Projet d’algorithmique et complexitE

# resolution de sudokus – utilisation du backtracking

## Introduction

Pour notre projet nous avons décidé de nous pencher sur la résolution d’une grille de sudoku classique. Pour résoudre ce problème, on pourrait d’abord penser à un programme « intelligent », qui remplirait la grille de façon logique. Seulement si cela est assez simple pour les sudokus faciles, il en va autrement avec des grilles plus complexes. Cela nous amène donc à se demander s’il ne serait pas plus judicieux de passer directement par un brute force.

Cependant, un brute-force est-il vraiment envisageable en pratique ?

En revanche en combinant le principe du backtracking avec la méthode brute-force on arrive à réduire considérablement le nombre de combinaisons à explorer. Le backtracking, ou retour sur trace, est une méthode qui peut s’apparenter à un parcours en profondeur d’un arbre, avec une condition sur les nœuds : dès que la condition n’est plus remplie sur le nœud courant, on stop la descente sur ce nœud. En pratique dans notre problème, si nous remplissons la grille au fur et à mesure en vérifiant constamment qu’elle reste toujours potentiellement valide, on arrive rapidement à des situations de blocage. Dans un tel cas, on reviendra en arrière pour conserver une grille valide et on continuera l’exploration avec une autre possibilité. Concrètement, on test les différentes valeurs possibles pour une cases vide, si on trouve une valeur qui donne une grille valide, on passe à la case suivante, sinon on retourne à la case précédente (sans oublier de réinitialiser la case courante) puis on cherche une autre valeur possible pour cette case. En appliquant cette méthode de manière récursive on arrive donc à obtenir une grille complète rapidement car on évite tous les cas où il est inutile de continuer.

Par la suite nous verrons qu’il est possible d’optimiser cette solution pour la rendre encore plus efficace, simplement en ordonnant les cases vides dès le début, afin de déterminer lesquelles remplir en premier. Cela nous assurera que le backtracking sera toujours effectué de manière optimal et en outre d’éviter le pire des cas que nous décrirons plus loin.

## Presentation de l’algorithme – Pseudo code

*fonction* cases\_vides() ; //retourne une liste contenant l’ensemble des cases vides du sudoku

Input : *int* M[9][9] la grille de sudoku

Output : *list* cases\_vides

*fonction* existe\_sur\_ligne() ; //retourne Vrai si la valeur est dans la ligne relative à la case(i,j)

Input : *int* M[9][9] la grille de sudoku, *int* valeur, *int* i, *int* j

Output : *booléen* existe

*fonction* existe\_sur\_colonne() ; //retourne Vrai si la valeur est dans la colonne relative à la case(i,j)

Input : *int* M[9][9] la grille de sudoku, *int* valeur, *int* i, *int* j

Output : *booléen* existe

*fonction* existe\_sur\_bloc() ; //retourne Vrai si la valeur est dans le bloc relatif à la case(i,j)

Input : *int* M[9][9] la grille de sudoku, *int* valeur, *int* i, *int* j

Output : *booléen* existe

*fonction* get\_bloc() ; //retourne les coordonnées du bloc relatif à la case(i,j)

Input : *int* i, *int* j coordonnées de la case

Output : *int* x, *int* y coordonnées du bloc

*fonction* estValide()

Input : *int* M[9][9] la grille de sudoku, *int* num\_case, *list* cases\_vides

Output : *booléen* succès

*if* (num\_case >= *len*(cases\_vides)) *then*

*return True ;*

position = cases\_vides[num\_case]

i = position[0] ;

j = position[1] ;

*for* k in [1 : 9]  *do*

*if* ( *not*(existe\_sur\_bloc(M,k,i,j)) *and not*(existe\_sur\_ligne(M,k,i,j)) *and not*(existe\_sur\_colonne(M,k,i,j)) ) *then*

M[i][j] = k ;

*if* ( estValide(M, num\_case+1, cases\_vides) *then* //appel récursif

*return True*;

M[i][j] = 0 ;

*return False*;

## Preuve de la validite – invariant de boulce

## Complexite

… case vide n° 1

… case vide n° 2

…

… case vide n° n-1

Situation bloquante

… case vide n° n

## conclusion