Projet Recherche Opérationnel: Génération de sudokus par recuit simulé

Anaïs Pham
Anthony Foulfoin
Groupe F

4 mai 2010

Résumé

On étudie le recuit simulé à partir d'un algorithme de génération de grilles de sudoku.

Table des matières

1 Présentation générale **2**
1.1 Introduction 2

2 Spécification **2**

3 Conception **2**
3.1 Génération d'une grille de sudoku pleine 2
3.1.1 Principe de l'algorithme 2
3.1.2 Spécifications 3
3.2 Génération du problème 3
3.2.1 Principe de l'algorithme 3
3.3 Le critère du recuit 4
3.4 La probabilité fournie par le recuit 4
3.5 La température 4
3.6 Test d'arrêt 4

4 Codage **4**

5 Tests et comparaison avec la méthode sans recuit **5**

6 Conclusion **5**

1 Présentation générale

1.1 Introduction

Un sudoku est un puzzle mathématique : donnée une matrice 9*9 partiellement valuée, il faut la remplir de telle sorte que tous les chiffres de 1 à 9 apparaissent une et une seule fois dans chaque ligne, dans chaque colonne et dans chacun des 9 "sous carrés" obtenus en coupant la matrice en 3 dans le sens de la hauteur et de la largeur.
Notre objectif est de définir un algorithme de générationde sudokus. A ce prétexte ludique se combine un thème plus technique : l'étude du recuit simulé.

2 Spécification

Le sujet n'imposait pas de langage de programmation pour la génération du sudoku. Nous avons choisi le langage objet Java. En effet ce langage nous a paru le plus facile dans la réalisation de l'application. Des algorithmes de résolution de sudoku existaient déjà en Java, et du point de vue intégration à notre application ce fut beaucoup plus simple.

3 Conception

Ce projet se décompose en deux parties. La première partie consiste à générer une grille de sudoku pleine à partir de la définition d'un critère à choisir. La seconde partie consiste à générer le problème du sudoku et à retirer un maximum de valeurs à la grille pleine, tout en gardant l'unicité de la solution. Le fichier qui réunit ces deux étapes est *SudokuGen.java* qui contient une fonction main exécutant les deux parties du problème.

3.1 Génération d'une grille de sudoku pleine

Pour générer une grille de sudoku pleine, l'idée générale a été de partir d'une grille de sudoku fausse et d'effectuer des permutations sur les cases jusqu'à obtenir une grille de sudoku correct.

Pour organiser les modules, on dispose d'une classe *Grid.java* qui permet de modeliser une grille d'une de sudoku. La grille est initialisée dans ce fichier comme ceci :

1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9

On a ensuite une classe *GridPermut.java* qui gère le problème de la génération de la grille.

3.1.1 Principe de l'algorithme

L'algorithme repose sur l'utilisation d'un critère et la génération aléatoire de permutations dans la grille du sudoku. Le critère choisi est un entier qui représente le nombre d'incoherences presentes dans le sudoku. Cet attribut de la classe *GridPermut* sera mis a jour lors des permutations. Si ce critère devient nul ceci signifie que la grille est correcte et que l'algorithme est terminé.

Le critère est initialement non nul du fait que la grille entrée par défaut est incorrecte (grille ci dessus). On initialise le critère aux nombres d'incohérences de la grille précédente. On va donc permuter les cases du sudoku pour diminuer le critère, jusqu'a le rendre nul.

On commence par choisir aléatoirement 2 cases à permuter entre elles On rappelle qu'une case est définie par une ligne et une colonne. On applique la permutation à la grille de sudoku puis on calcule le nouveau nombre d'incohérences de la grille Si le nombre d'incohérences a diminué et est donc inférieur au critère alors on conserve la permutation et on met a jour la valeur du critère. sinon on repermute les deux cases selectionnées précédemment. On applique cet algorithme tant que le critere n'est pas nul.

3.1.2 Spécifications

Le critère du recuit simulé n'a pas été utilisé pour générer une grille pleine. En effet, la grille pleine générée s'obtient très rapidement. Nous avons donc à ce stade une grille pleine générée, il nous reste désormais à générer le problème en enlevant des valeurs à la grille.

3.2 Génération du problème

3.2.1 Principe de l'algorithme

L'algorithme se base sur l'utilisation de deux listes (permettant d'accélérer l'algorithme), et d'un solveur de sudoku externe.

Etape 1 :

La première liste appelée « *modifs* » contient la liste de toutes les modifications qui peuvent être appliquées à la grille pour une itération donnée. Une modification est représentée par les indices d'une case et la modification à lui appliquer (masquer = supprimer, ou demasquer = ajouter la case)

Initialement la liste modifs contient 81 modifications, une par case, chacune de ces modifications représentant le masquage de la case concernée.
On tire une modification au sort dans cette liste. On applique la modification sur la grille et on vérifie le critère et la probabilité fournie par le recuit s'il s'agit d'un demasquage.
On vérifie également si la nouvelle grille obtenue suite à cette modification possède une solution unique dans le cas où l'on retire une case.

Etape 2 :

Si une des conditions n'est pas vérifiée, et que la modification est un masquage, alors on retire la modification de la liste modifs et on la transfert dans la seconde liste appelée testedModifs, qui contient les modifications qui ne sont pas applicables à la grille actuelle.

Cela nous permet de ne pas retirer au sort une modification qui serait inutile pour la grille en cours de calcul.

Si au contraire toutes les conditions sont vérifiées, alors la modification est appliquée définitivement.

Etape 3 :
Si une case a été démasquée (ajoutée), toutes les modifications inutiles pour la grille précédente, stockées dans *testedModifs*, sont transférées dans motifs et peuvent ainsi de nouveau être appliquées pour la nouvelle grille obtenue.

Si une case a été masquée (supprimée), les modifications inutiles pour la grille précédentes sont conservées pour la grille actuelle, car si les modifications inutiles de la grille précédente engendraient plusieurs solutions, alors cela sera toujours le cas pour la grille précédente à laquelle on aura supprimé une valeur, donnant ainsi la grille actuelle.
On enlève ensuite la modification de la liste modifs, et on y ajoute la modification inverse (masquage au lieu de démasquage et vice-versa). L'usage de ces deux listes permet d'optimiser l'algorithme en évitant les calculs inutiles.

3.3 Le critère du recuit

Le critère du recuit est représenté par le nombre de cases non encore supprimées dans le sudoku. En faisant diminuer le critère, on diminue ainsi le nombre de cases du sudoku.

3.4 La probabilité fournie par le recuit

Le recuit nous donne la probabilité d'accepter qu'une case soit démasquée selon la formule suivante : $exp^{-1/T}$, où T représente la température à l'itération k du système.

3.5 La température

A une itération k donnée du système, la température utilisée par le recuit est fournie par la formule suivante : $T_{k+1} = \frac{T_k}{1+\frac{log(\delta)}{\epsilon} * T_k}$
Avec :
- δ : un réel positif proche de 1, fixé à l'étape 1 par expérimentation.
- ϵ : la valeur initiale du système, fixée à 100 par expérimentation.

3.6 Test d'arrêt

Deux conditions peuvent mettre fin à l'algorithme :
- On a obtenu le nombre de cases que l'on souhaitait (fixé à 22 par défaut)
- La température est inférieure à une valeur définie expérimentalement. Cette valeur est de 0.03, en dessous de celle-ci la probabilité d'appliquer un démasquage devient trop faible pour impacter le résultat final.

4 Codage

Tout notre code a été commenté et la javadoc a été générée. Voici cependant comment les classes sont organisées.

Les classes sont organisés dans 2 packages principaux :
- default package : il contient le programme principal qui va lancer l'algorithme.
- sudoku : ce package contient le code de l'application. Ce package est lui même décomposé en 2 sous-packages :
  - sudoku.generator : il contient l'ensemble des classes permettant la génération d'une grille de sudoku : *Grid.java, GridPermut.java GridRemoveValues.java, Modification.java SudokuGen.java, Util.java*
  - sudoku.solveur [1] il contient les classes permettant la résolution d'un sudoku. N'ayant pas à traiter cette partie, nous avons récupéré sur internet un programme de résolution de sudoku.

[1] http ://www.exampledepot.com/egs/Programs/sudoku_solver_SudokuSolver.html

5 Tests et comparaison avec la méthode sans recuit

Après avoir effectué des tests empiriques, il apparaît que le recuit permet d'obtenir presque systématiquement des grilles de 22 cases, et dans le pire des cas, de 23 cases, en quelques secondes seulement (de 2 à 12 secondes en moyenne selon les cas).

Moyennant un temps de calcul supplémentaire, et quelques modifications au niveau des paramètres de la température (afin de « ralentir sa chute »), il est également possible d'obtenir des sudokus de 20 ou 21 cases, mais cela nécessite au moins une dizaine de minutes avant d'obtenir un tel résultat. Nous avons donc privilégié la rapidité en paramétrant l'algorithme pour fournir rapidement des grilles de 22 ou 23 cases.

La méthode sans recuit fonctionne comme suit : une liste de modifications identique à l'algorithme avec recuit est créée. On tire au hasard une modification, on l'applique et on vérifie grâce au solveur si la grille possède une solution. Si elle n'en possède pas, on annule la modification, et dans tous les cas on retire la modification de la liste. L'algorithme s'arrête lorsque la liste des modifications est vide.

La méthode sans recuit nous permet d'obtenir des grilles allant de 23 à 26 cases presque systématiquement dans de très rares cas 22 cases, le tout en un laps de temps extrêmement court.

6 Conclusion

Ce projet nous a permis d'étudier le recuit et d'analyser son efficacité. Nous avons pu constater que la méthode avec recuit n'est pas très intéressante pour générer des grilles de plus de 23 cases. Elle parvient certes à générer presque assurément une grille avec la quantité de cases demandées, mais en des laps de temps beaucoup plus long. Il est tout aussi rapide de lancer plusieurs fois l'algorithme sans recuit.

Cependant l'algorithme sans recuit a beaucoup de difficultés à fournir des sudokus de 22 cases ou moins, dans ce cas, l'utilisation du recuit est particulièrement utile.