Génération de sudokus par recuit simulé

Pascal Ostermann
pascal.ostermann@enseeiht.fr

21 février 2010

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 génération de sudokus. À ce prétexte ludique se combine un thème plus technique : l'étude du **recuit simulé**.

Remarque importante

> L'objectif concret du projet n'est **pas** de générer des sudokus, mais d'**étudier le recuit simulé**. Votre note dépendra donc moins de votre programme que de la manière dont vous l'avez testé. Si votre opinion du recuit importe peu (par exemple : "c'est nul" ; "c'est génial" ; "ben oui, c'est pas mal" ; "il y a mieux"), vous devez convaincre le chargé de TP sinon que vous avez raison (il en a vraisemblablement sa propre opinion), du moins que vous y avez suffisamment travaillé pour émettre une opinion autorisée : si par exemple, vous prétendez qu'"il y a mieux", il faut le prouver, dire quel est ce "mieux" et le comparer au recuit sur un jeu de tests.

1 Générer en optimisant un critère

La méthode généralement employée est en deux parties : on génère d'abord une grille solution – un sudoku plein –, dans laquelle on enlève des valeurs tant que la solution y est unique. Chacune de ces étapes présente cependant des difficultés techniques.

1.1 Générer un sudoku plein

On peut générer un sudoku plein en plaçant 9 fois les 9 chiffres de 1 à 9 sur une grille, puis en permutant aléatoirement ces valeurs jusquà obtenir un

sudoku. Si l'on est certain d'obtenir un résultat – tout comme les singes tapant à la machine finiront par écrire l'Iliade – il est pourtant peu vraisemblable que ce soit en un temps raisonnable.

On guidera donc ces permutations par la définition d'un **critère** représentant la distance de la grille en cours à l'état "sudoku plein" : ce critère – de préférence entier – prendra la valeur 0 pour un sudoku et une valeur positive sinon. Et seules seront permises les permutations faisant décroître le critère.

Si le critère est mal choisi, vous devrez recourir à un algorithme de **recuit simulé**... (mes tests suggèrent pourtant qu'un critère raisonnablement choisi rend inutile le recuit ; d'autre part d'autres tests sur un autre problème suggèrent que le recuit est plus qu'inefficace dans le cas – comme ici – où l'on cherche une solution **exacte** ; dans l'idée d'étudier le recuit, peut-être – je n'ai pas testé – est-ce le lieu de montrer qu'il ne fonctionne pas toujours) mais j'y reviendrai par la suite.

1.2 Générer un problème

Comme dit plus haut, on enlève des valuations jusqu'à ne plus le pouvoir, jusqu'à ce qu'en enlevant une valeur de plus, le puzzle ait plusieurs solutions.

Ceci nécessite un algorithme **complet** de résolution de sudokus. Il en existe un grand nombre sur la toile[1], et il est aisé d'en définir un vous-même, par essai et erreur. Limitez cependant le nombre d'essais, en complétant par exemple cette méthode de force brute par les deux règles d'existence et d'unicité de mon sujet précédent *Résolution naturelle de sudokus* :

- (**Règle d'unicité**) À chaque case du sudoku, on peut aisément associer l'ensemble des valeurs possibles. Toute case du sudoku où n'est possible qu'une valeur peut alors être renseignée.
- (**Règle d'existence**) Chaque valeur doit apparaître une fois dans chaque carré, ligne ou colonne. Dès lors, si une telle valeur ne peut y être qu'à une seule position, elle peut être placée sur le sudoku.

Remarque

Notre objectif est de générer n'importe quel sudoku, donc un sudoku "diabolique". Les sudokus faciles, moyens ou difficiles peuvent s'obtenir de la même façon, en bridant l'algorithme de résolution.

2 Optimiser un critère non linéaire – le recuit simulé

La formulation générale de notre démarche est d'optimiser un critère par un ensemble de transformations. Jusque là, l'objectif pouvait être atteint en choisissant n'importe quelle transformation améliorant le critère (méthode du

[1] Vérifiez-en dans ce cas la *complétude* et ... N'OUBLIEZ PAS DE CITER VOS SOURCES.

simplexe, qui peut se résumer à "suivre la pente"). Pour générer un problème par exemple, enlever n'importe quelle valeur renseignée permet de progresser vers notre précédent objectif : obtenir un ensemble de valeurs renseignées mimimal **pour l'inclusion**.

Mais que faire si l'objectif est maintenant de minimiser le **nombre** de ces valeurs ? Pour citer un exemple provenant de mes tests (que je ne prétends pas exhaustifs), une même grille solution (que vous trouverez en annexe) peut admettre des problèmes minimaux pour l'inclusion comportant de 22 à 27 valeurs renseignées. Comment y obtenir des problèmes de 22 valeurs, ou au moins – pour me limiter à ce que je semble avoir réussi (je le répète, mes tests ne sont pas exhaustifs : leur seul objectif était de veiller à ce que les exigences de cet énoncé ne soit pas trop déraisonnables) – des problèmes de moins de 25 valeurs renseignées ? Que faire, quand pour passer d'une grille minimale (pour l'inclusion) de 27 valeurs à une grille réellement optimale il nous faut passer par une grille de 28 valeurs ?

En considérant notre critère comme une "énergie," il est clair qu'il faut "chauffer" le résultat provisoire, de manière à lui faire franchir le col qui le sépare d'un autre mimimum local (espéré plus proche de l'optimum) : voilà l'image derrière l'expression **recuit simulé**.

Plus techniquement, on adoptera le processus de Metropolis : on tire au sort une modification (permutation dans le cas des sudokus pleins, retrait **ou ajout** d'une valuation pour la grille de problème). Si le critère diminue, on applique la modification, s'il augmente d'une valeur δ on l'applique avec la probabilité $\exp(-\delta/T)$, où T est la "température". Vous aurez à fixer – sans doute expérimentalement – température initiale, loi de décroissance de la température et arrêt du recuit (il devrait s'arréter quand la température est suffisamment basse, mais parfois selon d'autres critères, si vous voulez que la solution sans recuit n'en soit qu'un cas particulier – au zéro absolu, bien sûr).[2]

3 Ce qu'il faut faire

Réaliser, dans le langage de votre choix[3], un programme permettant de générer des sudokus.

Votre programme sera testé **lors de la dernière séance de TP.**

Afin de permettre un test semi-automatique, la partie génération de problèmes devra faire l'objet d'un programme à part, prenant en entrée un fichier Unix représentant la solution : 9 lignes de 9 caractères entre '1' et '9', comme représenté par l'exemple en annexe.

La partie recuit simulé nécessitant une mise au point expérimentale, vous disposerez de plusieurs versions de celle-ci, une (au moins) avec recuit, une sans

[2] La description du processus de Metropolis est librement adapté de la wikipedia.

[3] Prière cependant de choisir le langage en fonction du problème et non par routine : l'argument lu l'année dernière selon lequel matlab était "plus familier" à l'étudiant parce qu'il en avait beaucoup fait dans l'année n'est pas de très bon augure pour un futur ingénieur informaticien : se rappelle-t-il encore matlab ?

recuit. Si votre recuit devait mettre trop de temps (ma meilleure version passe le quart d'heure...), vous la lancerez **dès le début de la séance de test**.

Votre rapport, **rendu au début de la dite séance de test**, devra mettre en valeur les points suivants :

- Quel critère avez-vous employé pour générer des sudokus pleins ? Avez-vous ici eu besoin du recuit ?
- Quelle méthode de résolution de sudokus avez-vous employé ? Pourquoi êtes-vous certain qu'elle est complète ?

Mais le point le plus important devrait en être un test sérieux du recuit :

- température(s) initiale(s), loi(s) de décroissance, test(s) d'arrêt testés ;
- comparaison avec la méthode sans recuit.

Poussez suffisamment vos tests pour vous convaincre de l'intérêt du recuit... ou le cas échéant pour me convaincre de son manque d'intérêt.

Le programme enfin, dûment indenté et commenté (entrées / sorties de tout sous-programme, structures de données[4], tests unitaires) devra mettre clairement en valeur ces divers points, quitte à le compléter d'un arbre de raffinage dans le rapport.

Annexe

Le sudoku plein que j'ai employé pour mes premiers tests (généré par un premier jet de mon programme) :

```
142783569
659412873
387965421
873124695
924657138
561839742
798341256
236578914
415296387
```

[4] J'en emploie trois différentes pour modéliser un sudoku, une pour générer une solution (une matrice d'entiers de 1 à 9), une autre pour mon programme de résolution (où une matrice de listes de "possibilités" est plus pratique), et une dernière (que je vous laisse inventer) pour générer un problème.