

L'objectif de ce sujet est d'écrire des algorithmes de résolution par backtracking de problèmes combinatoires classiques. Commencez par récupérer le squelette de code sur le gitlab du cours <https://gitlabinfo.iutmontp.univ-montp2.fr/r5.a.04-qualite-algorithmique/>. Notez que les tests fournis ne sont *pas* suffisants, ils constituent une première étape de validation. Pensez donc à les compléter, et/ou à écrire un main avec quelques affichages pour contrôler ce qui se passe!

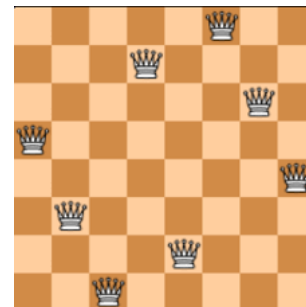
Ne lisez pas directement le squelette, mais plutôt ce sujet qui vous guidera. En général, le sujet vous fera d'abord compléter les classes `PartialSol*` qui représentent les solutions partielles, puis les classes `Algos*` qui contiendront les différentes variantes des algorithmes de backtracks.

## 1 Rappels

Commencez par lire le main de `AlgosNQueen` dans lequel vous trouverez des rappels sur les `Map` et les `ArrayList`.

## 2 Problème des $n$ -reines

On considère le problème vu en cours des  $n$ -reines dans lequel, étant donné un échiquier de taille  $n \times n$ , le but est de placer  $n$  reines de telle sorte qu'aucune reine ne puisse en manger une autre. On rappelle qu'aux échecs, les reines peuvent se manger dès qu'elles sont sur la même ligne, colonne, ou diagonale. On peut voir ci-contre une solution pour  $n = 8$ .





### Exercice 1. BackTrackQueenV0 et BackTrackQueenV1.

Dans cette partie l'objectif est d'écrire deux versions du backtracking : la V0 dans laquelle on énumère toutes les solutions (même les non valides, c'est équivalent au "générer tout et tester"), et la V1 dans laquelle on ne se limite qu'aux solutions partielles valides.

#### Question 1.1.

Lisez les commentaires au début de la classe `PartialSolNQueen`.

- Pour représenter la solution partielle ci-dessous, quels seraient les valeurs des attributs d'une `PartialSolNQueen` ? 
- Cette solution partielle est-elle valide ? 

.	X	.	.
.	.	.	.
.	.	.	X
.	.	.	.

#### Question 1.2.

Ecrire la méthode `constraintOK` de la classe `PartialSolNQueen`. Pensez à lancer les tests, et à en écrire d'autres.

#### Question 1.3.

Ecrire la méthode `checkNewVal` de la classe `PartialSolNQueen`.

#### Question 1.4.

Ecrire la méthode `getUnaffectedVar` de la classe `PartialSolNQueen`.

#### Question 1.5.

Ecrire

- la méthode `backTrackQueenV0` de la classe `AlgosNQueen` (s'inspirer du cours).
- la méthode `mainBackTrackQueenV0` de la classe `AlgosNQueen`.

Testez avec `testV0`.

### Question 1.6.

Ecrire

- la méthode `backTrackQueenV1` de la classe `AlgosNQueen` (s'inspirer du cours).
- la méthode `mainBackTrackQueenV1` de la classe `AlgosNQueen`.

Testez avec `testV1`, et lancez le `testComparaisonV0V1` pour comparer le nombre d'appels récursifs fait dans chacune des versions.

### Question 1.7.

Introduisons les notations suivantes :

- pour toute solution partielle  $s$ , soit  $nb(s)$  le nombre d'appels récursifs effectués par `backTrackQueenV0(s)`,
- pour tout entier  $n$ , soit  $nb(n)$  le nombre d'appels récursifs effectués `backTrackQueenV0` quand on lance `mainBackTrackQueenV0(n)`.
- pour tout  $h \geq 1$  et  $\Delta \geq 1$ , soit  $a(h, \Delta)$  le nombre de sommets dans un arbre de hauteur  $h$  où chaque sommet a  $\Delta$  fils (par exemple  $a(2, \Delta) = \Delta + 1$  (la racine et ses fils))

Nous allons de minorer  $nb(n)$  (c'est à dire prouver que  $nb(n) \geq ..$ ) pour se rendre compte de la quantité de calculs faits. Imaginons ce que va faire `backTrackQueenV0` : ses deux premières décisions vont être "placer une reine en  $(0, 0)$  (en haut à gauche), puis placer une reine en  $(1, 0)$  (juste en dessous de la première)". Soit  $s$  cette solution partielle. Observez que  $nb(n) \geq nb(s)$ , on va donc se contenter de minorer  $nb(s)$ .

- Que va retourner `backTrackQueenV0(s)` ?
- Donner (sans la justifier) une minoration du type  $nb(s) \geq a(., .)$
- Donner une minoration  $a(h, \Delta) \geq ..$  en fonction de  $h$  et  $\Delta$
- En déduire une minoration  $nb(n) \geq ..$  (en fonction de  $n$ )

### Exercice 2. BackTrackQueenV2 et BackTrackQueenV3

Dans cette partie l'objectif est d'écrire deux nouvelles versions du backtracking qui utilisent des domaines pour les variables restantes. La V2 se contentera de faire du forward checking, et la V3 fera en plus la sélection de sa prochaine variable avec du MRV (Minimum Remaining Value).

On rappelle que l'idée du forward checking est que, dès que l'on souhaite rajouter une nouvelle reine en case  $(l_0, c_0)$  alors :

- on est sûr que cette nouvelle reine ne viole aucune contrainte (car on suppose que  $(s, D)$  est FCC, ce qui signifie que les valeurs restantes dans les domaines ont été "vérifiées", et donc que si la valeur  $c_0$  est dans le domaine de la variable  $l_0$ , alors une nouvelle reine en  $(l_0, c_0)$  ne pose pas de problème avec les reines déjà placées
- par contre, il faut faire appel à `s.propagateConstraints(D, l0, c0)` pour enlever des valeurs des variables restants à décider

### Question 2.1.

Ecrire la méthode `propagateConstraints` de la classe `PartiaSolNQueen`. Lancez `testPropagateConstraints` et observez si les domaines sont corrects. Ajoutez au moins un autre test de domaine.

### Question 2.2.

Ecrire

- la méthode `backTrackQueenV2` de la classe `AlgosNQueen` (s'inspirer du cours). Pensez à utiliser la méthode `AlgosUtilitaires.deepCopyMap` pour sauvegarder le domaine avant de faire la propagation.
- la méthode `mainBackTrackQueenV2` de la classe `AlgosNQueen` (pensez à utiliser `prepareDomain`).

Testez avec `testV2`.

On va maintenant améliorer la V2 en choisissant la prochaine variable comme celle ayant un domaine le plus petit possible.

### Question 2.3.

Ecrire

- la méthode `getUnaffectedVariableMRV` de la classe `AlgosUtiles`.
- la méthode `backTrackQueenV3` de la classe `AlgosNQueen` (quasi copiée collée de V2!)
- la méthode `mainBackTrackQueenV3` de la classe `AlgosNQueen`

Comparez les V1, V2, et V3 avec `testsLongs`.

## 3 Problème du sudoku

### Exercice 3. BackTrackSudoku.

Dans cette partie l'objectif est d'écrire un algorithme de backtracking pour le problème du sudoku, en faisant du forward checking, et en utilisant MRV pour le choix de la prochaine variable.

#### Question 3.1.

Lisez les commentaires au début de la classe `PartialSolSudoku` pour comprendre comment est stockée une solution partielle, ainsi que les méthodes `coordToInt`, `intToCoord` qui vous seront bien utiles. Ecrivez les méthodes `add`, `remove`, `isFullSolution`.

#### Question 3.2.

Ecrire les méthodes `reviseC` (s'inspirer de `reviseL`) et `propagateConstraints` (utilisez les méthodes `revise*`) de la classe `PartialSolSudoku`. Testez avec `testReviseC` et `testPropagate1` (vérifiez bien l'affichage produit par `testPropagate1`).

#### Question 3.3.

Ecrire la méthode `backTrackSudoku` de la classe `AlgosSudoku`. Testez avec `test4x4feasible1()`, `test9x9feasible1()`, et `test9x9unfeasible1()`. Pour ces tests, vérifiez que les fichiers `sudoku4x4feasible.txt`, `sudoku9x9feasible.txt`, `sudoku9x9unfeasible.txt` sont présents au bon endroit : normalement vous devez avoir un dossier `src/Sudoku`, et dans ce dossier vos fichiers `*Sudoku.java`, et ces fichiers `*.txt`. Vous pouvez également jouer avec le test `test2525` : ce test lance votre algorithme de backtracking (qui utilise MRV), et le sudoku 25x25 devrait être résolu en un temps raisonnable. Modifiez ensuite (temporairement) votre algorithme de backtracking en choisissant arbitrairement la prochaine case (au lieu de prendre une de domaine minimum). Vous pouvez par exemple faire `int numcase = D.keySet().iterator().next();`. Relancez et constatez que même les tests 9x9 sont lents (et le 25x25 est catastrophique!).

### Exercice 4. BackTrackSudokuCount.

Vous savez peut être qu'une grille de sudoku "officielle" doit avoir exactement une solution. Il a longtemps été ouvert de savoir si l'on pouvait avoir une grille de sudoku officielle avec strictement moins de 17 chiffres déjà placés (ou autrement dit de savoir si il existait une grille avec  $x \leq 16$  chiffres ayant exactement une solution). Ce problème a été résolu en 2012 (voir <https://arxiv.org/abs/1201.0749>), et la réponse était que effectivement, il faut bien au moins 17 chiffres pour créer une grille de sudoku officielle. On s'intéresse donc ici à la question de compter le nombre de solutions d'une grille donnée.

#### Question 4.1.

Ecrire la méthode `backTrackSudokuCount` de la classe `AlgoSudoku`. Votre code devrait être très proche de `backTrackSudoku`. Testez avec `test9x9counting`. Vous pouvez vous amuser à essayer des grilles avec moins de 17 chiffres, et lancer votre `backTrackSudokuCount`. Si vous trouvez 1, il y a un problème!.