UFR Mathématique Informatique Strasbourg

Python project :

Sudoku solver

Clément BUSCHE
Oxana USHAKOVA

Instructor:
Christophe STEINER

Import a problem

Before being solved, the sudoku problem must be imported as .txt file where all the numbers are put in a row and zeros define empty cells.

Here we have two options: either we import an existing problem or generate a new one.

1.1 Generation of sudoku

In order to generate a new sudoku file we follow this algorithm:

1. We create a grid A:
   - The first line is done by order from 1 to 9, so 123 456 789;
   - The second line begins with the first number of second 3 digits bloc of the first line, so 4,and continues;
   - The third line begins with the first number of second 3 digits bloc of the second line, so 7 ,and continues;
   - When starting the new area, its first line starts with the second number of the second 3 digits bloc of the last line and continues;
   - Further lines are defined as in (2) & (3);

   Output:

   123 **456** 789

   456 789 123

   789 1**23 4**56

   234 567 891

   567 891 234

   891 2**34 5**67

   345 678 912

   678 912 345

   912 245 678

2. Mixing it with allowed operations

This grid satisfies the sudoku rules, but we have to mix it. We can use following operations that won't break the given structure:

- Total transposing the grid
- Switching rows/columns in one area
- Switching areas (horizontal/vertical)

After those methods are defined, we create a loop, that choose one of them in a random manner and apply it to the grid A; this operation is done several times.

3. We give 0 value to some randomly taken cells

This method does not guarantee the unicity of the solution. In order to be sure that the generated grid has only one solution, we could call the solver and follow this algorithm:

- Chose a cell randomly, save it in temp and delete in grid B
- Solve sudoku:
  - If the solution is unique – continue
  - If multiple solutions – come back and break

This idea is implemented in generate.py file.

Solver structure

The input file *sudoku.txt* has two lines. The first line is 0 or 1, the second – sudoku grid put in row. If we put 0 at the first line, then the solver will call *generate.py* to get a new grid; if we put 1, then it will work with the grid given in the second line. Empty cells are declared as '0'.

The grid is loaded by the solver - *sudoku3.py,* which is the executive file for this project. When the grid is loaded, the solver translates it into a dictionary with *grille_vers_dictionnaire(grille)* and create a dictionary named *voisins*, where for every cell of the grid are stored values from the same raw, the same column and the same bloc. For example, for a[A][d] *voisins* stores values of [A], of [d] and of II bloc.

Then, with *voisins* dictionary we find all the possible inputs for every cell using the *propagation* algorithm. So in the end our grid is filled with some new 'true' values and becomes easier to solve or even solved.

But if the solution is still not obvious – we use *backtracking_final* which implements backtracking algorithm. This method stops not when the first solution is found, but it continues until it finds all of them. All the possible solutions are stored in *liste_dictionnaires.*

When the imported problem is solved, the solver generates *solutions.txt,* where the initial grid and all the its possible solutions are nicely presented in usual sudoku form. In the solver there are several methods of type *affiche*_XXX: they are also designed to give the usual sudoku form but in terminal (at different steps of solution).

Remarque: Here we gave most important methods and common ideas used in the solver, you will find accurate explanation for every method in *sudoku3.py,.*

Time test

3.1 Backtracking

Idea (given in wiki):

```
Initialize 2D array with 81 empty grids (nx = 9, ny = 9)
Fill in some empty grid with the known values
Make an original copy of the array
Start from top left grid (nx = 0, ny = 0), check if grid is empty
if (grid is empty) {
  assign the empty grid with values (i)
  if (no numbers exists in same rows & same columns same as (i) & 3x3 square (i) is currently in)
    fill in the number
  if (numbers exists in same rows & same columns same as (i) & 3x3 square (i) is currently in)
    discard (i) and repick other values (i++)
}
else {
  while (nx < 9) {
    Proceed to next row grid(nx++, ny)
    if (nx equals 9) {
      reset nx = 1
      proceed to next column grid(nx,ny++)
      if (ny equals 9) {
        print solution
      }
    }
  }
}
```

This method is extremely expensive both for memory and time.

3.2 Propagation

Idea (given in project):

Algorithme. *Algorithme par liste chaînée*

1. *construire pour chaque cellule la liste des possiblités*
2. *on cherche les liste de plus courte longueur (on veillera à minimiser le nombre de parcours de la grille)*
3. *si la plus courte liste est de taille* 1, *on choisit cette valeur. On met à jour les listes des possibilités pour les cases alentours.*
4. *si la plus courte liste est de taille* > 1, *on crée une deuxième grille (copie de la première). On choisit la première valeur de la liste des possiblités pour cette nouvelle grille créée.*
5. *Revenir à l'étape 2 pour chacune des grilles*
6. *L'algorithme s'arrête dès qu'une grille est remplie.*

This method allows us cut the space of possible solutions.

3.3 Time test

We take a grid that has a unique solution, and we solve it with and without propagation method. Here are the results:

- with propagation: 0.00364398956299
- without propagation: 0.00793504714966

As you can see, the propagation method is two times faster than “raw” backtracking.

Now we take a grid with multiple solutions:

- with propagation: 0.134737968445
- without propagation: 0.136278152466

Here the execution time is almost the same, it can be explained by the fact that for this sudoku we have 37(!!) possible solutions, so simple backtracking has enough time to find the solution while propagation cleans the space of possibilities which is quite big in this case.

Visualization

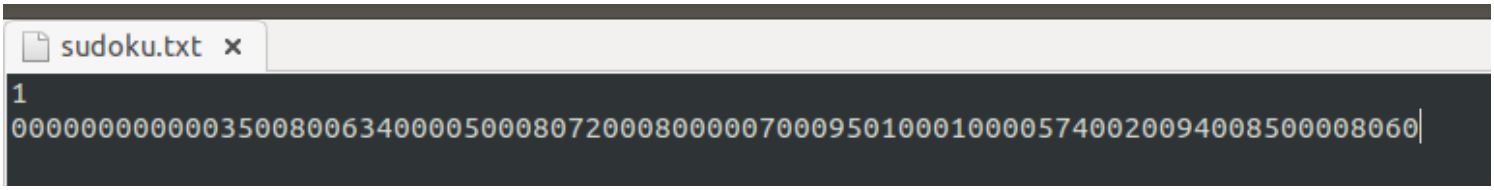
- Grid to solve

```
. . . | 4 . . | 8 7 .
. 4 7 | . 9 2 | . 5 .
2 . . | 6 . . | . 3 .
----------------------
9 7 . | 5 . . | 2 . 3
5 . 8 | . 2 4 | 7 . 6
6 . 4 | . . 7 | . 8 5
----------------------
. 9 . | 3 . 8 | . . 7
. . 3 | 2 4 . | 1 6 .
. 1 2 | . . . | . 9 .
```

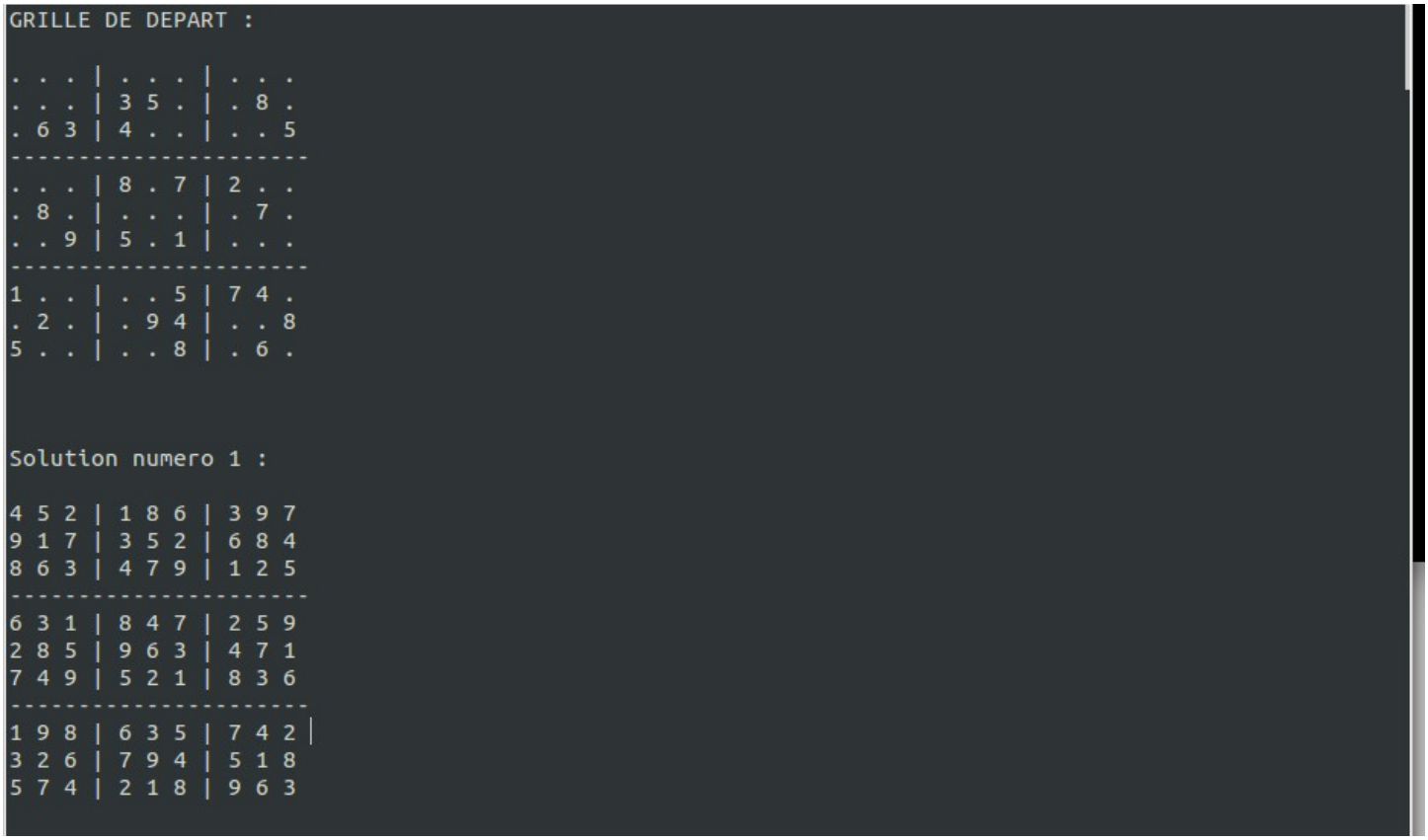
- Possibilities for every cell

```
123456789 123456789 123456789 | 4         123456789 123456789 | 8         7         123456789
123456789 4         7         | 123456789 9         2         | 123456789 5         123456789
2         123456789 123456789 | 6         123456789 123456789 | 123456789 3         123456789
----------------------
9         7         123456789 | 5         123456789 123456789 | 2         123456789 3
5         123456789 8         | 123456789 2         4         | 7         123456789 6
6         123456789 4         | 123456789 123456789 7         | 123456789 8         5
----------------------
123456789 9         123456789 | 3         123456789 8         | 123456789 123456789 7
123456789 123456789 3         | 2         4         123456789 | 1         6         123456789
123456789 1         2         | 123456789 123456789 123456789 | 123456789 9         123456789
```

- .txt file to import – sudoku.txt:



- .txt files exported- solutions.txt



Faced challenges

- Multiple ideas and approaches were used before final decision of using dictionary for grid (classes, tables, etc)
- Mastering dictionaries
- Mastering class structure
- Recursive structure of the code

Sources used:

-http://norvig.com/sudoku.html

-https://docs.python.org/