Le problème des n reines

Arnaud Malapert

11 octobre 2012

Le problème des huit reines est un bon exemple de problème simple mais non trivial. Pour cette raison, il est souvent employé comme support de mise en œuvre de techniques de programmation.

Définition 1 (Problème des n reines)**.** *Le problème des n reines consiste à placer n reines sur un échiquier de taille $n \times n$ sans que les reines ne soient en prise. Conformément aux règles du jeu d'échecs (la couleur des pièces étant ignorée), deux reines ne devraient jamais partager la même ligne, colonne, ou diagonale.*

Vous pouvez vérifier que la Figure 1 montre une solution du problème des 8 reines

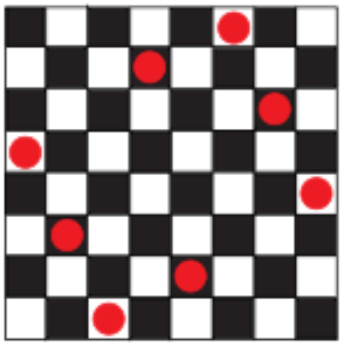

FIGURE 1 – Une solution du problème des 8 reines.

Exercice 1 Problème des 4 reines

1. Trouvez une solution à la main au problème des 4 reines.
2. Trouvez d'autres solutions en appliquant des transformations (rotations, réflexions) à la solution.
3. Combien de solutions admet le problème des 4 reines ?

Algorithmes classiques Ce problème peut être résolu avec un algorithme récursif. Une solution du problème des n reines est obtenue par récurrence à partir d'une solution quelconque du problème des $(n-1)$ reines par l'adjonction d'une reine. La récurrence commence avec la solution du problème avec 0 reine qui repose sur un échiquier vide.
Cette technique est beaucoup plus efficace que l'algorithme naïf de recherche exhaustive, qui parcourt chacun des $64^8 = 2^{48} = 281\ 474\ 976\ 710\ 656$ placements possibles des huit reines, pour retirer tous ceux pour lesquels plusieurs reines se trouvent sur une même case (laissant seulement $A_{64}^8 = 64!/56! = 178\ 462\ 987\ 637\ 760$ arrangements possibles) ou pour lesquels des reines se menacent mutuellement.
Ce très « mauvais » algorithme produira les mêmes résultats à plusieurs reprises en attribuant différentes places aux huit reines, et recommencera les mêmes calculs plusieurs fois pour différentes parties de chaque solution. Un algorithme légèrement meilleur de recherche exhaustive place une

seule reine par ligne, réduisant à seulement $8^8 = 2^{24} = 16\ 777\ 216$ placements possibles.
Il est possible de faire beaucoup mieux que cela. Par exemple, un programme de recherche en profondeur examinerait seulement 15 720 placements possibles des reines en construisant un arbre de recherche et en parcourant les lignes de l'échiquier une par une, éliminant la plupart des positions possibles à un stade très primitif de leur construction.
Remarquez qu'on peut trouver une solution du problème des n reines en temps polynomial. Ces algorithmes permettent généralement de générer uniquement des solutions appartenant certaines classes. Ainsi, certaines solutions ne peuvent jamais être générées par ces méthodes
La programmation par contraintes est une approche simple et efficace pour résoudre ce problème.

Un Modèle Naïf en PPC Nous allons tout d'abord discuter d'un modèle naïf basé sur des variables L_i et C_i associées à chaque reine i ($i = 1, \ldots, n$).

- L_i indique la *ligne* sur laquelle la *reine* i est placée.
- C_i indique la *colonne* sur laquelle la *reine* i est placée.

Les contraintes (1) et (2) imposent respectivement que les reines soient placées sur des lignes et colonnes différentes. Les contraintes (3) et (4) imposent que les reines ne soient pas placées sur la même diagonale.

$$L_i \neq L_j \qquad \forall\ 1 \leq i < j \leq n \tag{1}$$
$$C_i \neq C_j \qquad \forall\ 1 \leq i < j \leq n \tag{2}$$
$$L_i + (C_j - C_i) \neq L_j \qquad \forall\ 1 \leq i < j \leq n \tag{3}$$
$$C_i + (L_j - L_i) \neq C_j \qquad \forall\ 1 \leq i < j \leq n \tag{4}$$

On peut remarquer que deux reines quelconques ne peuvent être placées sur une même ligne. En se basant sur cette observation, *on peut supposer que la i-ème reine est sur la ligne i*. Nous allons voir dans les exercices 2 et 3 comment exploiter cette observation pour améliorer le modèle.

Remarques sur la résolution d'un CSP Les méthodes de résolution des CSPs sont génériques, c'est-à-dire qu'elles ne dépendent pas de l'instance à résoudre. Dans notre cas, la résolution est basée sur la réduction de l'espace de recherche par des techniques de consistance couplée si nécessaire à un algorithme de recherche arborescente. Pour ce premier TP, nous utiliserons une recherche arborescente lexicographique. À chaque point de choix, l'affectation partielle courante est étendue en instanciant les variables et les valeurs selon l'ordre lexicographique. L'intérêt de cette approche statique est qu'elle ne dépend ni du modèle, ni du filtrage des contraintes.

Remarques générales **Consultez toujours le site web et la documentation de Choco avant de poser une question. Vous pouvez aussi consulter mes notes en français sur le modèle de projet utilisé dans les TPs.**

Exercice 2 Modèle « Lignes »

Sans perte de généralité, on suppose maintenant que la reine i est sur la ligne i ($L_i = i$). Le modèle « Lignes » est uniquement basé sur les variables C_i (variables `queens`).
Indices : `makeIntVarArray` (variables), `eq`, `neq` (contraintes), `plus` `minus` (expressions)

1. *(Implémentation)* Posez les contraintes suivantes (inspirez vous du modèle naïf).
   a) Les reines sont sur des colonnes différentes.
   b) Les reines sont sur des diagonales différentes.

2. *(Observations)* Lancez le programme, changez le niveau de verbosité du solveur et analysez la trace du programme.

```
mvn assembly:single
java -jar target/nqueens-0.0.1-SNAPSHOT-jar-with-dependencies.jar -v SEARCH \
          src/main/resources/instances/n004.txt
```

3. *(Validation)* Validez (partiellement) le modèle en vérifiant plusieurs résultats connus.
   a) Le problème des 3 reines n'a pas de solution.
   b) Le problème des 4 reines a 2 solutions.
   c) Le problème des 8 reines a 92 solutions.
4. *(Évaluation)* Un modèle est évalué en terme de temps de résolution, nombre de nœuds, nombre de backtracks et nombre d'échecs.
   a) Évaluez la recherche d'*une solution* avec $n = 15, 20, 25$.
   b) Évaluez la recherche de *toutes les solutions* avec $n = 8, 10, 12$.
5. *(Contraintes globales)* Nous allons remplacer les contraintes inégalités binaires `neq` par des contraintes globales `alldifferent`. Pour cela, nous allons introduire les variables intermédiaires m_i et p_i qui représentent les projections de la reine i sur la colonne 0 selon les deux diagonales.
$$m_i = c_i - i \qquad p_i = c_i + i \qquad \forall\ 1 \leq i \leq n \tag{5}$$
   a) Intégrez les variables m_i et p_i dans le modèle avec les contraintes d'inégalité binaires.
   b) Remplacez les contraintes d'inégalité binaires par des contraintes `alldifferent`.
6. *(Comparaison)* Nous allons comparez les performances de différents modèles.
   a) Évaluez le modèle avec contraintes globales.
   b) Comparez les résultats obtenus par le modèle avec et sans contraintes globales.
   c) Évaluez le modèle avec contraintes globales en ajoutant l'option `C_ALLDIFFERENT_BC` ou `C_ALLDIFFERENT_AC` à `alldifferent`. Lorsqu'aucune option n'est spécifiée, le solveur choisit le niveau de consistance en appliquant une heuristique utilisant le nombre de variables ainsi que les types et tailles de domaines.

Je vous conseille de noter les résultats obtenus dans un tableau suivant ce modèle :

modèle	time (s)	nodes	backtracks	failures
		$n =$? (# solutions)		
`algo 1`		...		
`algo 2`		...		
		$n =$? (# solutions)		
`algo 1`		...		
`algo 2`		...		

Dorénavant, vous avez tout intérêt à utiliser la classe `NQueensSettings` pour paramétrer l'algorithme à l'aide d'un fichier `.properties`. Vous trouverez les instances et les fichiers `.properties` du TP1 dans l'arborescence du projet. Les fichiers `.properties` suivent la convention de nommage du tableau 1 (qui anticipe sur le contenu des TPs suivants). Vous trouverez aussi deux scripts *shell* qui vous aideront peut-être à réaliser les évaluations.

```
[nono@arrakis nqueens]$ tree src/main/resources/
src/main/resources/
|-- instances
|   |-- n003.txt
|   |-- n004.txt
|   |-- ...
|   |-- n075.txt
|   `-- n100.txt
|-- make-results.sh
|-- run-benchmarks.sh
`-- tp1
    |-- PA_PL.properties
    |-- PB_PL.properties
    |-- PN_PL.properties
    `-- PS_PL.properties
```

Exercice 3 Modèle « Lignes – Colonnes »

Nous allons maintenant proposer un second modèle dans lequel nous allons (ré-)introduire les variables L_i (variables `queensdual`).

C_i indique la *colonne* de la *reine* placée *sur la ligne* i.
L_i indique la *ligne* de la *reine* placée *sur la colonne* i.

Les variables L_i et C_i définissent la même solution et sont soumises à des *contraintes de liaison* :

$$L_i = j \Leftrightarrow C_j = i \qquad \forall\ 1 \leq i \leq n$$

Indice : `InverseChanneling`

1. Implémentez, validez et évaluez ce modèle (c.f exercice 2).
2. Comparez les résultats obtenus à ceux de l'exercice 2.
3. Détaillez et expliquez les résultats marquants obtenus pendant toutes ces expériences.

Modèle	`P` – Modèle primal (variables C_i) `A` – Modèle primal-dual (variables C_i et L_i)
Contraintes de différences	`N` – Contraintes `neq` `A` – Contraintes `allDifferent` avec consistance d'arcs `B` – Contraintes `allDifferent` avec consistance de bornes `S` – Contraintes `allDifferent` avec consistance automatique
Élimination des symétries	`_` – Absent `S` – Présent
Variables de décision	`P` – Variables primales `D` – Variables primales et duales
Stratégie de branchement	`L` – *lexicographic search* `R` – *random search* `D` – *minDomain (minVal)* `W` – *dom/wdeg* `C` – *dual-dom*

TABLE 1 – Algorithmes candidats pour le problème des n reines. Chaque acronyme spécifie un algorithme. Par défaut, le programme executera l'algorithme PN_PL.