DM10

La technique du *retour sur trace*[1] construit la solution d'un problème incrémentalement, en s'interrompant dès que l'on est certain qu'une solution partielle ne pourra aboutir. Dès lors, on rebrousse chemin pour changer l'une des décisions prises précédemment. On peut s'arrêter dès qu'une solution est trouvée ou énumérer toutes les solutions. En termes de programmation, le retour sur trace est plus une méthode générale qu'un algorithme. Sa mise en œuvre diffère pour chaque problème, les constantes étant seulement les idées générales d'exploration systématique et d'interruption prématurée.

Sudoku

Le *problème du Sudoku* consiste à remplir une grille 9×9 en utilisant les chiffres de 1 à 9 en obéissant aux contraintes suivantes : chaque ligne, chaque colonne et chaque sous-groupe 3×3 doit contenir exactement une seule occurrence de chaque chiffre. Le problème est en général posé sous la forme d'une grille où certaines cases sont déjà remplies. La figure 1 contient un exemple de problème (à gauche) et sa solution (à droite). Les sous-groupes 3×3 y sont délimités par des traits gras. On se propose d'utiliser la technique du retour sur trace pour résoudre ce problème.

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

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

Figure 1 – Un problème de Sudoku et sa solution.

Comme dit plus haut, le principe du retour sur trace consiste à construire la solution *incrémentalement*, en s'interrompant dès qu'on peut déterminer qu'une solution partielle ne peut être complétée. On revient alors sur ses pas, en modifiant les choix faits précédemment. Dans le problème qui nous intéresse ici, cela revient à choisir une case vide (arbitrairement), tester successivement les valeur $1, \ldots, 9$ pour cette case et, si cette valeur est compatible avec les valeurs déjà placées dans la grille, recommencer.

On va écrire cet algorithme de retour sur trace sous la forme d'une fonction C `solve` qui reçoit en argument un tableau de 81 entiers[2] et qui renvoie un booléen.

```
bool solve(int grid[81]) { ...
```

Dans le tableau `grid`, on a des valeurs entre 0 et 9, la valeur 0 représentant une case encore vide. Avant d'écrire le code de cette fonction, nous allons donner sa spécification. En entrée, la fonction `solve` reçoit une grille qui ne contient pas de contradiction. Ce sera notamment le cas avec la grille initiale qui constitue le problème. En sortie, la fonction renvoie un booléen. Il vaut `true` si le tableau `grid` a pu être complété en une solution. Sinon, il vaut `false` pour indiquer que ce n'est pas possible et le tableau `grid` est inchangé. Ce dernier point est particulièrement important.

Si i (resp. j) est un numéro de ligne (resp. de colonne) compris entre 0 et 8, on choisit de faire correspondre la case (i, j) de la grille avec la case $9i + j$ du tableau. On se donne trois fonctions pour calculer respectivement le numéro de ligne, de colonne et de sous-groupe de la case représentée par l'indice `c`.

```
int   row(int c) { return c / 9; }
int   col(int c) { return c % 9; }
int group(int c) { return 3 * (row(c) / 3) + col(c) / 3; }
```

Il est en particulier très facile d'en déduire si deux cases `c1` et `c2` appartiennent à la même colonne, la même ligne ou le même sous-groupe.

```
bool same_zone(int c1, int c2) {
  return row(c1) == row(c2) || col(c1) == col(c2) || group(c1) == group(c2);
}
```

Pour interrompre prématurément notre recherche d'une solution, on écrit une fonction booléenne `check` qui vérifie si la case `p` contient une valeur en conflit avec une autre case. Pour cela, on parcourt toutes les cases.

1. *Backtracking* en anglais.
2. On pourrait utiliser un tableau bidimensionnel de taille 9×9 mais cela simplifie un peu notre code que de n'avoir qu'un seul tableau.

```
bool check(int grid[81], int p) {
  for (int c = 0; c < 81; c++)
```

et, pour chacune, on teste l'existence d'un conflit avec la case p. Le cas échéant, on le signale immédiatement.

```
    if (c != p && same_zone(p, c) && grid[p] == grid[c])
      return false;
```

Si en revanche on parvient à la fin de boucle, on signale l'absence de conflit.

```
  return true;
}
```

On peut désormais écrire le code de la fonction `solve` qui réalise l'algorithme de retour sur trace. Elle commence par une recherche de la première case vide de la grille, par un simple parcours de toutes les cases.

```
bool solve(int grid[81]) {
  for (int c = 0; c < 81; c++)
    if (grid[c] == 0) {
```

Pour cette case, on va essayer successivement toutes les valeurs v possibles.

```
      for (int v = 1; v <= 9; v++) {
        grid[c] = v;
```

La valeur v étant affectée à la case c, on teste l'absence de conflit avec la fonction `check`. Le cas échéant, on rappelle `solve` récursivement pour continuer la résolution du problème. Si `solve` trouve une solution, on termine immédiatement en signalant le succès de la recherche.

```
        if (check(grid, c) && solve(grid))
          return true;
      }
```

Noter que la fonction `solve` n'est pas appelée si `check` renvoie `false`, car l'opérateur `&&` est paresseux. Si en revanche on sort de la boucle `for`, c'est que les 9 valeurs possibles ont toutes été essayées sans succès. Dans ce cas, on restaure la valeur 0 dans la case c, puis on signale l'échec de la recherche.

```
    grid[c] = 0;
    return false;
  }
```

Il est important de comprendre que seule la première case vide a été considérée. En effet, si une solution existe alors la valeur correspondante de la case c aurait dû mener à cette solution. Il est donc inutile de considérer les autres cases vides. Ce serait une perte de temps considérable. Si en revanche on sort de la boucle `for`, c'est que la grille ne contient aucune case vide. Vu qu'on a supposé la grille sans contradiction en entrée, c'est donc une solution et on signale le succès.

```
  return true;
}
```

⌨ Programme 1 – code complet de résolution du Sudoku

```
int   row(int c) { return c / 9; }
int   col(int c) { return c % 9; }
int group(int c) { return 3 * (row(c) / 3) + col(c) / 3; }
bool same_zone(int c1, int c2) {
  return   row(c1) == row(c2)
       ||  col(c1) == col(c2)
       || group(c1) == group(c2);
}
// vérifie que la valeur à la position p
// est compatible avec les autres cases
bool check(int grid[81], int p) {
  for (int c = 0; c < 81; c++)
    if (c != p && same_zone(p, c) && grid[p] == grid[c])
      return false;
  return true;
}
// algorithme de retour sur trace (backtracking)
// entrée
// - grid ne contient pas de contradiction
// sortie
// - true si grid a pu être complétée en une solution
```

```
// - false si ce n'est pas possible *et* grid est inchangée
bool solve(int grid[81]) {
  for (int c = 0; c < 81; c++)
    if (grid[c] == 0) {
      for (int v = 1; v <= 9; v++) {
        grid[c] = v;
        if (check(grid, c) && solve(grid))
          return true;
      }
      grid[c] = 0;
      return false;
    }
  return true;
}
```

Un tel code résout le problème en un dixième de seconde. On pourrait penser que ce résultat doit beaucoup à la chance mais un test plus poussé sur 243 problèmes de Sudoku montre que ce n'est pas le cas. Il faut moins de 7 secondes pour résoudre tous ces problèmes, soit moins d'un tiers de seconde par problème. Toutefois, il n'est pas facile d'analyser ou de prédire ce qui se passe exactement dans un tel programme. Quelques observations empiriques peuvent apporter quelques éclaircissements. Une instrumentation facile du code montre qu'il fait exactement 142 256 appels à `solve`. C'est très peu au regard de l'espace potentiel de recherche. Pour une grille comportant 58 cases vides, avec 9 valeurs pour chacune, il est de taille $9^{58} \approx 2,22 \times 10^{55}$. Même en prenant en compte les contraintes initiales, qui limitent les valeurs possibles pour chaque case vide, on aurait encore $4 \times 4 \times 3 \times \cdots \times 3 \approx 1,42 \times 10^{32}$ ensembles de valeurs dans l'espace de recherche. Une recherche brutale, testant systématiquement chaque n-uplet de valeurs possibles pour les cases vides, n'a donc aucune chance de terminer dans un délai raisonnable. Ce qui fait la force de l'algorithme de retour sur trace, c'est *l'interruption prématurée* de la recherche dès lors que la valeur placée est incompatible avec les valeurs déjà présentes, sans chercher à donner des valeurs aux autres cases.

On le comprend mieux encore si on observe comment se répartissent tous les appels à `solve` *à chaque profondeur* dans la pile d'appels. Ici, la profondeur varie entre 0 pour le tout premier appel et 58 pour un appel sur une grille pleine. Là encore, une instrumentation simple du code donne cette information.

profondeur	0	1	2	3	...
nombre d'appels	1	4	11	28	...

En particulier, on constate que le programme a essayé successivement quatre valeurs pour la première case libre (la case $(0, 1)$ en l'occurrence) avant de trouver une valeur permettant une solution (la valeur 7 en l'occurrence). La figure 2 trace l'histogramme de ces profondeurs.

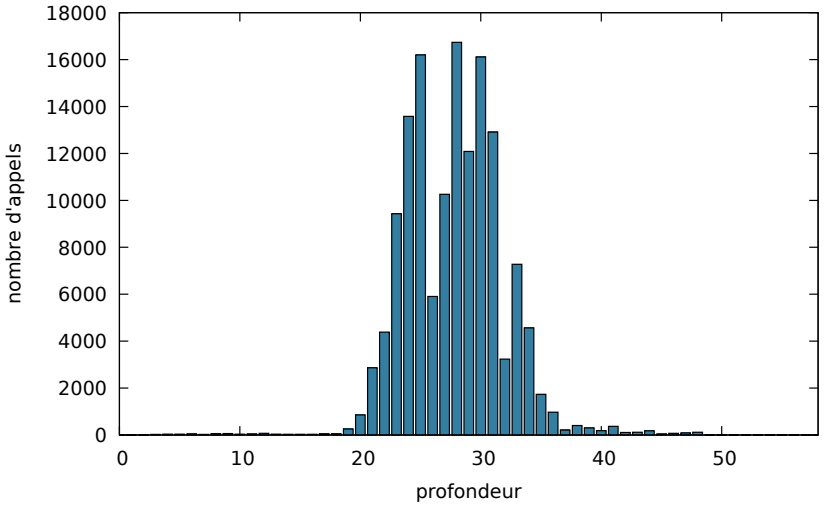

Figure 2 – Profondeur des appels à `solve`.

Il y a peu d'appels à des profondeurs faibles car la combinatoire est encore peu importante. Il y en a également peu à des profondeurs élevées car plus la grille est remplie et plus la fonction `check` renvoie la valeur `false`. En revanche, dans les profondeurs intermédiaires, entre 20 et 40, on a de très nombreux appels à `solve`. Cette distribution est caractéristique des algorithmes de retour sur trace.

Il est important de noter que le programme occupe *très peu de mémoire*, même s'il devait tourner très longtemps et explorer un très grand espace de recherche. Il n'y a en particulier aucune allocation sur le tas. Et par ailleurs, il n'y a aucun risque de faire déborder la pile d'appels car il n'y a jamais plus de 82 appels imbriqués.

Il est facile de l'adapter pour trouver *toutes* les solutions du problème. On trouve ici que notre grille n'a en fait qu'une seule solution, et cela prend à peine plus de temps (124 ms) et à plein plus d'appels à `solve` (148 983). Sur une grille moins contrainte, cependant, il pourrait y avoir de nombreuses solutions et le temps de calcul pourrait devenir beaucoup plus important.

Question 1. Écrire une variante du programme 1 qui ne s'arrête pas à la première solution trouvée, mais explore toutes les solutions et renvoie leur nombre. Sur la grille ci-dessous, on doit trouver 433 solutions.

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

Il convient de préciser que pour des problèmes de Sudoku plus grands (16×16, 25×25, etc.), il deviendrait nécessaire d'améliorer l'efficacité de notre programme dans sa recherche des cases vides et dans son exploration des valeurs possibles. Maintenir par exemple en permanence l'ensemble des valeurs possibles pour chaque case est une bonne approche. Pour autant, le principe du retour sur trace resterait exactement le même.

3-coloration de graphe

Question 2. En OCaml, écrire une fonction `color3 : graph -> int array` qui utilise la technique du retour sur trace pour 3-colorier un graphe non orienté. La fonction doit renvoyer un tableau affectant une couleur dans $\{0, 1, 2\}$ à chaque sommet, si une 3-coloration est possible, et lever l'exception `Not_found` dans le cas contraire.

Problème des reines

Question 3. En utilisant la technique du retour sur trace, écrire un programme qui résout le problème des N reines, à savoir placer N reines sur un échiquier $N \times N$ sans qu'elles soient en prise deux à deux.
Indication : en remarquant qu'il n'y a qu'une seule reine sur chaque ligne de l'échiquier, procéder ligne par ligne avec un tableau indiquant pour chaque ligne de l'échiquier dans quelle colonne se situe la reine de cette ligne.

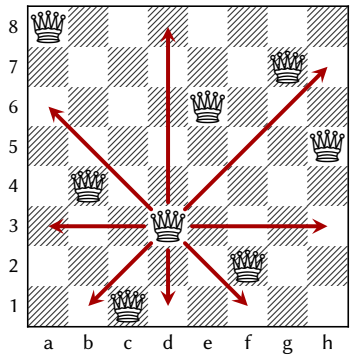

Figure 3 – Problème des reines avec $N = 8$.