Backtracking (retour sur trace)

Florian Bourse

1 Résolution de Sudoku par recherche exhaustive

Ce sujet propose implémentations de résolution de Sudoku en C, en utilisant une recherche exhaustive avec un retour sur trace (backtracking). L'idée de l'algorithme est le suivant :
Étant donné une grille partiellement remplie, nous allons essayer toutes les grilles de Sudoku contenant les nombres inscrits dans l'ordre lexicographique.
Pour chaque case vide, nous testons tous les chiffres dans l'ordre croissant, et si aucun chiffre ne permet d'obtenir une grille valide, nous retournons en arrière à la dernière position valide.

On représentera une grille par une matrice de dimension 9×9, voici un exemple :

Exemple de grille — C

```c
int grille[9][9] = {
  {9,0,0,1,0,0,0,0,5},
  {0,0,5,0,9,0,2,0,1},
  {8,0,0,0,4,0,0,0,0},
  {0,0,0,0,8,0,0,0,0},
  {0,0,0,7,0,0,0,0,0},
  {0,0,0,0,2,6,0,0,9},
  {2,0,0,3,0,0,0,0,6},
  {0,0,0,2,0,0,9,0,0},
  {0,0,1,9,0,4,5,7,0}
};
```

Pour mettre en place la résolution, nous aurons besoin de pouvoir tester si un chiffre peut être mis dans une case. On choisit pour cela de tester si ce chiffre appartient déjà à la ligne, la colonne, ou la région de cette case.

Une fois ces fonctions codées, nous mettons en place la résolution avec une fonction récursive.
Le cas de base est une grille complète, auquel cas la fonction renvoie `true`, ce qui signifie qu'une solution a été trouvée.
Si la grille n'est pas complète, on choisit une case, on la remplit avec un chiffre, et on teste via un appel récursif si la grille est résoluble. Si on ne peut mettre aucun nombre dans cette case, ou que toutes les possibilités ont été testées, on renvoie `false`, ce qui signifie que cette grille n'est pas résoluble.

Attention, il faut prendre garde à laisser la grille telle quelle, ou à la remettre dans l'état initial, si aucune solution n'est trouvée. Si une solution est trouvée, on laisse la grille complète.

2 Problème du sac à dos : 0-1 Knapsack

On s'intéresse à la résolution du problème d'optimisation du sac-à-dos 0-1 : à partir d'un ensemble d'objets définis par une valeur v_i et un poids w_i, on veut maximiser la valeur des objets emportés en ayant une contrainte sur le poids maximal W que l'on peut emporté. Plus formellement, le problème est défini par le problème d'optimisation linéaire en nombres entiers (integer linear programming) suivant :

0-1-knapsack :

Instance : un ensemble de couples $(w_i, v_i)_{i\in[\![0;k-1]\!]}$ de flottants, et un flottant W.
Solution : un ensemble de booléens $(x_i)_{i\in[\![0;k-1]\!]} \in \{0;1\}^k$ tels que

$$\sum_{i=0}^{k-1} x_i w_i \leq W.$$

Optimisation : maximiser

$$\sum_{i=0}^{k-1} x_i v_i$$

On l'implémente en OCaml par le type définit comme suit, avec un exemple de liste d'objets.

type des objets du sac à dos

```
type objet = { value : float; weight : float}

let exemple = [
    {value = 10.; weight = 0.5};
    {value = 10.; weight = 1.5};
    {value = 7.; weight = 0.3};
    {value = 10.; weight = 0.3};
    {value = 7.; weight = 0.3};
    {value = 3.; weight = 0.2};
  ]
```

On souhaite résoudre ce problème en testant toutes les possibilités. Si o nconsidère qu'il y a n objets, afin de ne pas tester les 2^n possibilités, nous allons introduire les objets 1 à 1, à condition que le poids ne dépasse pas le total autorisé.

Donner une relation de récurrence qui lie la solution du problème sur l'instance $(w_i, v_i)_{i\in[\![0;k-1]\!]}, W$ et des solutions du problème sur des instance contenant les objets $(w_i, v_i)_{i\in[\![0;k-2]\!]}$.

Nous sommes maintenant prêt à résoudre le problème en utilisant une fonction récursive. Tester sur la liste d'objets de l'exemple avec $W = 1$.