GENERATEUR

Nous avons choisi d'implanter un générateur de Sudokus 9*9.
Nous allons présenter ici notre solution.

Structure de données et variables principales :

Le type matrice est donc utilisé. Nous avons 3 matrices principales.

```
memo:=table():Solution:=matrix(9,9,0):Sudoku:=matrix(9,9,0):
```

une première qui subit nos essais (*matrice*). Une matrice *Solution* contenant la solution du Sudoku et une matrice *Sudoku* qui est la grille de jeu.
Nous disposons aussi d'un type table() (*memo)* qui nous permet de mémoriser *matrice* après chaque modification. On accède à un état précis de *matrice* par un indice *compteur*. *memo[compteur]* nous renvoie une sous-table de la forme *[j,k,matrice]* où *j* et *k* représentent les coordonnées de la case qui a été modifiée à l'étape *compteur* et *matrice* reflétant l'état à *compteur*. La clarté s'imposera d'elle-même par la suite.
Nous disposons aussi d'ensembles et de leurs opérations ensemblistes pour entre autres définir :

```
chiffresPossibles:= {$11..19};
tous:= {0} union chiffresPossibles;
```

Il sera expliqué par la suite pourquoi *chiffresPossibles* contient des possibilités de 11 à 19 (normalement de 1 à 9).

Signatures des fonctions utilisées :

- `init:=proc() global Voisins, chiffresPossibles, tous;`
  - `elem:= proc(ii, jj)`

Nous faisons appel à une fonction d'initialisation *init()* qui nous permet de générer les voisins d'une cellule où { v[1,1] } représente les voisins de la cellule [1,1] de la forme {bloc,ligne puis colonne} ou chaque élément appelé $matAuxiliaire_{x,x}$ correspond à la cellule [j,k] qui est voisine où *matAuxiliaire* n'est pas attribué. On substitue ensuite pour retrouver les valeurs de *matrice* (dans *Candidater()).*
Fonctionnement : *init()* est composée d'une sous fonction *elem(),* prenant en paramètre 2 entiers repérants une case, qui se charge d'attribuer à la case [i,j] ses voisins.
On repère le début du bloc grâce à `ix:= iquo(ii-1,3)*3; jx:= iquo(jj-1,3)*3;`
On calcule *X/3* avec *X* de 1 à 9 qui représente l'indice i ou j de la case, *on obtient (0, 0, 0, 1, 1, 1, 2, 2, 2*), on multiplie par 3 pour obtenir (0, 0, 0, 3, 3, 3, 6, 6, 6).
Ainsi, après avoir calculer le bloc, la ligne puis la colonne pour une case, on construit notre matrice *Voisins* grâce à `Voisins:=Matrix(9,9,elem):`

- `Candidater:=proc() global matrice, Voisins, chiffresPossibles;`

On appelle candidat pour une case vide les chiffres non interdits de la case parmi *tous* selon les règles du sudoku. Affecter un chiffre dans cet algorithme signifie que c'est la seule possibilité pour cette case.
On examine toutes les cellules de la matrice successivement du coin haut gauche au coin en bas à droite.
à [j,k], la cellule contient donc les chiffres qui ne sont pas interdits pour cette cellule. Si l'ensemble dans [j,k] ne contient qu'une seule possibilité (singleton) alors il ne reste plus rien à faire avec cette cellule.
C'est ici que l'on se sert de la matrice *Voisins* afin de juste examiner les chiffres déjà affectés dans le groupe de la cellule [j,k]. Si on appelle cette fonction de l'algorithme *pick* une erreur est retournée signifiant que l'ensemble des valeurs pour une case est vide ou ne contient que le zéro. Nous verrons ce cas lorsque nous parlerons de *pick.*
Un booléen *changement* nous indique qu'un chiffre a été affecté ce qui peut changer les autres cases.
Algorithme: changement =false ;

Pour i de 1 à 9 **faire**
Pour j de 1 à 9 **faire**
Si taille matrice(i,j)>1 **alors**
Tmp= chiffresPossibles minus (chiffres affectés dans cellules voisines)
si Tmp=1 **alors** changement=true; matrice[i,j]=Tmp;
sinon matrice[i,j]:= (Tmp union {0}) intersect matrice[i,j];
fin si;
fin si;
fin pour;
fin pour;

Si changement=true **alors** Candidater() ;
Sinon si member({0},matrice) **alors** erreur « possibilité 0 affectée à une case » **fin si** ;
fin algo ;

- **`verif_grp:=proc() global matrice, chiffresPossibles;`**

Affecter un chiffre dans cet algorithme signifie que c'est la seule possibilité pour toutes les cellules du groupe excepté pour cette case. On appelle *Candidater()* pour raccourcir les possibilités de chaque case. Ensuite on examine tous les groupes les uns après les autres et on vérifie si on peut à chaque fois affecter un chiffre. Ici aussi un booléen représente si un changement a été effectué durant la procédure. Il est cependant vérifié à la fin lorsque l'on a examiné tous les groupes.
Les possibilités sont de 11 à 19 car lorsque l'on examine *ici* dans l'algorithme détaillé ci-dessous, on obtient une liste de la forme [[i,j,matrice[i,j]] [i1,j2,matrice[i2,j2]]…] Si nos possibilités sont égales aux coordonnées i,j alors on ne pourra pas efficacement trouver nos chiffres isolés.
on regroupera selon que k=11..19,
Exple : Dans ligne=1 on a :
{0,11,12,13,14},{0,11,13,14},{0,11,13,16,18,17}… on remarque que le 2 dans la première case est le seul chiffre possible pour tout le groupe donc on affecte le 2. On obtient pour k de 11 à 19 :
tmp=[[1,1,{0,11,12,13,14}],[1,2,{0,11,13,14}],[1,3,{0,11,13,16,18,17}]…] ;
k=11 -> ici=[[1,1,{0,11,12,13,14}],[1,2,{0,11,13,14}],[1,3,{0,11,13,16,18,17}]];
k=12 -> ici=[[1,1,{0,11,12,13,14}]] ;
k=13 -> ici=[[1,1,{0,11,12,13,14}],[1,2,{0,11,13,14}],[1,3,{0,11,13,16,18,17}]];µ
On a pour k=12 une seule possibilité (12) et comme le zéro est présent cela signifie que la case n'a pas de chiffre affecté.
Algorithme: Candidater(); changement =false ;

```
Pour a de 0 à 2 faire
    Pour b de 0 à 2 faire //sélectionner les différents blocs
    Construire liste de la forme tmp=[i,j, matrice[i,j]] pour tous les blocs
        Pour k dans chiffresPossibles faire ici=select(has,k,tmp)
            Si taille(ici)=1 et has(ici,0) alors //seule possibilité, case pas affecté
            matrice[ici[1,1],ici[1,2]]={k}; changement=true ;
            fin si;
        fin pour;
fin pour; fin pour;

//Examiner lignes et colonnes de la même manière
Si changement=true alors verif_grp(); fin si;
fin algo ;
```

- **`pivot:= proc() global matrice, encore, minimalCase, minimalCasePlus;`**

Renvoie la case avec le moins de possibilités (*minimalCase*) ou ce chiffre plus 1 (*minimalCasePlus*) pour plus de rapidité. La case est choisie aléatoirement parmi les cases non instanciées.
encore représente le nombre de cases restantes.

- **`pick:= proc(j,k) global matrice, compteur, memo, essai;`**

pick prend en paramètre j,k représentant le pivot choisi.
Algorithme : *caseCourante=matrice[j,k] ;*
//On enlève le 0 de la liste pour ne pas le choisir parmi les essais.
caseCourante= caseCourante minus {0} ;
memo[compteur]=j,k,copy(matrice) ; *//On sauvegarde*
tant que taille(caseCourante)>0 **faire**
Choisir une possibilité (de manière aléatoire) de la caseCourante.
try verif_grp() ; break ; *// essayer de résoudre avec l'essai.*
//Si on n'obtient pas d'erreur de Candidater(),le chiffre mène vers une solution alors on break et on choisit un autre pivot.
catch : *//Sinon on n'a pas obtenu de solutions avec ce chiffre, on le supprime de l'ensemble et on essaie un autre chiffre dans l'ensemble (fin de l'algo).*
//En revanche si c'était la dernière possibilité, alors nous devons backtracker[1].
si taille(matrice[j,k]=1) **alors**
//pour ne pas revenir à l'état initial

[1] Backtrack: Action de revenir en arrière.

```
        tant que compteur > 1 faire
                // revenir a l'état précédent
                compteur=compteur-1
                //reprendre la matrice précédente
                matrice:= copy(memo[compteur][3]);
                //reprendre les coordonnées de la case précédemment modifiée j,k
                loc:= memo[compteur][1..2];
                matrice[loc]:= memo[compteur][3][loc] minus memo[compteur+1][3][loc];
                //Enlever la possibilité qui a échoué a compteur+1
                try verif_grp()
                        //pour que note matrice reflète les possibilités sans l'essai qui a échoué.
                        //S'il y a une erreur de Candidater() alors on continue de backtracker. Sinon on
                        s'arrête(break)
                        compteur=compteur-1 ; break ;
                catch :
                end try ;
        fin pour ;
fin si ;
caseCourante= caseCourante minus {essai};
matrice:= copy(memo[compteur][3]);
end try;
fin pour; fin algo;
```

Syntaxe du try :
Exemple diviser a par b lorsque a=2 et b=0 ;
Si l'on fait a/b Maple renvoie une erreur `Error, numeric exception: division by zero`
Avec le try on obtient :

```
try a/b
catch "numeric exception: division by zero":
error "Division par 0 (%1/0)",a
end try;
```

```
Error, Division par 0 (2/0)
```

Le catch permet d'attraper une erreur et de ne pas la retourner.
Donc dans le cas du backtrack, si l'on enlève le catch et que l'on doit backtracker deux fois de suite, on obtiendra l'erreur de *Candidater()* et l'arrêt du programme.

Limitations
Nous avons limité notre implantation du générateur à des Sudokus de forme carrée et de 9*9cases. Il existe cependant des Sudokus de forme rectangulaire (3 blocs de 4 cases chacun en longueur et l'inverse en largeur) ou de dimension plus grande (12*12cases ou bien 16*16) et il est facile de faire évoluer notre programme pour les gérer. Il suffit en effet de remplacer les chiffres Possibles par les chiffres adéquats et de remplacer nos dimensions (3,3,9) par des lettres (m,n,N par exemple) permettant de définir nos dimensions. On appellerait init(m,n) et on remplacerait les chiffres par ces lettres dans tous nos algorithmes.
N'étant pas demandé dans le sujet nous avons décidé de ne pas implémenter ceci dans l'interface graphique afin de ne pas surcharger le code. Cependant ce travail a été effectué, et nous parvenons à générer des sudokus de très grande dimension.

Performances
Notre programme est capable de générer des Sudokus de différents niveaux et de différentes dimensions avec une unique solution assez rapidement pour les sudokus 9*9cases. (~1à2secondes).
Nous partons d'une grille complètement instanciée (81 chiffres instanciés) et nous enlevons des chiffres jusqu'à ce qu'il y ait une solution unique.
Nous pouvons générer jusqu'à 2 niveaux différents divisés en 2sous groupes. Cependant nous avons décidé d'implémenter dans notre partie graphique seulement deux niveaux. Nous pouvons enlever entre 40 et 50 chiffres ou bien entre 50 et 60 chiffres.
Nous pouvons donc faire des grilles faciles enlevant entre 40 et 50 cases et le séparer en 2 sous partie [FACILE – MOYEN] ainsi que des grilles difficiles ou l'on ne laisse qu'entre 20 et 30 cases apparentes [DIFFICILE – DIABOLIQUE]. Il sera expliqué plus tard comment il est possible de trouver ses deux sous groupes.

Méthode utilisée pour générer une grille complète :

- Nous remplaçons les 81 éléments de notre matrice par une liste contenant toutes les possibilités {$11..19}.

```
matrice:=matrix(9,9,(i,j)->tous):
```

- Ensuite nous choisissons une case aléatoire dans laquelle nous affectons un chiffre parmi ceux que la case contient. On appelle *pivot*. Notre algorithme s'arrête donc lorsqu'il n'y a plus de cases restantes donc lorsque la grille est complète (encore=0).
  On appelle la fonction « Pick » avec le pivot.
- Une fois que pick a affecté un chiffre, nous essayons de résoudre notre grille en « candidatant » notre grille et en cherchant un chiffre isolé dans un groupe (respectivement Candidater() et verif_grp()). Ici trois possibilités s'offrent a nous (après plusieurs répétions des deux derniers points) :
  - La grille est résolue, c'est-à-dire que l'on a trouvé une solution respectant les contraintes (en général, après une quarantaine de chiffres instanciés, nous parvenons à une solution) donc il ne reste plus que des ensembles à un élément.
  - La grille n'est pas résolue, il reste des cases où il y a des possibilités.
    on rappelle donc pivot pour choisir la case avec le minimum de possibilités suivi d'un appel a pick pour instancier une valeur au hasard et essayer de résoudre.
    La grille ne mène à aucune solution. C'est-à-dire qu'il ne reste dans une ou plusieurs cases que la possibilité d'instancier le chiffre 0.
    Alors nous backtrackons comme détaillé dans l'algorithme pick au dessus jusqu'à ce que nous trouvions une solution.
    Le backtrack n'est pas systématique, nous pouvons trouver une série de chiffres qui nous amènent à une solution et ce aléatoirement.
    On répète ses étapes autant de fois que nécessaire jusqu'à avoir note grille complète.

Méthode utilisée pour trouver une grille à solution unique :
Maintenant que nous avons une grille complète, nous pouvons passer à l'étape suivante qui consiste à enlever des chiffres.

```
casesRestantes:= {seq(seq([j,k],j=1..9),k=1..9)} :
casesEnlevees:= NULL:
```

{4}	{2}	{7}
{0, 1, 9}	{0, 1, 9}	{5}
{3}	{8}	{6}
{2}	{4}	{9}
{7}	{6}	{1}
{5}	{3}	{8}
{8}	{7}	{2}
{0, 1, 9}	{0, 1, 9}	{3}
{6}	{5}	{4}

On crée un ensemble contenant toutes les coordonnées des cases de la matrice, et une pile vide au départ correspondant aux cases enlevées. Nous choisissons avant de démarrer une *méthode*[2] correspondant à la manière de procéder pour la suite. On choisit une case *essai* parmi casesRestantes, on la rajoute dans casesEnlevees. *Matrice[Essai]* reçoit *tous*. On appelle *méthode()*. Après plusieurs cases enlevées on obtient une matrice avec des trous ayant leur possibilités a tous avant *methode()* et les candidats après *methode()*. Jusqu'à un certain moment, nous parviendrons à trouver par simple appel a *methode()* la même solution qu'initialement.
Par la suite, nous enlèverons une case qui engendra une possibilité de *permutations*[3] entre d'autres cases (cases entourées en rouges sur l'image). Donc nous la fixons pour être sûr de ne trouver qu'une seule solution.

Choix des niveaux
C'est donc à partir d'ici que nous avons la possibilité de choisir notre premier groupe de niveaux.
Les chiffres restants sont représentés en tant qu'élément à un ensemble. La case engendrant la permutation a été fixée mais c'est la manière avec laquelle on fixe cette case qui définie le niveau.
En effet, lorsque l'on enlève une case, on rappelle *methode()* pour bien vérifier que l'on ait qu'une solution.
Si on a une permutation alors on fixe simplement toutes les cases restantes jusqu'à *casesRestantes={ }*.
On obtient un niveau FACILE-MOYEN (~45 cases enlevées)
Cependant si lorsque l'on trouve une permutation qui peut engendrer une autre solution, on fixe la case

[2] Nous avons le choix entre Candidater() qui applique simplement les candidats à toutes les cases et verif_grp() qui en plus de Candidater cherche le chiffre isolé d'un groupe
[3] On peut échanger x chiffres dans un sudoku et trouver une autre solution. La matrice contient au moins 2 cases avec un ensemble contenant des candidats

coupable de cela et que l'on continue en rappelant *méthode()* tout en prenant compte de la case qui a généré une autre solution, nous pouvons minimiser notre grille. Si on trouve une case qui cause une permutation alors la fixe. Mais si la case ne cause pas de permutation, alors on peut l'enlever.

```
chiffresJoueurs:{{11},{12},{13},{14},{15},{16},{17},{18},{19}};
```

Ce test est effectué en convertissant la matrice en une liste. Si on obtient simplement les chiffresJoueurs alors on peut enlever cette case. Mais si il y a permutation alors on obtiendra *chiffresJoueurs union {la permutation}*. Donc autre solution donc on doit fixer cette case.
Bien entendu, *casesRestantes* et *casesEnlevees* sont mis à jour à chaque changement.
Il est alors possible de diviser nos 2 niveaux en 2sous groupes en modifiant la méthode à utiliser.
Candidater() représente grossièrement la méthode de découpage qu'un humain peut faire pour résoudre un sudoku. Alors que *verif_grp()* est une méthode de recherche plus sophistiquée qui permet de rechercher les chiffres isolées et que l'humain ne parvient pas aisément à reproduire. La grille est donc plus difficile à résoudre.

Principales difficultés

- Nous avons eu quelques difficultés à implémenter une méthode récursive qui consistait juste à instancier un chiffre et à rappeler la même fonction avec la modification faite (Comme *Candidater()*) Le principe de récursivité n'avait pas été saisi de ce point de vue.
- La manière de générer rapidement les voisins (*init()*) n'a pas été évidente du moins la transformation de boucles FOR vers les SEQ et d'imbriquer la fonction elem dans init.
- La manière algorithmique de trouver une seule solution possible pour un sudoku sachant que nous n'avions pas de compteur de solution assez rapide à ce moment là.
- La difficulté de générer une grille à niveau.
  Nous pensions qu'il suffisait d'enlever plus de chiffres pour obtenir une grille difficile mais cela ne suffisait pas.
- Trouver un moyen efficace de rappeler la fonction *Candidater()* ou *verif_grp()* lorsque un chiffre a été affecté. Car si un chiffre est affecté dans *Candidater()* alors il est possible de trouver d'autres chiffres. Cependant la méthode utilisée n'est pas optimale, un booléen est utilisé, initialisé a FALSE au départ, il est mis a TRUE lorsqu'un chiffre est trouvé et s'il est vrai à la fin des fonctions on rappelle la fonction récursivement.
  Et il existe sûrement une méthode de mettre à jour la matrice instantanément dès qu'une affectation a eu lieue.