



Rapport TD/TP Sudoku

HAI916I

M1 Informatique - Génie Logiciel

Faculté des Sciences de Montpellier

ROMERO Gaétan

9 octobre 2022

Sommaire

1 Partie TD	2
2 Partie TP	6
2.1 Tests	6
2.2 Code	7

1 Partie TD

Sudoku - TD

Q1 $N = \langle X, D, C \rangle$

$$X = X[i][j] \quad i, j \in [1, \dots, n]$$

$$D = X[i][j] \in \{1, \dots, n\}$$

$C =$

$\forall i \text{ AllDifferent}(X[i][1], \dots, X[i][n]) \rightarrow \text{les lignes}$

$\forall j \text{ AllDifferent}(X[1][j], \dots, X[n][j]) \rightarrow \text{les colonnes}$

On découpe notre Sudoku en squares de $\sqrt{n} \times \sqrt{n}$ de façon à ce que chaque square ne contienne que des valeurs différentes (comme pour les lignes et les colonnes que l'on a définies au dessus).

Soit un square :

$\forall i, j \in [0, \dots, n] \text{ et } \forall a, b \in [0, \dots, \sqrt{n}]$

$a \times \sqrt{n} < i < (a+1) \times \sqrt{n} \text{ et } b \times \sqrt{n} < j < (b+1) \times \sqrt{n}$

et $\forall x, y$ appartenant à un square :

AllDifferent (Square[x][y])

Q2 La taille de l'espace de recherche est n^{n^2}

Q3) Soit l'instanciation partielle $I = \{ \langle X_{1,1}, 4 \rangle, \langle X_{1,3}, 2 \rangle, \langle X_{1,4}, 3 \rangle, \langle X_{2,2}, 2 \rangle, \langle X_{2,4}, 4 \rangle, \langle X_{3,2}, 3 \rangle, \langle X_{3,1}, 1 \rangle, \langle X_{4,4}, 2 \rangle \}$

On déroule l'algo Backtrack($\langle x, d, c \rangle, I$) := true

1^{ère} - itération: $a = X_{1,2}$

$v = 1$

$\text{Backtrack}(\langle x, d, c \rangle, I \cup \langle X_{1,2}, 1 \rangle) = \text{true}$

2^{ème} - itération: $a = X_{2,1}$

$v = 3$

$\text{Backtrack}(\langle x, d, c \rangle, I \cup \langle X_{2,1}, 3 \rangle) = \text{true}$

3^{ème} - itération: $a = X_{2,3}$

$v = 1$

$\text{Backtrack}(\langle x, d, c \rangle, I \cup \langle X_{2,3}, 1 \rangle) = \text{true}$

4^{ème} - itération: $a = X_{3,1}$

$v = 2$

$\text{Backtrack}(\langle x, d, c \rangle, I \cup \langle X_{3,1}, 2 \rangle) = \text{true}$

5^{ème} - itération: $a = X_{3,3}$

$v = 4$

$\text{Backtrack}(\langle x, d, c \rangle, I \cup \langle X_{3,3}, 4 \rangle) = \text{true}$

6^{ème} - itération: $a = X_{3,4}$

$v = 1$

$\text{Backtrack}(\langle x, d, c \rangle, I \cup \langle X_{3,4}, 1 \rangle) = \text{true}$

7ème - itération: $a = x_{4,2}$

$$v = 4$$

$\text{Backtrack}(\langle x, D, C \rangle, I \cup \langle x_{4,2}, 4 \rangle) = \text{true}$

8ème - itération: $a = x_{4,3}$

$$v = 3$$

$\text{Backtrack}(\langle x, D, C \rangle, I \cup \langle x_{4,3}, 3 \rangle) = \text{true}$

Sudoku final

obtenue:

4	1	2	3
3	2	1	4
2	3	4	1
1	4	3	2

Q4 Déroulement de l'algorithme AC3 :

On prend $x_{1,2}$ avec $D(x_{1,2}) = \{1, 2, 3, 4\}$. On a $D(x_j) = \{2, 3, 4\}$

avec la contrainte ~~$\forall v_i \in D(x_{1,2}) \text{ et } \forall v_j \in D(x_j),$~~

$v_i \neq v_j$. On trouve $v_i = 1$ donc $D(x_{1,2}) = \{1\}$. On teste en plus des contraintes dans l'autre sens donc avec $x_j = x_{1,2}$. Si elles ne sont pas dans Q.

En l'occurrence, il n'y a aucun changement autre part.

Pour $x_{2,1}$, $D(x_{2,1}) = \{1, 2, 3, 4\}$ et $D(x_j) = \{2, 4\}$.

$\rightarrow D(x_{2,1}) = \{1\}$ après l'algorithme $\text{revise}(x_{2,1}, C_{ij})$.

Ensuite $x_{2,3} = \{1, 3\}$, or $D(x_{2,1}) = \{1\}$ et après $\text{revise}(x_{2,3}, C_{ij}) \rightarrow D(x_{2,3}) = \{3\}$

Ensuite $x_{3,1} = \{1, 2\}$

Ensuite $x_{3,3} = \{1, 3, 4\}$, or $D(x_{3,4}) = \{1\}$ et $1 = 1$ donc après $\text{revise}(x_{3,3}, C_{ij})$

$D(x_{3,3}) = \{4\}$

Ensuite $x_{4,2} = \{4\}$

Ensuite $x_{4,3} = \{3,4\}$ or $D(x_{4,3}) = \{3\}$, donc $D(x_{4,3}) = \{4\}$.

Q5

Backtrack ($\langle x, D, C \rangle, I$):

If I is complete then return false

Select a variable x_i not in I

for Each v in $D(x_i)$ do

If $I \cup \langle x_i, v \rangle$ is locally consistent then

If Backtrack ($\langle x, D, C \rangle, I \cup \langle x_i, v \rangle$) then
return true

return false

2 Partie TP

2.1 Tests

Pour évaluer les tests, deux méthodes différentes ont été créées. Leur fonctionnement est quelque peu identique, dans le sens où elles évaluent la différence de temps d'exécution entre SudokuBT et SudokuPPC. Cependant, une le fait sur la recherche d'une seule solution, alors que l'autre le fait pour la recherche de toutes les solutions.

En premier, nous avons `testOneSolutionExecutionTime()` :

```
1  @Test
2  void testOneSolutionExecutionTime() {
3      int n = 4;
4
5      SudokuPPC sudokuPPC = new SudokuPPC();
6      SudokuBT sudokuBT = new SudokuBT(n);
7      long begginingTimeBT, endingTimeBT, executionTimeBT;
8      long begginingTimePPC, endingTimePPC, executionTimePPC;
9
10     //Save the current time before execution
11     begginingTimeBT = System.currentTimeMillis();
12
13     sudokuBT.findSolution(0, 0);
14
15     //Save the current time after execution
16     endingTimeBT = System.currentTimeMillis();
17
18     //Save the execution time
19     executionTimeBT = endingTimeBT - begginingTimeBT;
20
21     //Same with SudokuPPC
22     begginingTimePPC = System.currentTimeMillis();
23     sudokuPPC.solve(4);
24     endingTimePPC = System.currentTimeMillis();
25     executionTimePPC = endingTimePPC - begginingTimePPC;
26
27     //Print the executionTime
28     System.out.println("\n");
29     System.out.println("Pour une grille 4x4 : ");
30     System.out.println("Temps d'exécution pour une solution de SudokuBT : " +
31                         executionTimeBT + "ms");
32     System.out.println("Temps d'exécution pour une solution de SudokuPPC : " +
33                         executionTimePPC + "ms");
34 }
```

Listing 2.1 – `testOneSolutionExecutionTime()`

Ici, pour calculer le temps d'exécution, on utilise la méthode `currentTimeMillis()` qui récupère le temps en millisecondes entre l'instant présent et le 1er janvier 1970. Grâce à cela, on peut récupérer le temps actuel avant l'exécution, et le temps après celle-ci. De cette manière, on obtient le temps total de celle-ci en soustrayant le temps avant au temps d'après.

En second nous avons donc `testAllSolutionsExecutionTime()`. Cette méthode est construite de la même manière que la première, avec la seule différence qu'elle appelle `solveAll(n)` et `findSolutionAll(0,0)`, au lieu de `solve(n)` et `findSolution(0,0)`.

Voyons maintenant les résultats obtenus pour une grille 4x4 :

Pour une grille 4x4 :

Temps d'exécution pour une solution de SudokuBT : 5173ms

Temps d'exécution pour une solution de SudokuPPC : 2ms

Pour une grille 4x4 :

Temps d'exécution pour toutes les solutions de SudokuBT : 48519ms

Temps d'exécution pour toutes les solutions de SudokuPPC : 89ms

On voit clairement que celui réalisé avec la programmation par contraintes (donc SudokuPPC) est beaucoup plus rapide que celui qui implémente l'algorithme de Backtrack (donc SudokuBT). Et plus il y a de solutions, plus l'écart se fait ressentir.

2.2 Code

Lors du TP, plusieurs questions nécessitaient de pouvoir lire des fichiers contenant l'instance d'un sudoku particulier. Par exemple, l'instance du Sudoku 16x16, ou alors le Greater Than Sudoku. Pour faire cela, nous avons une méthode qui s'appelle `loadInstanceFromFile(String filePath)` et qui prend un chemin vers un fichier en paramètre, afin d'en extraire les informations.

Ces fichiers sont stockés dans le dossier ressources de notre projet, et leur contenu est fait de telle sorte que l'on rentre chaque élément de notre sudoku avec ses coordonnées, en sachant que pour une grille de 9x9, les coordonnées vont de 0 à 8 pour les lignes et les colonnes. Voyons un extrait du fichier `SudokuPPC9x9.txt` :

```
1      8;0;0
2      3;1;2
3      6;1;3
4      7;2;1
5      9;2;4
6
```

Listing 2.2 – SudokuPPC9x9.txt

Ici, on peut voir que l'élément 8 se situe à la première case de notre grille. De même, 3 est dans la case qui se situe à la deuxième ligne et à la troisième colonne.

Voici donc le code de la fonction `loadInstanceFromFile(String filePath)` :

```

1   public List<SudokuElement> loadInstanceFromFile(String filePath) throws
2       IOException {
3       BufferedReader fileReader =
4           Files.newBufferedReader(Paths.get(filePath));
5       String nextLine = null;
6       List<SudokuElement> listElement = new ArrayList<SudokuElement>();
7       String[] splitFileLine;
8       boolean isLetter;
9
10      while ((nextLine = fileReader.readLine()) != null) {
11          splitFileLine = nextLine.split(";");
12          isLetter = false;
13          for (int i=0; i < letterList.size(); i++) {
14              if (letterList.get(i).equals(splitFileLine[0])) {
15                  isLetter = true;
16                  listElement.add(new SudokuElement(i + 10,
17                      Integer.parseInt(splitFileLine[1]),
18                      Integer.parseInt(splitFileLine[2])));
19              }
20          }
21          if (!isLetter) {
22              switch(splitFileLine[0]) {
23
24                  case "<":
25                      splitFileLine[0] = "-1";
26                      break;
27
28                  case ">":
29                      splitFileLine[0] = "-2";
30                      break;
31
32                  case " ":
33                      splitFileLine[0] = "-3";
34                      break;
35
36                  case "v":
37                      splitFileLine[0] = "-4";
38                      break;
39
40              }
41          }
42      }
43  }
44

```

Listing 2.3 – loadInstanceFromFile(String filePath)

Cette fonction permet donc de charger l’instance d’un sudoku de n’importe quelle taille dans une liste de *SudokuElement* qui est une classe permettant de créer des objets ayant pour attributs un élément et des coordonnées dans une grille. Une fois cela fait, on envoie la liste nouvellement créée, à une méthode *addConstraintToModel(List<SudokuElement> listElement)*, qui va s’implémenter ajouter au modèle, les contraintes acquises par l’intermédiaire du fichier. Pour cela, on utilise la méthode *model.arithm()* de Choco Solver.

loadInstanceFromFile(String filePath) permet, comme on peut le voir sur l’extrait du code, de charger aussi bien un sudoku pré rempli, que le Greater Than Sudoku de la question 12. Pour détecter s’il s’agit de celui-ci, elle regarde à chaque nouvelle ligne qu’elle lit avec le *BufferedReader*, si le premier élément correspond à un des signes suivant $<$, $>$, \vee ou \wedge . Si oui, elle leur assigne une valeur pré définie et ensuite, la fonction *addConstraintToModel(List<SudokuElement> listElement)* comprend de quel signe il s’agit, et ajoute donc la bonne contrainte en fonction.

Pour ce qui est du fonctionnement lors de la grille 16x16, avec les lettres qui se rajoutent, la fonction ne fait que transformer chaque lettre en un nombre au-delà de 10. C'est à dire que A devient 10, B devient 11 et ainsi de suite. Et lors de l'affichage, on retrouve la lettre avec le même principe. C'est à dire que l'on va chercher la lettre correspondante dans une liste contenant déjà les lettres, grâce à la valeur de la lettre - 10 pour obtenir son indice dans la liste. Par exemple, A (qui vaut 10 dans notre système), se situe à l'indice 10-10 donc l'indice 0 dans la liste, B à l'indice 11-10, donc 1 et ainsi de suite.