Saé S1.02 : comparaison d'approches algorithmiques

Résolution de Sudoku : temps d'exécution et backtracking

Les programmes écrits lors de la première partie de la SAE ne permettaient pas une résolution certaine d'une grille quelconque de Sudoku. Le "taux" de résolution dépendait du niveau de difficulté de la grille et des techniques mises en œuvres.

Dans cette deuxième partie de la SAE, nous allons nous intéresser à un algorithme qui, lui, résout n'importe quelle grille de Sudoku. Mais les critères de performance utilisés précédemment (taux de remplissage de la grille et taux d'élimination des candidats) ne peuvent plus s'appliquer. Nous allons donc mettre en place un autre critère de performance de nos programmes : leur vitesse d'exécution.

1. Mesurer le temps d'exécution

1.1. Pemière méthode : les fonctions `time()` et `difftime()`[1]

La fonction `time()` permet d'obtenir le temps écoulé (en général le nombre de secondes) depuis le premier janvier 1970 à 00 h 00 mn 00 s, sous forme d'un entier positif : on parle de "**timestamp**". Le résultat est de type `time_t`.

Deux utilisations de cette fonction sont possibles : le temps écoulé peut être récupéré soit comme résultat de la fonction, soit comme paramètre de sortie.

Exemple 1 : utilisation comme une fonction, on ne se sert pas du paramètre.

```
time_t  horaire = time(NULL);
```

Exemple 2 : utilisation d'un paramètre de sortie.

```
time_t  horaire;
time(&horaire);
```

La fonction `difftime()` calcule la différence entre deux "timestamp", c'est-à-dire le nombre de secondes entre un timestamp de fin et un timestamp de début. Le résultat est de type `double`. Bien sûr le timestamp de début doit être antérieur au timestamp de fin.

```
//debut et fin sont deux "timestamp"

double duree = difftime(fin, debut);
```

Exercice 1

Récupérez les deux programmes de tri sur Moodle. Modifiez-les de manière à ce qu'ils affichent la durée d'exécution du tri (ne pas inclure la phase d'initialisation du tableau).

1 *Les fonctions et types abordés dans cette partie se trouvent dans <`time.h`>.*

1.2. Deuxième méthode : la fonction `clock()`

La méthode précédente possède plusieurs inconvénients. D'une part, le résultat obtenu est un nombre entier de secondes, ce qui est une précision insuffisante pour des programmes rapides. D'autre part, le résultat correspond au temps qui s'est écoulé entre le début et la fin du programme (ou de la partie du programme qu'on souhaite chronométrer). Mais un système multi-tâches alloue à tour de rôle un laps de temps CPU à chaque processus en cours. Le temps d'exécution mesuré précédemment ne donne donc pas le temps de calcul réel c'est-à-dire le temps réellement consacré par le processeur à votre programme.

La fonction `clock()` retourne le nombre de "ticks" (les "tops d'horloge") consommés par l'application en cours d'éxécution. Cela correspond à sa consommation CPU. Le résultat retourné est de type `t_clock`.

Pour connaître le temps CPU utilisé par un programme il faut donc calculer le nombre de ticks consommé et le diviser par le nombre de ticks par seconde (constante CLOCKS_PER_SEC), comme dans cet exemple, que vous pouvez tester :

```
clock_t begin = clock();
for (int i=0 ; i<100000 ; i++){
      for (int j=0 ; j<100000 ; j++){
      }
}
clock_t end = clock();
double  tmpsCPU = (end - begin)*1.0 / CLOCKS_PER_SEC;
printf( "Temps CPU = %.3f secondes\n",tmpsCPU);
```

Pour obtenir une différence encore plus flagrante, ajoutez l'instruction `sleep(3);` à ce petit programme. Cette instruction va allonger le temps d'exécution de 3 secondes, mais pas le temps CPU, puisque le programme va "dormir" pendant ces 3 secondes et ne consommer aucun temps CPU :

```
clock_t begin = clock();
for (int i=0 ; i<100000 ; i++){
      for (int j=0 ; j<100000 ; j++){
      }
}
sleep(3);
clock_t end = clock();
double  tmpsCPU = (end - begin)*1.0 / CLOCKS_PER_SEC;
printf( "Temps CPU = %.3f secondes\n",tmpsCPU);
```

Exercice 2

Reprenez les deux programmes de tri précédents et complétez-les de manière à ce qu'ils affichent le temps CPU de leur exécution (ne pas inclure la phase d'initialisation du tableau).

Exercice 3

Adaptez maintenant vos programmes de résolution de Sudoku "ELIMINATION" (versions 1 et 2, l'une utilisant la structure `tCase1` et l'autre utilisant la structure `tCase2`) : faites-en sorte qu'ils affichent le temps d'exécution du remplissage de la grille (ne pas inclure la phase d'initialisation de la grille).

Pour chaque grille fournie, comparez la vitesse de vos deux programmes.

2. Algorithme de résolution d'une grille de Sudoku

2.1 Principe de base : le "backtracking"

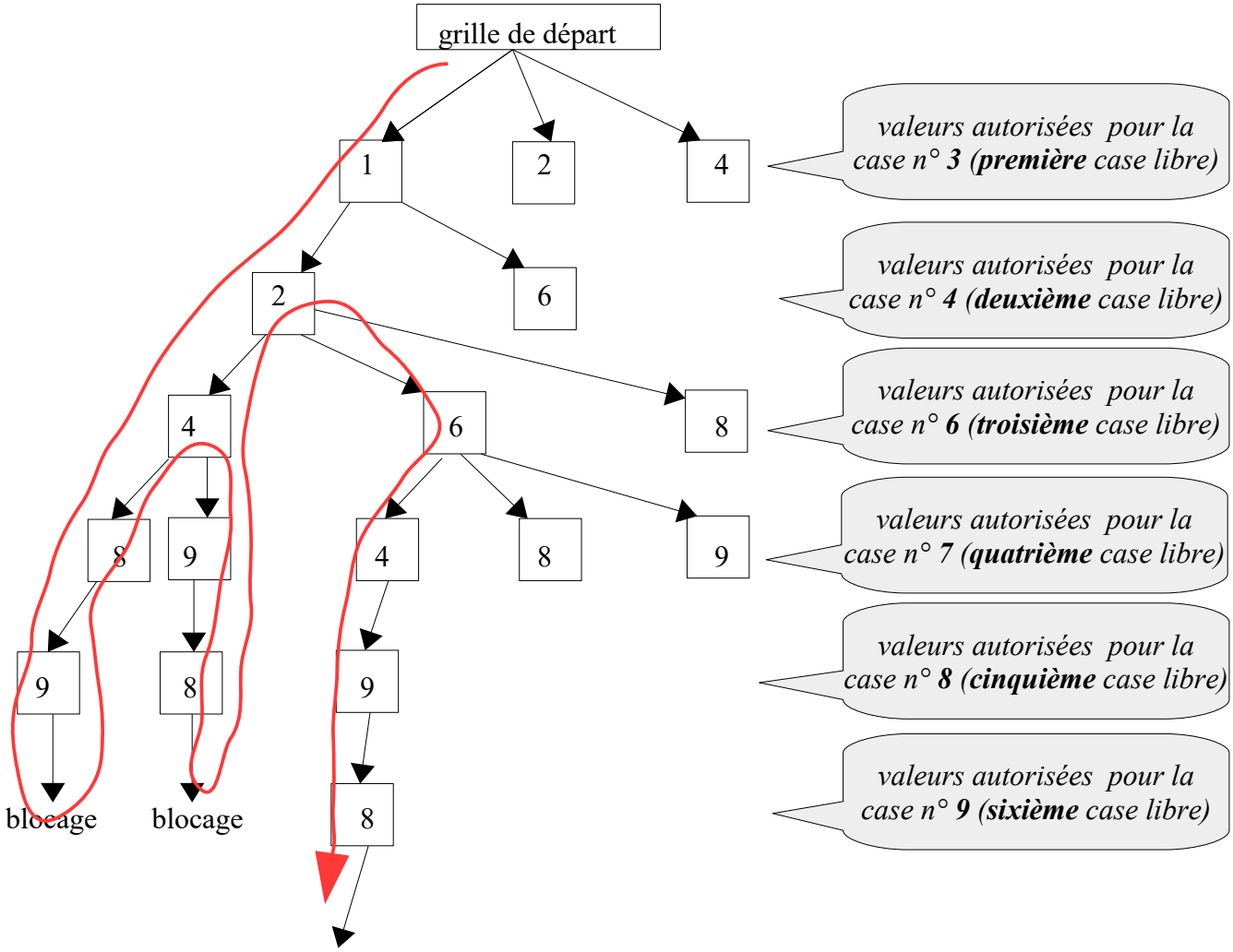
Le principe de l'algorithme est d'inscrire une valeur valide dans la première case libre, puis une autre valeur valide dans la deuxième case libre et ainsi de suite jusqu'à résoudre la grille ou bien jusqu'à un blocage quand plus aucune valeur ne peut être inscrite dans la case courante. En cas de blocage, on revient en arrière : on efface la dernière valeur inscrite et on en essaie une autre.

Exemple sur cette grille :

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

2.2 Parcours d'arbre

Cette technique de backtracking revient à parcourir un arbre des possibilités, en commençant toujours l'exploration par la branche la plus à gauche.



etc.

2.3 L'algorithme

On donne ici la fonction backtracking qui résout une grille de Sudoku. Elle doit être appelée avec la grille initiale et la case numéro 0 ;

```
constante entier N := 3;
constante entier TAILLE := (N*N);

type tGrille = tableau[TAILLE] [TAILLE] de entier;

fonction backtracking(tGrille grille, int numeroCase)
                                        délivre booléen c'est
  ligne : entier;
  colonne : entier;
  resultat : booléen;
début
  resultat := FAUX;
  si numeroCase = TAILLE * TAILLE alors
    //on a traité toutes les cases, la grille est résolue
    resultat := VRAI;
  sinon
    // On récupère les "coordonnées" de la case
    ligne := numeroCase / TAILLE;
    colonne := numeroCase % TAILLE;
    si grille[ligne][colonne]!=0 alors
      // Si la case n'est pas vide, on passe à la suivante
      // (appel récursif)
      resultat := backtracking(grille, numeroCase+1);
    sinon
      pour valeur de 1 à TAILLE faire
        si absentSurLigne(valeur,grille,ligne)
           ET absentSurColonne(valeur,grille,colonne)
           ET absentSurBloc(valeur,grille,ligne,colonne) alors
           // Si la valeur est autorisée on  l'inscrit
           //dans la case...
           grille[ligne][colonne] := valeur;
           // ... et on passe à la suivante : appel récursif
           // pour voir si ce choix est bon par la suite
           si backtracking(grille,numeroCase+1) = VRAI alors
             resultat := VRAI;
           sinon
             grille[ligne][colonne] := 0;
           finSi
         finSi
      finFaire
    finSi
  finSi
  retourne resultat;
fin
```