Université Paris Diderot - L2 informatique Année 2015–2016

TD d'Éléments d'Algorithmique n° 6

Backtracking

Exercice 1. *Sudoku.*

Une grille de sudoku est un tableau à deux dimensions de taille 9×9 – on parle aussi de matrice carrée 9×9 –, composé de 9 sous-grilles carrées 3×3. Une instance de sudoku est un remplissage partiel de la grille par des entiers entre 1 et 9.

Une solution d'une instance de sudoku est une grille complétement remplie par des entiers entre 1 et 9, telle que :

- chaque ligne, colonne et sous grille de la solution est sans répétition, et contient donc une et une seule fois chaque entier de 1 à 9.
- les cases initialement remplies de l'instance ne changent pas de valeur dans la solution.

On peut voir une instance de sudoku comme une *étape* dans la résolution d'une grille. Voici une instance de sudoku et une de ses solutions.

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

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

Il existe environ $6,67 \times 10^{21}$ solutions distinctes de l'instance vide (la grille sans cases remplies). Le but de cet exercice est de concevoir un algorithme qui, pour une instance donné, trouve une solution (ou échoue, si l'instance n'a pas de solutions). En d'autres termes, étant donné une grille partiellement remplie, on veut la compléter en une solution qui satisfasse les contraintes du jeu de Sudoku.

Backtracking, piles et récurrence :

La recherche d'une solution pour une instance de sudoku utilise le backtrack, et consiste en la construction d'un arbre modélisant les extensions possibles de la grille de Sudoku initialement donnée. On fixe tout d'abord une énumération des cases vides [1] , qui décrira l'ordre dans lequel on va chercher à remplir la grille initiale. L'arbre d'exploration des grilles est alors défini comme suit :

- La racine est étiquetée par l'instance de sudoku initialement donnée,

1. L'énumération choisie peut être $(9 * i) + j \mapsto (i + 1, j + 1)$, $0 \leq i, j \leq 8$, qui correspond à "avancer" ligne par ligne. Des énumérations plus complexes prenant en compte les contraintes de chaques cases donnent des meilleures performances.

- Les fils d'un nœud étiqueté G sont étiquetés par toutes les grilles $G_1, ..., G_k$ obtenues en remplissant correctement – c'est-à-dire en respectant les contraintes du Sudoku – la prochaine case vide de la grille G, selon l'énumération fixée initialement. Notons donc qu'il y en a au plus 9.

Rappelons que les feuilles d'un arbre sont les nœuds sans fils : d'après la définition, il y a donc deux cas :

- Le nœud n'a pas de fils, car il n'y a plus de cases à remplir : en ce cas, la feuille correspondante est une solution de l'instance initiale du Sudoku, on qualifie une telle feuille de "succès".
- Le nœud n'a pas de fils, car aucun remplissage de la case vide donnée par l'énumération ne satisfait les règles : en ce cas, la grille partielle qui l'étiquette est contradictoire et ne peut mener à une solution. On appelle "échec" une telle feuille.

Par conséquent, dans cet arbre, les étiquettes rencontrées sur un chemin menant de la racine à une feuille décrivent case par case le remplissage de la grille initiale aboutissant à l'instance qui étiquette la feuille. A chaque étape, il s'agit d'une hypothèse ; si la liste d'hypothèses était bonne, on atteint une feuille de type "succès", qui est une solution de la grille initiale, et on peut s'arrêter. Si la feuille est de type "échec", il faut remonter dans la chaîne des hypothèses, pour tenter une autre possibilité.

On construit l'arbre selon l'ordre préfixe. Pendant sa construction, on maintient une pile qui contient une entrée par noeud *ouvert*, c'est-à-dire une entrée par nœud menant de la racine au nœud que l'on construit actuellement. Selon l'explication donnée plus haut, cette pile contient donc une entrée par hypothèse ayant mené de la racine (la grille initiale) au nœud courant – la grille que l'on considère actuellement.

Chaque entrée contient la prochaine valeur à essayer pour remplir la case lui correspondant, ou 0 si toutes les valeurs entre 1 et 9 on été déjà essayées pour cette case. Même si cela n'est pas nécessaire (pouvez-vous dire pourquoi ?), on peut aussi stocker dans cette valeur les coordonnés de la case en question.

On empile une entrée quand on construit un noeud, et on la dépile quand ce noeud échoue (que ce soit une feuille échec ou un noeud dont tous les fils ont échoué). Quand on dépile, il faut *défaire* la dernière tentative de remplissage, c.à.d. il faut remettre à 0 la case gérée par le noeud dépilé.

Au moment de la construction d'une feuille "succès", on termine et on annonce la solution trouvée. Une variante de l'algorithme consiste à poursuivre la construction de l'arbre à la recherche d'éventuelles autres solutions.

Mise en place :

Deux choix sont possibles pour mettre en oeuvre cet algorithme :

- **itératif :** Le coeur du programme est une boucle qui s'exécute jusqu'à ce qu'une solution soit trouvée, ou que la recherche échoue, cas dans lequel la grille initiale n'a pas de solution. La gestion de la pile de backtrack est dans ce cas à la charge du programmeur.
- **récursif :** Le coeur du programme est une fonction récursive, dont chaque appel correspond à la création d'un noeud de l'arbre, qui prend en argument les coordonnés de la case qu'elle gère, qui essaye de remplir cette case et se rappelle sur la case suivante. La grille est dans ce cas une variable globale ou un paramètre. Cette fonction retournera un booléen. La pile de backtrack est dans ce cas implémentée par la pile des appels (récursifs) de fonction. Elle est gérée au niveau de la machine abstraite du langage, et non pas par

le programmeur.

1. Écrire une fonction qui prend en paramètre une grille, les coordonnées d'une case et un entier entre 1 et 9 et qui teste si l'ajout de cet entier aux coordonnées données mène à une grille satisfaisant les contraintes du Sudoku.
2. Donner une solution itérative du sudoku.
3. Donner une solution récursive du sudoku.

Exercice 2. *Les interrupteurs.*

Perdu dans un temple ancien à la recherche d'un trésor, vous avez déclenché par maladresse un piège mortel qui bloque la sortie. Votre seule issue est maintenant une porte dérobée à laquelle font face plusieurs étranges statues qui semblent liées à des leviers présents dans la salle.

Après quelques essais, vous parvenez aux conclusions suivantes :

- Chaque levier a deux positions, Nord et Sud.
- Chaque statue est reliée à plusieurs leviers.
- Pour ouvrir les yeux, une statue a besoin qu'au moins un des leviers auxquels elle est reliée soit dans une certaine position. Par exemple, que le levier 4 soit au Nord, ou que le levier 2 soit au Sud. Vous savez quels leviers actionnent quelles statues.
- La porte ne s'ouvrira que lorsque toutes les statues auront leurs yeux ouverts.

Par exemple, imaginons des statues commandées de la façon suivante :

1. 1 Nord ou 2 Nord ou 3 Sud
2. 2 Sud ou 3 Nord
3. 1 Nord
4. 1 Sud ou 3 Sud

Une configuration possible des leviers pour débloquer la porte est (Nord, Sud, Sud). **Y'en a-t-il d'autres ?**

Hélas, le temps presse, et activer un levier est une tâche épuisante. Fort heureusement, comme vous avez suivi assidûment le cours d'Enseignement à l'Algorithmique et que vous avez pensé à emporter avec vous votre ordinateur programmable pour mettre à jour votre statut Facebook pendant votre exploration, vous vous empressez d'écrire un programme qui vous donnera une possibilité de placement des leviers pour sortir de la salle et goûter à un repos bien mérité sur une plage où le miel et le coca-cola coulent à flots.

Voici comment vous comptez vous y prendre : initialement, aucun levier n'a de position affectée. Le déroulement informel de l'algorithme est le suivant :

1. Aller vers le prochain levier à activer
2. Le mettre dans la position Nord
3. Continuer avec le levier suivant

4. S'il n'y a plus de levier activable, mettre le dernier levier au Nord en position Sud et reprendre à partir de ce levier

Cet algorithme continue jusqu'à ce que la porte soit ouverte ou qu'il ne soit plus possible de remonter en arrière sur un levier dans une mauvaise position.

Question 1. Écrivez cet algorithme.

Question 2. Donnez des idées d'amélioration pour qu'il trouve une solution plus vite. Une piste possible est par exemple, pour une configuration partielle donnée, de regarder quels autres leviers ont alors une position déterminée. Ainsi, si on a positionné dans l'exemple précédent le levier 1 vers le Nord, que peut-on dire du levier 3 ?

Exercice 3. *Tentatives.*

Une carte est un couple d'entiers compris entre 1 et 6. La carte (k, l) est juxtaposable à la carte (i, j) si $j = k$. Une tentative est une séquence de carte. Elle est correcte si chaque carte de la séquence, mis à part la première, est juxtaposable à la carte qui la précéde dans la séquence.

Un tableau c de n cartes est donné ; c'est notre *jeu de carte*. Les cartes du jeu sont toutes distinctes, donc $n \leq 36$.

Une tentative utilisant uniquement des cartes du jeu est représentée par la séquence des indices dans c des cartes de la tentative. Une tentative est complète si elle utilise toutes les cartes du jeu (une et une seule fois).

Par exemple, si $n = 3$, $c[1] = (1, 2), c[2] = (2, 3), c[3] = (3, 4)$, alors la seule tentative correcte et complète est $1; 2; 3$, les autres tentatives correctes étant :

- ϵ (la tentative vide),
- 1
- 2
- 3
- $1; 2$
- $2; 3$.

Étant donné un jeu de cartes quelconque c, écrire un algorithme qui énumère toutes les tentatives correctes composées par de cartes du jeu c – chaque carte étant utilisable au plus une fois dans une tentative.

On pourra s'inspirer de l'approche par *backtracking* mise en œuvre dans la résolution du Sudoku.