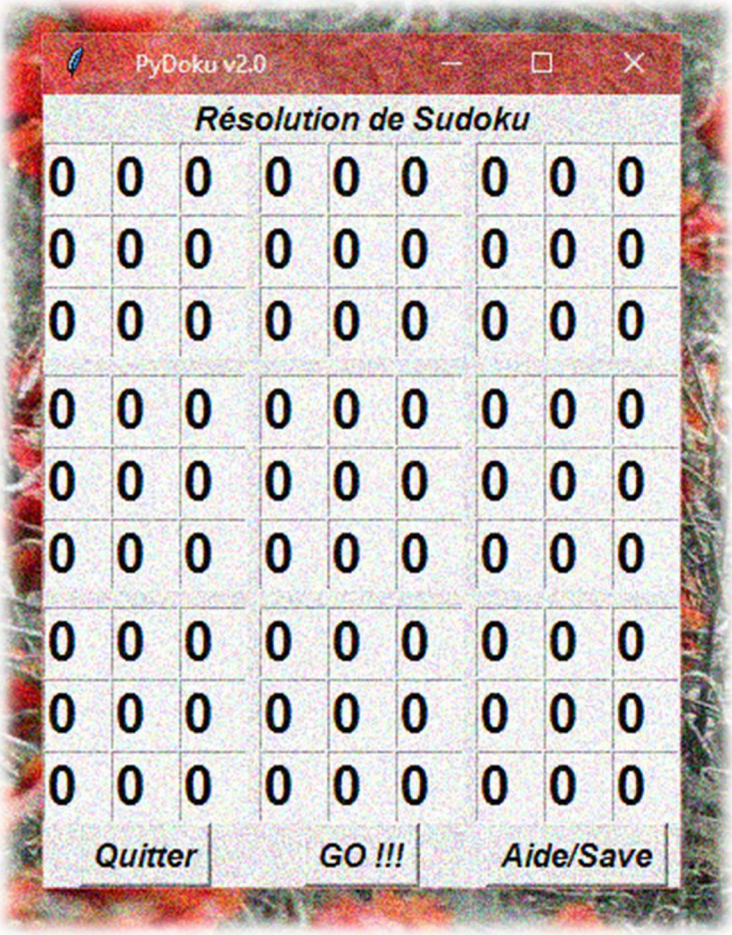
Alexandre l'Heritier

TS6

Projet final d'ISN

PyDoku



Avec Ricardo Ramos et Rita Dos Santos

Le but visé et les moyens choisis pour atteindre ce but

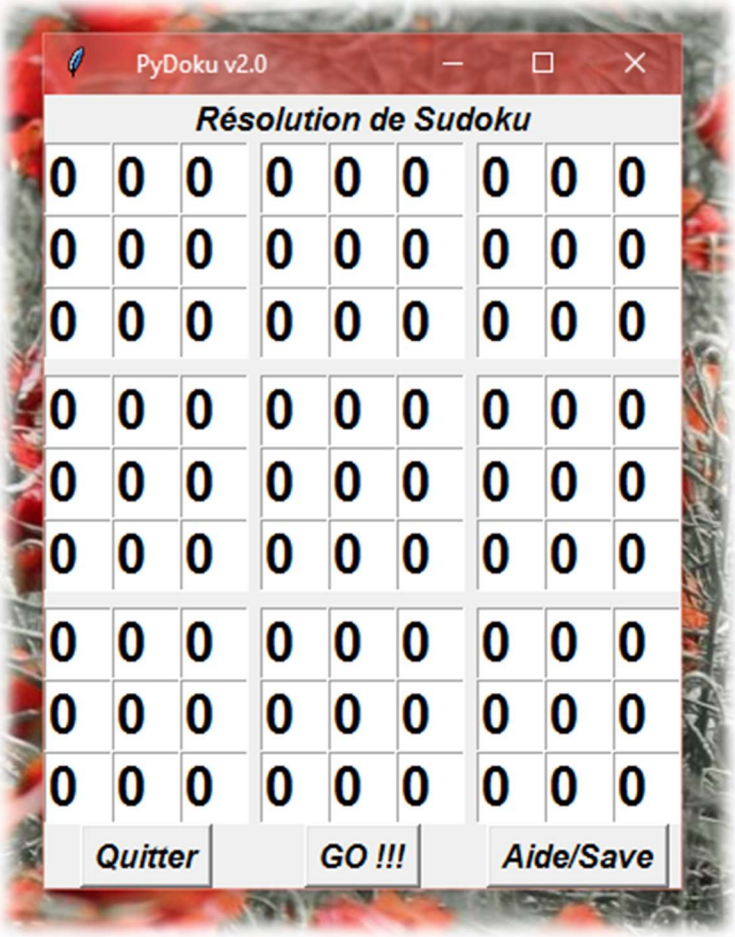
Le but visé au début du projet était de résoudre n'importe quel sudoku que l'on trouve dans les journaux, sur internet…de façon totalement automatique donc à l'aide d'un programme : le projet sudoku est né. Pour atteindre ce but, nous nous sommes tourné vers le langage Python d'abord parce que nous avons appris Python tout au long de l'année et donc nous le maitrisions suffisamment bien pour créer un programme complet dessus puis parce que nous avons estimez que le projet pouvait être créé sur ce langage. Nous nous sommes mis à trois parce que nous avons estimé que c'était suffisant pour notre projet.

La démarche de projet qui a conduit au résultat tel que présenté

Au tout début, nous sommes parti sur une résolution aléatoire, c'est-à-dire que le programme rempli aléatoirement le sudoku en entier puis, en fonction du nombre d'erreur par blocs, le bloc où il y a le plus d'erreur est une nouvelle fois rempli aléatoirement jusqu'à que le nombre d'erreur diminue. Pour cela, nous avions fait plusieurs fonctions et une fonction main qui coordonne le tout :

```
1  def main():
2      sudoku_initial = interface_debut()
3      coord_zeros = coordzero(sudoku_initial)
4      sudoku_rempli = remplissage_total(sudoku_initial)
5      condition = ([0,0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0,0])
6      while doublon(sudoku_rempli) != condition:
7          sudoku_rempli = remplace_bloc(sudoku_rempli, coord_zeros)
8      interface_fin(sudoku_rempli)
9
10 main()
```

La fonction « interface_debut() » est une fonction qui affiche une interface pour entrer le sudoku à résoudre :



Les cases contenant un 0 sont des cases faire remplir par le programme, on remplace les 0 par les chiffres du sudoku que l'on veut remplir.

La fonction « coord_zero(liste) » détermine les coordonnées des zéros de la liste et renvoie une liste qui contient les coordonnées des zéros.

La fonction « remplissage_total() » rempli le sudoku avec des chiffres aléatoire avec la condition d'éviter les doublons dans le bloc.

La fonction « remplace_bloc() » qui remplit à nouveau aléatoirement le bloc qui contient le plus d'erreur.

Et la fonction « interface_fin() » qui affiche le sudoku rempli.

Une fois la fonction main créé, nous avons testé le programme mais il ne fonctionnait pas… Au début, c'était des erreurs dans le code donc nous l'avons corrigé mais ça ne fonctionnait pas, mais pas d'erreur. Donc le programme

fonctionnait en boucle, sans s'arrêter. Donc on a mis deux print :

```
def main():
    sudoku_initial = interface_debut()
    coord_zeros = coordzero(sudoku_initial)
    sudoku_rempli = remplissage_total(sudoku_initial)
    condition = ([0,0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0,0])
    while doublon(sudoku_rempli) != condition:
        print(sudoku_rempli)
        sudoku_rempli = remplace_bloc(sudoku_rempli, coord_zeros)
    print(sudoku_rempli)
    interface_fin(sudoku_rempli)

main()
```

L'un dans la boucle pour voir la liste avec laquelle le programme travaillait et un autre print a la fin, au cas où l'interface_fin planterai. Nous avons constaté que la liste était modifiée seulement d'un bloc (selon le sudoku entré). On a essayé de résoudre ce problème mais, après quelques recherches, il s'est avéré que c'était impossible comme cela, le bloc modifié ne pouvait pas être le bloc avec le moins d'erreur, donc il modifiait seulement un seul bloc indéfiniment :

```
[[6, 1, 7, 9, 8, 3, 1, 7, 2], [9, 4, 8, 6, 5, 4, 3, 8, 6], [5, 2, 3, 1, 7, 2, 4, 9, 5], [3, 6,
7, 5, 2, 3, 6, 8, 6, 4]]
[[6, 4, 5, 9, 8, 3, 1, 7, 2], [2, 9, 7, 6, 5, 4, 3, 8, 6], [3, 8, 1, 1, 7, 2, 4, 9, 5], [3, 6,
7, 5, 2, 3, 6, 8, 6, 4]]
[[3, 7, 5, 9, 8, 3, 1, 7, 2], [1, 6, 9, 6, 5, 4, 3, 8, 6], [4, 8, 2, 1, 7, 2, 4, 9, 5], [3, 6,
7, 5, 2, 3, 6, 8, 6, 4]]
[[1, 5, 8, 9, 8, 3, 1, 7, 2], [7, 6, 9, 6, 5, 4, 3, 8, 6], [2, 3, 4, 1, 7, 2, 4, 9, 5], [3, 6,
7, 5, 2, 3, 6, 8, 6, 4]]
[[7, 2, 9, 9, 8, 3, 1, 7, 2], [4, 3, 1, 6, 5, 4, 3, 8, 6], [8, 5, 6, 1, 7, 2, 4, 9, 5], [3, 6,
7, 5, 2, 3, 6, 8, 6, 4]]
[[2, 6, 1, 9, 8, 3, 1, 7, 2], [7, 8, 9, 6, 5, 4, 3, 8, 6], [4, 3, 5, 1, 7, 2, 4, 9, 5], [3, 6,
7, 5, 2, 3, 6, 8, 6, 4]]
[[8, 3, 5, 9, 8, 3, 1, 7, 2], [7, 1, 4, 6, 5, 4, 3, 8, 6], [9, 2, 6, 1, 7, 2, 4, 9, 5], [3, 6,
7, 5, 2, 3, 6, 8, 6, 4]]
[[8, 5, 2, 9, 8, 3, 1, 7, 2], [1, 7, 9, 6, 5, 4, 3, 8, 6], [6, 3, 4, 1, 7, 2, 4, 9, 5], [3, 6,
7, 5, 2, 3, 6, 8, 6, 4]]
[[5, 1, 3, 9, 8, 3, 1, 7, 2], [8, 4, 9, 6, 5, 4, 3, 8, 6], [6, 2, 7, 1, 7, 2, 4, 9, 5], [3, 6,
7, 5, 2, 3, 6, 8, 6, 4]]
[[9, 2, 8, 9, 8, 3, 1, 7, 2], [3, 1, 4, 6, 5, 4, 3, 8, 6], [6, 5, 7, 1, 7, 2, 4, 9, 5], [3, 6,
7, 5, 2, 3, 6, 8, 6, 4]]
```

Les trois premiers éléments des trois premières listes se modifient indéfiniment car le nombre d'erreur de ce bloc est toujours plus élevé par rapport aux autres blocs. Ensuite, on a mis un temps de pause pour voir les premières listes et le programme, au lancement, modifie un peu tous les blocs puis reste bloqué sur un bloc.

Suite à cela, nous avons décidé de changer d'algorithme, nous avons donc choisi la méthode « backtracking ». Cette méthode, va faire une boucle dans laquelle le programme va rechercher les possibilités dans chaque cases et va choisir la case où il y à le moins de possibilité. Ensuite il va la remplir avec un des chiffres possible. Pour faire ce programme, on a réutilisé les fonctions de l'ancien programme comme l'interface et ce programme fonctionnait ! Mais, on a remarqué que, pour les sudokus compliqués, le programme ne fonctionnait pas, l'erreur était que, si il n'y a pas de possibilité, le programme plantait. Donc on a fait en sorte que le programme, lors de l'erreur, retournait en arrière pour changer le chiffre précédant. Une fois cela fait, et les bugs restant corrigé, le programme était enfin entièrement fonctionnel, pour résoudre les sudokus, même les plus durs.

La dimension collaborative du projet liée au travail en équipe

Au début du projet, lors les versions avec la méthode aléatoire, nous avions déterminé les fonctions à créer pour faire fonctionner le programme. Nous nous sommes donc partager les fonctions. J'ai fait l'interface, une fonction disponibilité colonne, Ricardo a fait la fonction doublon, une fonction disponibilité bloc, Rita a fait la fonction de disponibilité ligne et case et coordonnée des zéros. Nous avions fait la fonction main et corriger les bugs à trois.

Comme cette méthode ne fonctionnait pas, on a décidé de faire autrement. Avec Ricardo et Rita, on a décidé de partir sur la méthode de backtracking. On a décidé de faire des programmes complets de nos côtés et de les mélanger. Ricardo à faire un programme récursif mais avec une erreur d'indice. J'ai fait un programme itératif mais qui résolvait seulement les sudokus les plus faciles car pas de retour en arrière. A la fin, on a réuni les deux programmes, on a gardé la méthode itératif de mon programme, on a gardé le retour en arrière du programme de Ricardo. On a fait un programme à partir des deux autres. Le programme que l'on a obtenu fonctionne parfaitement !

Conclusion

Ce projet nous a donné un aperçu du travail en groupe sur un programme, on a pu s'entrainer à résoudre des problèmes sur le programme. On a donc un programme qui fonctionne, qui ne plante pas et donc on a obtenu exactement le programme que l'on voulait depuis le début.

Programme récursif de Ricardo :

```
def ordre(sudoku:list, coordzero:list):
    res = []
    for x in range (9):
        for y in range (9):
            if [x, y] in coordzero:
                disp = disponibilite_sur_case(sudoku, (x,y))
                res_inter = [(x,y), disp]
                res.append(res_inter)
    final = []
    i = 1
    while i < 10:
        for x in range (len(res)):
            if len(res[x][1]) == i:
                final.append(res[x])
        i += 1
    return final


def commun(sudoku:list, sudoku_entamme:list, coord:tuple):
    disp1 = disponibilite_sur_case(sudoku, coord)
    disp2 = disponibilite_sur_case(sudoku_entamme, coord)
    res = []
    for i in range(1,10):
        if i in disp1 and i in disp2:
            res.append(i)
    return res




def remplissage(sudoku:list, sudoku_modif, coordzero, N):
    order = ordre(sudoku, coordzero)
    print(sudoku_modif, N)
    if commun(sudoku, sudoku_modif, order[N][0]) != [] and N != len(order):
        disp = commun(sudoku, sudoku_modif, order[N][0])
        random.shuffle(disp)
        x = order[N][0][0]
        y = order[N][0][1]
        sudoku_modif[x][y] = disp[0]
        remplissage(sudoku, sudoku_modif, coordzero, N+1)
    elif commun(sudoku, sudoku_modif, order[N][0]) == [] and N != len(order):
        remplissage(sudoku, sudoku_modif, coordzero, N-1)
    elif N == len(order):
        return sudoku_modif

def main():
    sudoku_initial = interface_debut()
    coord_zeros = coordzero(sudoku_initial)
    sudoku_final = sudoku_initial
    fin = remplissage(sudoku_initial, sudoku_final, coord_zeros, 0)
    interface_fin(fin)

main()
"""
```

Mon programme itératif :

```python
from interfaces import *
from disponibilite import *
import random

def creer_liste(liste_user):
    liste_dispo = []
    for i in range(9):
        for j in range(9):
            a_traiter = liste_user[i][j]
            if a_traiter == 0:
                dispo_ligne = disponibilite_sur_ligne((i, j), liste_user)
                dispo_colonne = disponibilite_sur_colonne((i, j), liste_user)
                dispo_bloc = disponibilite_sur_bloc((i, j), liste_user)
                nb_dispo = []
                for k in range(1,10):
                    if k in dispo_ligne and k in dispo_colonne and k in dispo_bloc:
                        nb_dispo.append(k)
            else:
                nb_dispo = [0]
            liste_dispo.append(nb_dispo)
    return liste_dispo


def main():
    liste_user = interface_debut()
    mémoire = []
    while True:
        liste_dispo = creer_liste(liste_user)
        comptage = []
        for i in range(81):
            print(i)
            if liste_dispo[i] == [0]:
                comptage.append(10)
            else:
                comptage.append(len(liste_dispo[i]))
        petit = 10
        for i in range(81):
            if comptage[i] < petit:
                petit = comptage[i]
                position = i
        if liste_dispo[position] == []:
            position = prec

        if petit == 10:
            break
        prec = position
        a = position // 9
        b = position % 9
        print(liste_user)
        random.shuffle(liste_dispo[position])
        liste_user[a][b] = liste_dispo[position][0]
        mémoire.append(position)



    print(liste_user)

main()
```