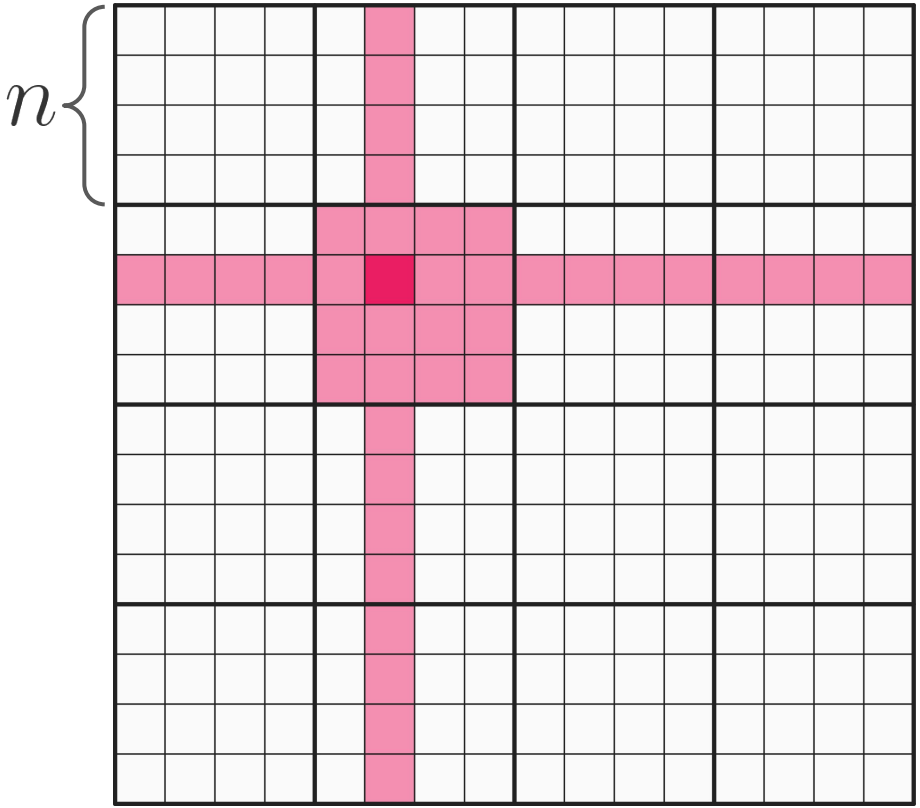
Résolution d'une grille de Sudoku

Hugo SALOU, Timothée SALOMON, Thomas LEGUÉRÉ, Bastien FAVRE

Plan

1. Présentation du problème
2. Méthode de programmation utilisée
3. Structures et algorithmes
4. Implémentation en C
5. Conclusion

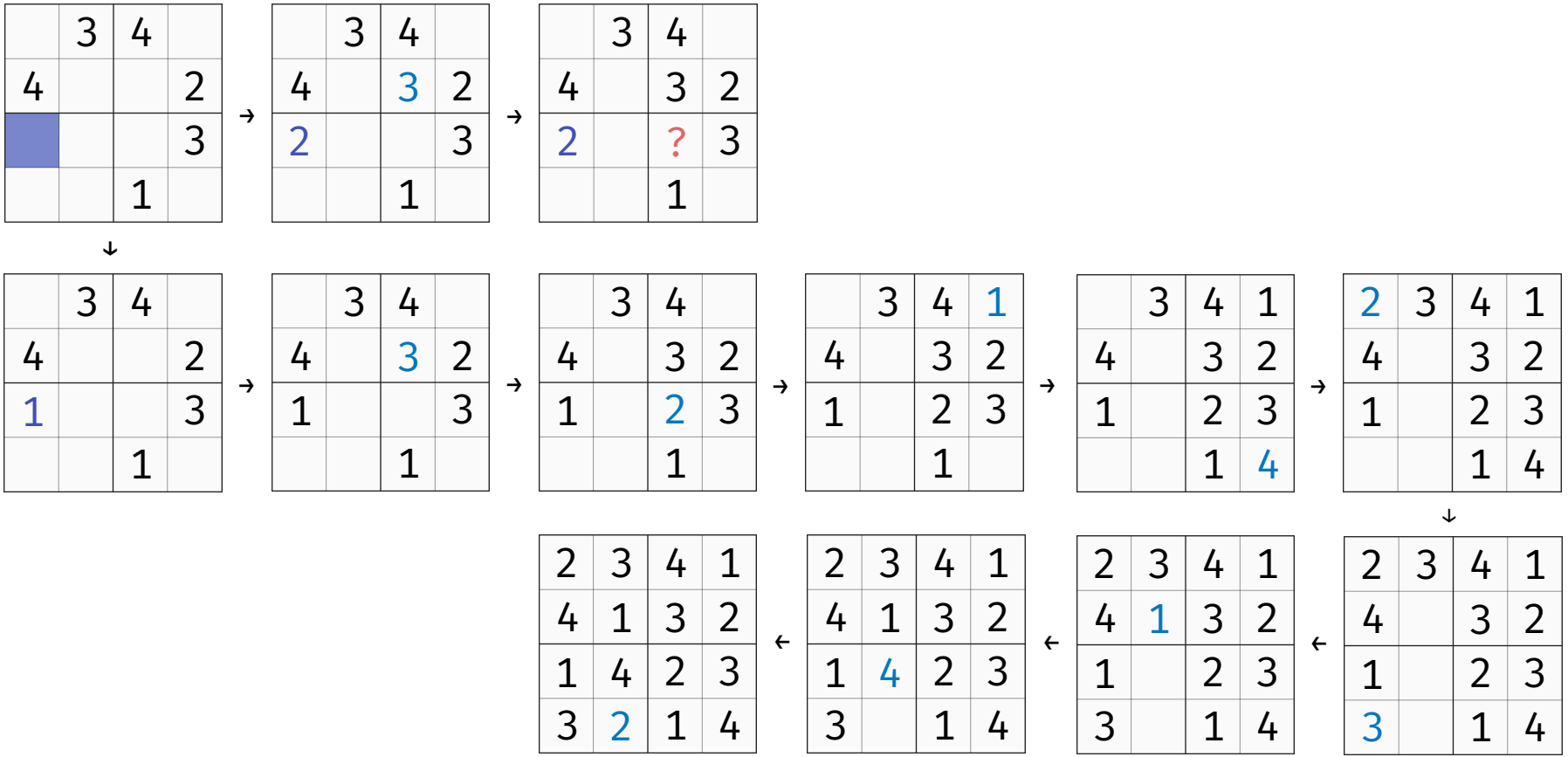
Retour sur le principe du Sudoku



Une même valeur ne se retrouve pas sur la même **ligne**, **colonne** ou **bloc**.

1. Présentation du problème

Un exemple



2. Méthode de programmation utilisée : le backtracking

- Méthode d’exploration exhaustive

- Choix successifs

- Retour en arrière si une contradiction est atteinte

2. Méthode de programmation utilisée

Le backtracking dans notre problème

Problème : complexité élevée

→ Liste des cases à remplir classée par ordre croissant de possibilités

3. Structures et algorithmes

Structures

- Enregistrement `cases` : case du Sudoku

- Enregistrement `noeud` : nœud de la liste chaînée → `cases`

3. Structures et algorithmes

Algorithme de vérification de la validité d'un nombre

```
fonction Validité
Début
  Pour i = 1 à n² faire
    Si sudoku[i][y] == cellule.valeur alors
       Retourne faux

  Pour j = 1 à n² faire
     Si sudoku[x][j] == cellule.valeur alors
       Retourne faux

  Pour i = 1 à n faire
     Pour j = 1 à n faire
       Si sudoku[i + n × ⌊x / n⌋][j + n × ⌊y / n⌋] == cellule.valeur alors
          Retourne faux

  Retourne vrai
Fin
```

ligne

colonne

bloc

Complexité en $\mathcal{O}(n^2)$

3. Structures et algorithmes

Algorithme de résolution du Sudoku

```
fonction Résolution
Début
  Tant qu'il y a des éléments dans la liste faire
    Pour i = 1 à n² faire
      Si i est valide alors
        sudoku[x][y].valeur = i
        liste.suivant

    Si aucune valeur de i n'est valide et l'élément n'est pas le premier alors
      liste.précédent
Fin
```

Invariant de boucle

$\forall\, c_{ij} \in$ liste avec $i \in [\![1, n]\!],\ j \in [\![1, n]\!]$

Soient $v_{\text{temporaire}}$ la valeur de la case à chaque itération et

v_{finale} la vraie valeur finale que prendra la case.

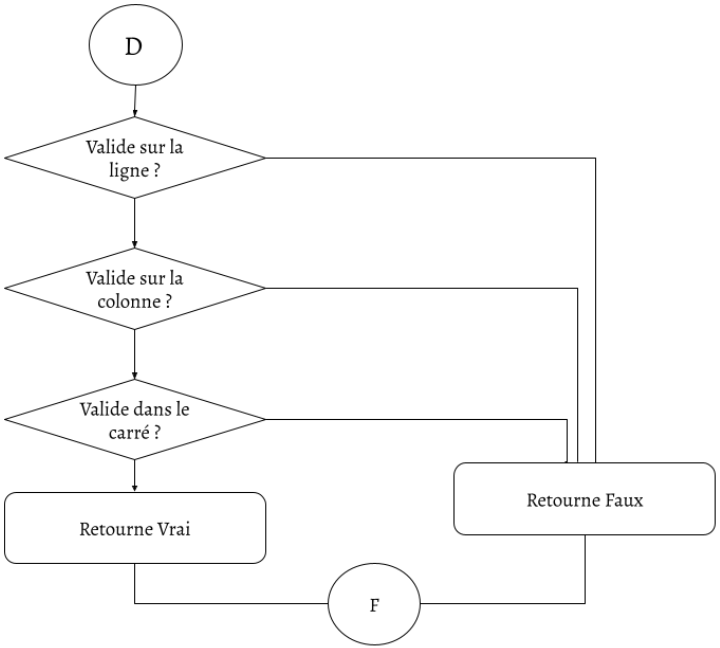
L'algorithme vérifie pour chaque itération :

$$v_{\text{temporaire}} \leqslant v_{\text{finale}}$$

3. Structures et algorithmes

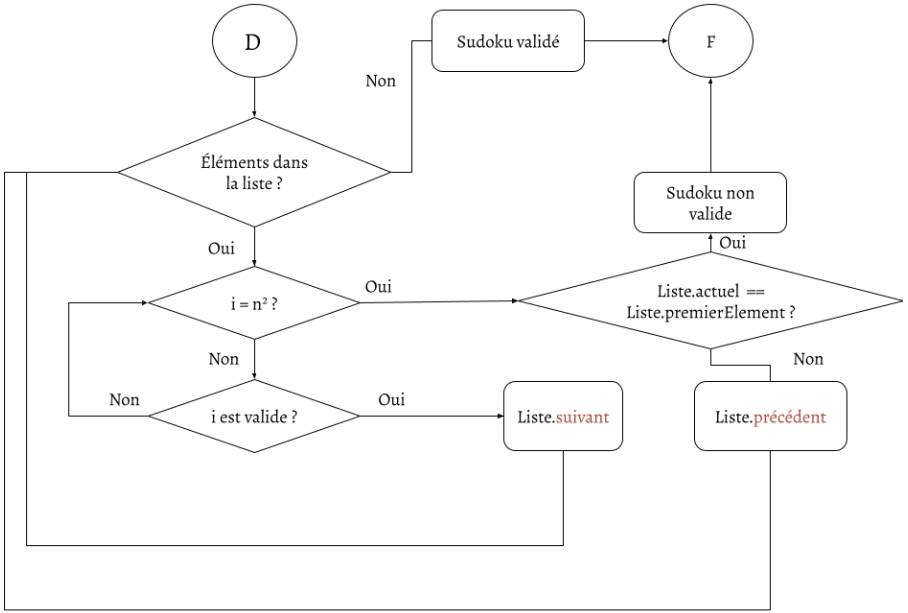
Fonction validité :

Terminaison valide :



Fonction Résoudre :

Terminaison valide :



4. Implémentation en C

Types

- Enregistrement pour les cases du sudoku :

```
typedef struct {
  int possibilites;
  unsigned int valeur;
  couple coordonnees;
} cases;
```

```
typedef struct {
  unsigned int y;
  unsigned int y;
} couple;
```

4. Implémentation en C

Types

- Enregistrements pour la liste chaînée stockant les cases à traiter

```
typedef struct str_noeud {
    cases carre;
    struct str_noeud* suivant;
    struct str_noeud* precedent;
} noeud;
```

```
typedef struct tete_liste {
    noeud* premierElement;
    int longueur;
} listeChainee;
```

4. Implémentation en C

```c
// Vérification de la validité d'une cellule
// Entrée : - Sudoku
//          - Cellule à vérifier
//          - Taille n
// Sortie : - Booléen : - true si la cellule est valide
//                      - false sinon
// Pré-condition : la cellule fait partie de la grille de Sudoku
// Testée et validée
bool estValide(cases** sudoku, cases cellule, int n) {
    // Déclaration des variables

    int x = cellule.coordonnees.x;
    int y = cellule.coordonnees.y;

    // Vérification sur la ligne
    for (int j = 0; j < n * n; j++) {
        if (sudoku[x][j].valeur == cellule.valeur) {
            return false;
        }
    }

    // Vérification sur la colonne
    for (int i = 0; i < n * n; i++) {
        if (sudoku[i][y].valeur == cellule.valeur) {
            return false;
        }
    }

    // Vérification dans le grand carré de taille n :

    // Retrouve l'abscice du grand carré
    int entierex = x / n;
    // Retrouve l'ordonnée du grand carré
    int entierey = y / n;

    // Boucle de parcours du grand carre
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
            if (sudoku[i + entierex * n][j + entierey * n].valeur == cellule.valeur) {
                return false;
            }
    }

    return true;
}
```

4. Implémentation en C

```c
// Résout le sudoku
// Entrée : - Matrice de cases grille : sudoku
//          - Liste des cases à traiter
//          - Entier n : taille du sudoku
// Pré-conditions : - Sudoku non nul
//                  - Taille non nulle
// Sortie : - Sudoku complété
// Post-condition : - Sudoku valide
// Testée et validée
void remplitSudoku(cases** grille, listeChainee liste, int n)
{
    noeud* noeudActuel = (noeud*)malloc(sizeof(noeud));
    assert(noeudActuel != NULL);
    noeudActuel = liste.premierElement;

    // pour chacune des cases à remplir
    while (noeudActuel != NULL)
    {
        // on tente chacune des possibilités pour remplir cette case
        for (int i = noeudActuel->carre.valeur + 1; i <= n * n; i++)
        {
            noeudActuel->carre.valeur = i;
            if (estValide(grille, noeudActuel->carre, n))
            {
                int x = noeudActuel->carre.coordonnees.x;
                int y = noeudActuel->carre.coordonnees.y;

                // La valeur testée est valide, on la place dans la grille
                grille[x][y].valeur = i;

                // On passe à l'élément suivant dans la liste
                noeudActuel = noeudActuel->suivant;
                break;
            }
        }
        if (noeudActuel != NULL && noeudActuel != liste.premierElement && noeudActuel->carre.valeur != 0)
        {
            // On réinitialise à 0 car on est arrivé à une contradiction
            grille[noeudActuel->carre.coordonnees.x][noeudActuel->carre.coordonnees.y].valeur = 0;
            noeudActuel->carre.valeur = 0;

            // On recule d'un élément dans la liste
            noeudActuel = noeudActuel->precedent;
        }
    }
}
```

5. Conclusion

- Donne le résultat mais complexité élevée

  → Exécution parfois très longue

- Possibilités d’amélioration :

  → Mettre à jour la liste à chaque étape

  → Stocker les possibilités pour chaque case