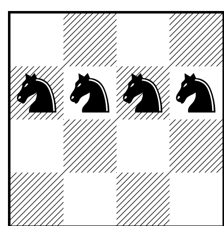
Il s'agit simplement d'un problème de domination avec un nombre indéterminé de cavaliers. En ajoutant une contrainte de minimisation sur ce nombre, le solveur va chercher les solutions au problème de domination en réduisant progressivement le nombre de cavaliers jusqu'à en trouver le minimum.

C'est le même programme qui gère cela, avec l'option `-minmax`.

Pour faciliter l'implémentation de ce problème, aux variables nécessaires pour les deux problèmes précédents nous avons ajouté une variable pour chaque pièce (et pour les cases vides) qui comptabilise leur nombre d'occurrences. Ainsi il suffit d'indiquer au solveur l'objectif demandé. Ce qui se fait en une ligne dans notre méthode `modelAndSolve()`.

Ici aussi nous avons généralisé le problème. Le programme peut prendre en compte n'importe quelle type de pièces, isolée ou plusieurs d'entre-elles. C'est la raison pour laquelle plutôt que de minimiser le nombre de pièces considérées nous avons maximisé le nombre de cases vides.

Exemple de la page 2 de l'énoncé : [9]



```
>>> java -jar ChocoChess.jar -d -minmax -c 1 -n 4
```

ou

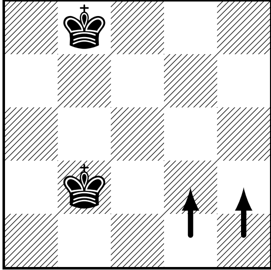
```
>>> ./domination_min_knights.sh -n 4
```

```
4
* * * *
C C C C
* * * *
* * * *
```

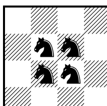
Exemple avec d'autres types de pièces posssibles (toute pièce dont le nombre mentionné est non nul est une pièce disponible pour minimiser le nombre de pièces nécessaires à la domination) :

```
>>> java -jar ChocoChess.jar -d -minmax -c 1 -f 1 -k 1 -tn 1 -n 5
```

En autorisant des cavaliers, fous, rois et "tours" nord le solveur trouve une solution optimale en n'utilisant que 2 rois et 2 "tours" nord : 4 pièces



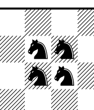
8. La solution différente présentée dans l'énoncé est la solution n°232 dans la liste de toutes les solutions obtenues avec l'option `-all`.



9. Avec ce genre de commande on peut forcer la position de deux cavaliers :

```
>>> echo $'* * * *\n* C * *\n* * C *\n* * * *'
    | java -jar ChocoChess.jar -d -minmax -c 1 -load -
```

La solution est alors identique à celle présentée dans l'énoncé.

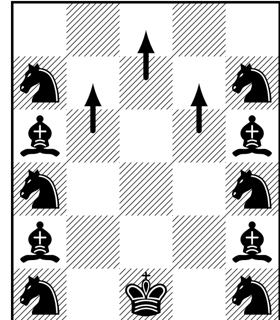


Nous avons de plus généralisé l'optimisation au problème d'indépendance. C'est un problème symétrique.[10] Pour celui-ci l'objectif est de maximiser le nombre de pièces sur l'échiquier tant qu'elles ne se menacent pas entre-elles (en fait le programme minimise le nombre de cases vides).

Exemple de maximisation tout en restant indépendant :

```
>>> java -jar ChocoChess.jar -i -minmax -c 1 -f 1 -k 1 -tn 1 -n 5
```

14 pièces[11]



2 Question 5. Surveillance de musée

Remarquons que ce problème est similaire à un problème de minimisation du nombre de tours nécessaires à la domination d'un échiquier. Avec la particularité que certaines cases de l'échiquier sont initialement occupées par un obstacle (qui n'a aucune autre action que de celle d'occuper une case), et qu'à la place de tours proprement dite il y a quatre types de "tours", chacune ne pouvant se déplacer que dans une seule direction. Ces 5 types de pièces ont été ajoutées à notre programme général comme expliqué page 4.

Soit l'instance du problème $I = (m, n, O)$, avec $m, n \in \mathbb{N}_*$, $M = \{1, 2, 3, \ldots, m\}$, $N = \{1, 2, 3, \ldots, n\}$ et $O \subseteq M \times N$.

Il s'agit d'un musée de taille $m \times n$ qui contient un ensemble d'obstacles[12] prédéterminé représenté par l'ensemble O des couples de coordonnées.

2.1 Formulation CSP du problème

Variables

L'ensemble des variables correspond aux mn cases du musée :

$$X = \{ \begin{array}{l} x_{1,1},\ x_{1,2},\ x_{1,3},\ \ldots,\ x_{1,n},\ . \\ x_{2,1},\ x_{2,2},\ x_{2,3},\ \ldots,\ x_{2,n}, \\ x_{3,1},\ x_{3,2},\ x_{3,3},\ \ldots,\ x_{3,n}, \\ \ldots \\ x_{m,1}, x_{m,2}, x_{m,3}, \ldots, x_{m,n}\} \end{array}$$

Une variable $x_{i,j}$ correspondant à la case de la ligne i et de la colonne j du musée.

Domaines

$\forall i \in M, \forall j \in N$: le domaine associé à la variable $x_{i,j}$ est $D_{i,j} = \{0, 1, 2, 3, 4, 5\}$.

Chacune des variables ayant le même domaine nous le noterons D.

Pour chacune des cases :

— 0 représente une case vide

— 1 une case occupée par un obstacle

— 2 par un capteur pointant vers le nord (appelé "tour" nord dans notre programme général)

10. C'est la raison pour laquelle l'option commune est nommée `-minmax`.

11. La ligne du haut n'est présente que pour permettre au paquetage LaTeX `chessboard` de représenter la flèche de la "tour" nord.

12. Des obstacles proprement dit `*` ■ , et non des capteurs.

— 3 par un capteur pointant vers le sud
— 4 par un capteur pointant vers l'est
— 5 par un capteur pointant vers l'ouest.

Contraintes

— Contrainte assurant un nombre minimum de capteurs :
$$c_{\min} = \Big((x_{1,1}, x_{1,2}, x_{1,3}, \ldots, x_{m,n}), \Big\{(p_{1,1}, p_{1,2}, p_{1,3}, \ldots, p_{m,n}) \in D^{mn} \Big| \sum_{\substack{i\in M\\ j\in N\\ p_{i,j}\geq 2}} 1 = \min_{(p'_{1,1},p'_{1,2},\ldots,p'_{m,n})\in D^{mn}} \sum_{\substack{i\in M\\ j\in N\\ p'_{i,j}\geq 2}} 1\Big\}\Big)$$

— Contrainte fixant les cases occupées par un obstacle :
$$c_O = \Big((x_{1,1}, x_{1,2}, x_{1,3}, \ldots, x_{m,n}), \Big\{(p_{1,1}, p_{1,2}, p_{1,3}, \ldots, p_{m,n}) \in D^{mn} \Big| (i,j) \in O \leftrightarrow p_{i,j} = 1\Big\}\Big)$$

— Contrainte assurant que chaque case est surveillée par au moins un capteur :
Si une case est vide alors soit un capteur Nord doit être sur la même colonne en-dessous sans obstacle s'interposant, soit un capteur Sud doit être sur la même colonne au-dessus sans obstacle s'interposant, soit un capteur Est doit être sur la même ligne à gauche sans obstacle s'interposant ou soit un capteur Ouest doit être sur la même ligne à droite sans obstacle s'interposant.
$$\begin{aligned} c_{\text{surveille}} = \Big(&(x_{1,1}, x_{1,2}, x_{1,3}, \ldots, x_{m,n}), \\ &\Big\{(p_{1,1}, p_{1,2}, p_{1,3}, \ldots, p_{m,n}) \in D^{mn} \Big| \\ &(p_{i,j} = 0) \longrightarrow \bigvee_{i<i'\leq m} (p_{i',j} = \mathbf{2}) \wedge \Big(\bigwedge_{i<i''<i'} p_{i'',j} = 0\Big) \\ &\quad \vee \bigvee_{1\leq i'<i} (p_{i',j} = \mathbf{3}) \wedge \Big(\bigwedge_{i'<i''<i} p_{i'',j} = 0\Big) \\ &\quad \vee \bigvee_{1\leq j'<j} (p_{i,j'} = \mathbf{4}) \wedge \Big(\bigwedge_{j'<j''<j} p_{i,j''} = 0\Big) \\ &\quad \vee \bigvee_{j<j'\leq n} (p_{i,j'} = \mathbf{5}) \wedge \Big(\bigwedge_{j<j''<j'} p_{i,j''} = 0\Big) \Big\}\Big) \end{aligned}$$

Ce qui donne l'ensemble de contraintes
$C_3 = \{c_{\min}\} \cup \{c_O\} \cup \{c_{\text{surveille}}\}$

2.2 Implémentation

Pour permettre à notre programme de traiter ce problème nous avons dû (encore) le généraliser de deux manières. En scindant les deux dimensions n^2 en m et n qui peuvent être différentes (via l'option `-m n`).

Et en permettant de fixer une configuration de départ à partir d'un fichier (via l'option `-load FILENAME`, ou à partir de l'entrée standard via l'option `-load -`).

L'option `-museum` adapte le format d'entrée/sortie à celui demandé pour ce problème de surveillance de musée.

À noter que nous avons privilégié une complète généralisation et symbiose de tous les problèmes plutôt que de tenter d'optimiser chacun. À noter aussi que ces problèmes sont en général intraitables sur de grandes tailles.

Exemple de la page 3 de l'énoncé :

```
>>> java -jar ChocoChess.jar -d -museum -minmax -tn 1 -ts 1 -te 1 -to 1 -load
    data/museum_7x9.txt
```

ou

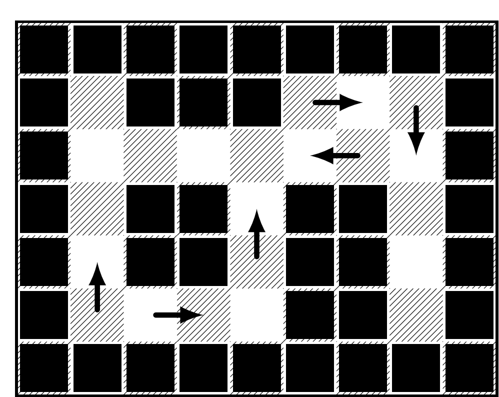
```
>>> ./museum.sh data/museum_7x9.txt
```

Configuration de départ :

```
* * * * * * * * *
*   * * *       *
*               *
*   * *   * *   *
*   * *   * *   *
*         * *   *
* * * * * * * * *
```

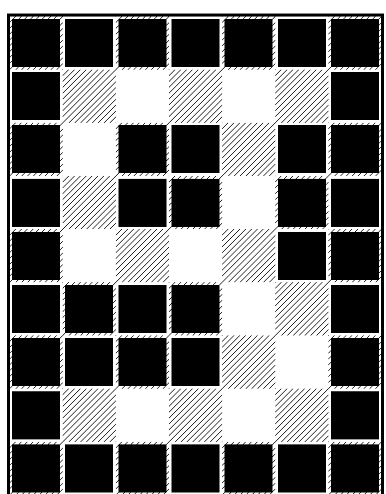
Solution :

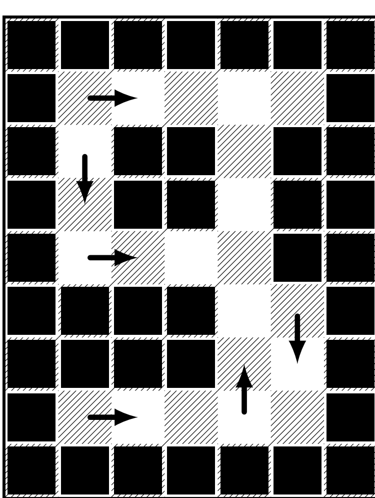
```
6
* * * * * * * * *
*   * * * E   S *
*           O   *
*   * *   * *   *
*   * * N * *   *
* N E     * *   *
* * * * * * * * *
```



Autre exemple, avec la même configuration de départ mais tourné de 90° (sens antihoraire) :

```
>>> java -jar ChocoChess.jar -d -museum -minmax -tn 1 -ts 1 -te 1 -to 1 -load
data/museum_9x7.txt
```



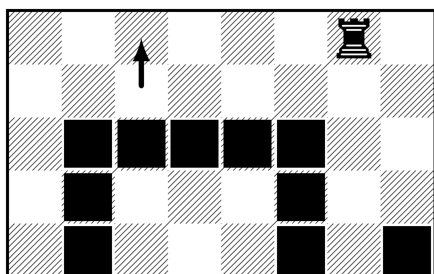


Remarquons que le solveur trouve une autre solution que la solution précédente tournée de 90°.

3 Exemple général

Deux exemples d'optimisations, à partir de la même configuration de départ.

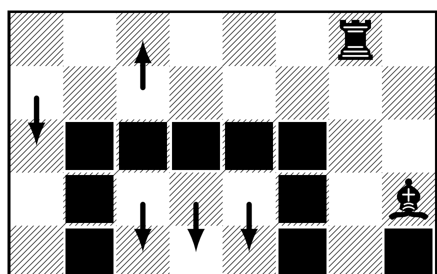
Configuration de départ de 12 pièces :



Problème d'indépendance, maximisation :

```
>>> java -jar ChocoChess.jar -i -minmax -f 1 -c 1 -k 1 -ts 1 -load
    data/general_5x8.txt
```

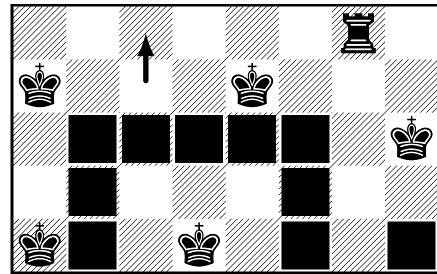
7 pièces en dehors des obstacles (qui n'ont aucune action, mais ne peuvent tout de même pas être menacés), total de 17 pièces



Problème de domination, minimisation :

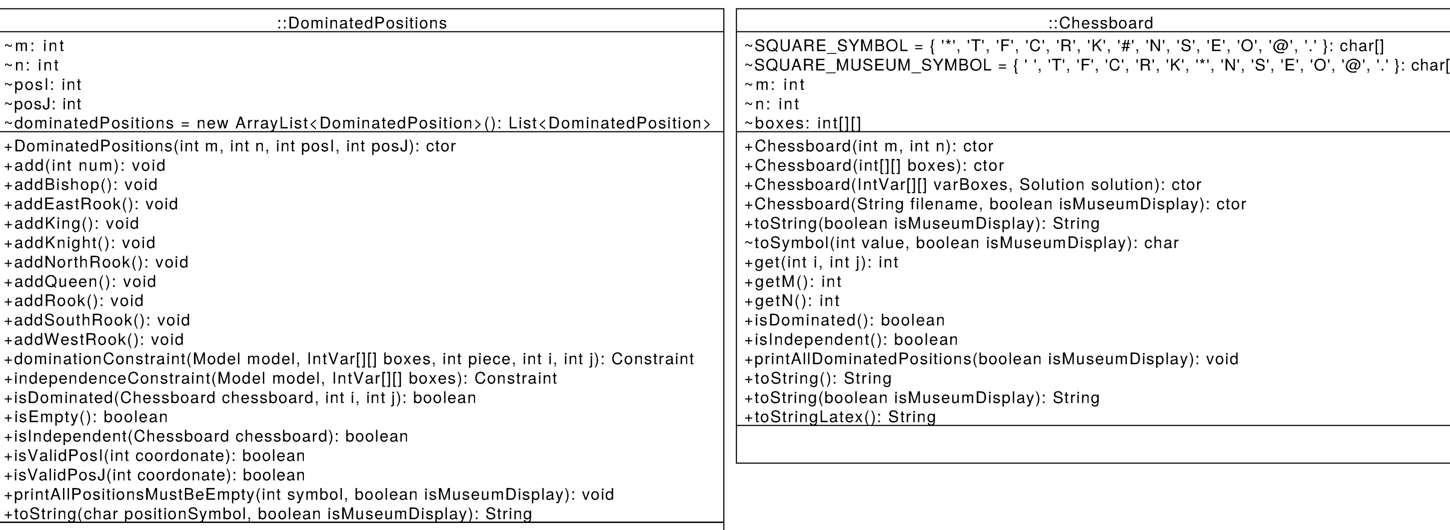
```
>>> java -jar ChocoChess.jar -d -minmax -f 1 -c 1 -k 1 -ts 1 -load
    data/general_5x8.txt
```

7 pièces en dehors des obstacles, total de 17 pièces



Annexe

Diagrammes de classe de *ChocoChess*



::DominatedPosition

~dominatedPosition: Position
~positionsMustBeEmpty = new ArrayList<Position>(): List<Position>

+DominatedPosition(int posI, int posJ): ctor
+addPositionsMustBeEmpty(Position position): void
+getPosI(): int
+getPosJ(): int
+getPositionsMustBeEmpty(): List<Position>
+toString(): String

::Position

~posI: int
~posJ: int

+Position(int posI, int posJ): ctor
+getPosI(): int
+getPosJ(): int
+toString(): String

Table des matières

Introduction **1**

1 Problèmes d'échecs **1**
1.1 Question 1. Problème d'indépendance . . . 1
1.2 Question 2. Problème de domination . . . 2
1.3 Question 3. Implémentation *ChocoChess* . . . 3
1.3.1 Question bonus. Généralisation . . . 4
1.3.2 Exécution . . . 4
1.3.3 Exemples – Problème d'indépendance . . . 5
1.3.4 Exemple – Problème de domination . . . 6
1.4 Question 4. Minimisation du nombre de cavaliers nécessaires pour dominer . . . 6

2 Question 5. Surveillance de musée **7**
2.1 Formulation CSP du problème . . . 7
2.2 Implémentation . . . 8

3 Exemple général **10**

Annexe **11**
Diagrammes de classe de *ChocoChess* . . . 11

FIGURE 1 – *Chocolate checkmate your father this June* [13]

13. http://pressreleases.responsesource.com/news/31174/chocolate-checkmate-your-father-this-june/