

## TD n°6

### Backtracking

## 1 Les élections présidentielles à l'américaine

Les élections présidentielles américaines se déroulent en gros de la façon suivante : dans chacun des 50 États, les électeurs votent pour un des deux candidats, démocrate ou républicain. Si c'est le candidat républicain qui dépasse l'autre en nombre de voix dans cet État, l'État enverra à Washington des « grands électeurs », tous républicains. Si c'est le candidat démocrate qui gagne dans cet État, l'État enverra à Washington le même nombre de grands électeurs, tous démocrates. Le nombre des grands électeurs dépend de la population de l'État. Pour finir, les grands électeurs se retrouvent à Washington et votent conformément à leur étiquette. Le président élu est celui qui a le plus de voix de grands électeurs.

En pratique, sur un total de 538 grands électeurs, la Californie en a 55, le Texas en compte 34, l'État de New York 31, la Floride 27, la Pennsylvanie et l'Illinois 21, etc. Les autres chiffres sont plus faibles.

**Exercice 1** Donner un algorithme pour énumérer toutes les configurations où les deux candidats se retrouvent avec le même nombre de voix de grands électeurs. Cet algorithme doit fonctionner pour les États-Unis et pour tous les pays où se pratique ce type d'élections. Évidemment, on supposera connaître le nombre d'États et le nombre de grands électeurs par État.

**Exercice 2** Montrer que le nombre de telles configurations est pair.

**Exercice 3** Donner une modification de l'algorithme qui n'en produit que la moitié.

**Exercice 4** Proposer une modification très simple du système électoral américain pour que l'élection soit toujours effective, c'est-à-dire que les deux candidats ne puissent pas avoir le même nombre de voix de grands électeurs.

## 2 Résolution d'un Sudoku [un grand classique by Paris Diderot]

### 2.1 Description du problème (pour ceux qui ne connaissent pas encore...)

Le but de cet exercice est de remplir une grille de sudoku, un carré de 9 cases de coté, subdivisé en 9 blocs carrés identiques, de sorte que chaque ligne, colonne et bloc contienne une fois et une seule chaque chiffre de 1 à 9. L'algorithme doit retourner toutes les possibilités si plusieurs existent et être écrit en pseudo-code.

### 2.2 Stratégie

Nous allons implémenter une stratégie simple (vous ne jouez pas comme cela) mais efficace. L'idée est simplement de placer tous les 1 possibles (et que des 1). Si on arrive à en placer 9, on continue avec les 2, et ainsi de suite.

## 2.3 Structures de données et fonctions utilitaires

On dispose de

- le tableau pré-rempli `int sudoku[9,9]` (une valeur 0 indique une case blanche)
- Un `main` qui crée et pré-remplit cette grille (de façon valide, on suppose). Certaines cases ont donc une valeur, d'autres restent blanches – et votre algorithme doit les remplir !
- `appartient_ligne(k, i)` qui teste si un élément  $k$  est présent ligne  $i$
- idem `appartient_col(k, i)` et `appartient_carré(k, i)` (les 9 carrés sont numérotés de haut en bas et de gauche à droite)
- `numcarre(i,j)` qui donne le numero du carré pour la ligne  $i$  col  $j$

On vous épargne d'avoir à les programmer, mais vous devez savoir le faire les yeux fermés !

## 2.4 Exercices

**Exercice 5** On modélise la position des 1 par un vecteur `int pos_un [9]` tel que `pos_un[i]` donne le numéro de colonne du 1 de la ligne  $i$ . Une valeur 0 signifie “non encore placé”. Faire une fonction `code` qui construit ce vecteur à partir d'une grille.

**Exercice 6** Ecrire une fonction récursive `place_un(i, pos_un, sudoku)` utilisant le principe du back-tracking qui place le 1 de la ligne  $i$  et des lignes suivantes en fonction des contraintes de colonne et de bloc. Décrivez le traitement des erreurs (impossible de placer tous les 1)

**Exercice 7** Meme question pour placer les 2, 3 et jusqu'à 9. Plus précisément, on a un tableau `pos[k,i]` indiquant la colonne de l'élément  $k$  de la ligne  $i$ . On écrit une fonction `place(k, i, pos, sudoku)`. Combien d'appels récursifs fait-elle

- pour des valeurs de  $k$  différentes ?
- pour des valeurs de  $i$  différentes ?

Comment écrire le cas terminal selon qu'on veut afficher

- une grille solution, ou
- toutes les solutions ?

Et le traitement des erreurs ?

**Exercice 8** Quel temps cela prend-il ? Et l'espace mémoire occupé ?

**Exercice 9** Et si on inversait l'ordre de recherche : on tente d'abord de remplir la première ligne, en y mettant le 1, puis le 2, etc jusqu'au 9. On continue en remplissant la deuxième ligne, et ainsi de suite. Est-ce que ça marche ?

- Si oui, modifiez le code en conséquence (et justifiez)
- Si non, pourquoi ?

## 3 Les anagrammes

On cherche à trouver toutes les anagrammes d'un mot donné, c'est à dire tous les mots qui s'écrivent avec exactement les mêmes lettres, mais dans un ordre différent. Par exemple, le mot DINAR est une anagramme du mot RADIN. Par extension, on peut dire qu'un mot est une anagramme de lui-même et donc que le mot RADIN possède deux anagrammes, DINAR et RADIN (dans l'ordre alphabétique).

### 3.1 Un cas simple

On supposera d'abord que toutes les lettres du mot de départ sont différentes, comme dans l'exemple précédent.

La liste des mots est donnée par un dictionnaire  $D$ . On suppose qu'il existe une procédure `chercher(mot, D)` qui répond `VRAI` si le mot `mot` est dans  $D$  et `FAUX` sinon. On ne s'intéresse pas à la manière dont fonctionne cette procédure.

La structure de données utilisée pour représenter un mot est un vecteur, dont la première composante est la première lettre du mot, la seconde composante la seconde lettre, et ainsi de suite.

On commence par fabriquer un ensemble  $L$ , composé de toutes les lettres du mot de départ, sous la forme d'une liste dans l'ordre alphabétique. Dans notre exemple :  $L = \{A, D, I, N, R\}$ .

On veut écrire un programme qui produise toutes les anagrammes du mot de départ une fois et une seule, y compris ce mot de départ.

**Exercice 10** Écrire le pseudo-code de la procédure récursive et de son appel dans le programme principal. On pourra utiliser un ensemble `DéjàPris` dans lequel se trouvent les lettres qui composent la solution partielle.

**Exercice 11** Dans quel ordre le programme écrit-il les anagrammes ? Pourquoi ?

**Exercice 12** Quelle est la complexité au pire de la procédure, en nombre d'appels à `chercher` ?

### 3.2 Le cas général

On enlève maintenant la contrainte que toutes les lettres du mot de départ sont différentes. On part de mots qui peuvent avoir au moins une lettre qui arrive deux fois, comme `RATER`. Dans ce cas,  $L$  n'est plus un ensemble, mais un sac de lettres. Dans cet exemple :  $L = \{A(1), E(1), R(2), T(1)\}$ .

**Exercice 13** Comment aménager l'algorithme précédent pour écrire :

- toutes les anagrammes avec les répétitions dues à la multiplicité des lettres ? Par exemple, à partir de `RATER`, la liste de ces anagrammes sera `ARRET`, `ARRET`, `RATER`, `RATER`, `TARER`, `TARER` (on suppose pour cet exemple que le dictionnaire ne tient pas compte des accents).
- chaque anagramme une fois et une seule ? Dans ce cas, à partir de `RATER`, la liste des anagrammes doit être `ARRET`, `RATER`, `TARER`.

### 3.3 Un autre algorithme

Voici un autre algorithme, complètement différent, qui n'est pas basé sur le backtracking.

Pour chaque mot du dictionnaire, on regarde si le sac de lettres qui le compose est égal au sac de lettres du mot dont on cherche les anagrammes.

**Exercice 14** Quelle est la complexité de cette méthode en nombre de comparaisons ? Conclusion ?