UNIVERSITÉ DE STRASBOURG

Projet – *GRAPHES*
L2 – S4
2014 – 2015

RESOLVEUR DE SUDOKU

HAAS Antoine
MUHARREMAJ Juvena

1 Introduction

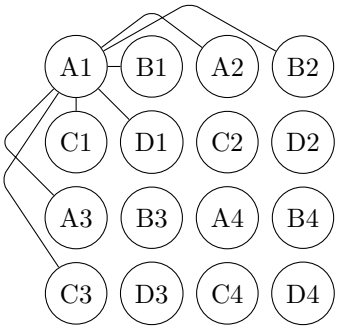
Les jeux de logiques ainsi que les jeux de réflexions occupent une place de plus en plus importante dans la vie quotidienne. Un de ces jeux a pris un tel ampleur qu'une théorie mathématique ainsi que des études théoriques ont été réalisées sur le sujet, il s'agit du Sudoku.

Une des méthodes couramment utilisée, et celle qui nous intéresse pour le cadre du projet, consiste à se servir de la théorie des graphes.

Pour ce faire nous allons utiliser une structure de données adaptée au besoin de ce type de graphes, une fois ceci fait nous allons implémenter un algorithme de backtracking pour la résolution, et enfin pour tester cette implémentation nous allons générer nos propres sudokus.

2 Structure de données

2.1 Représentation graphique

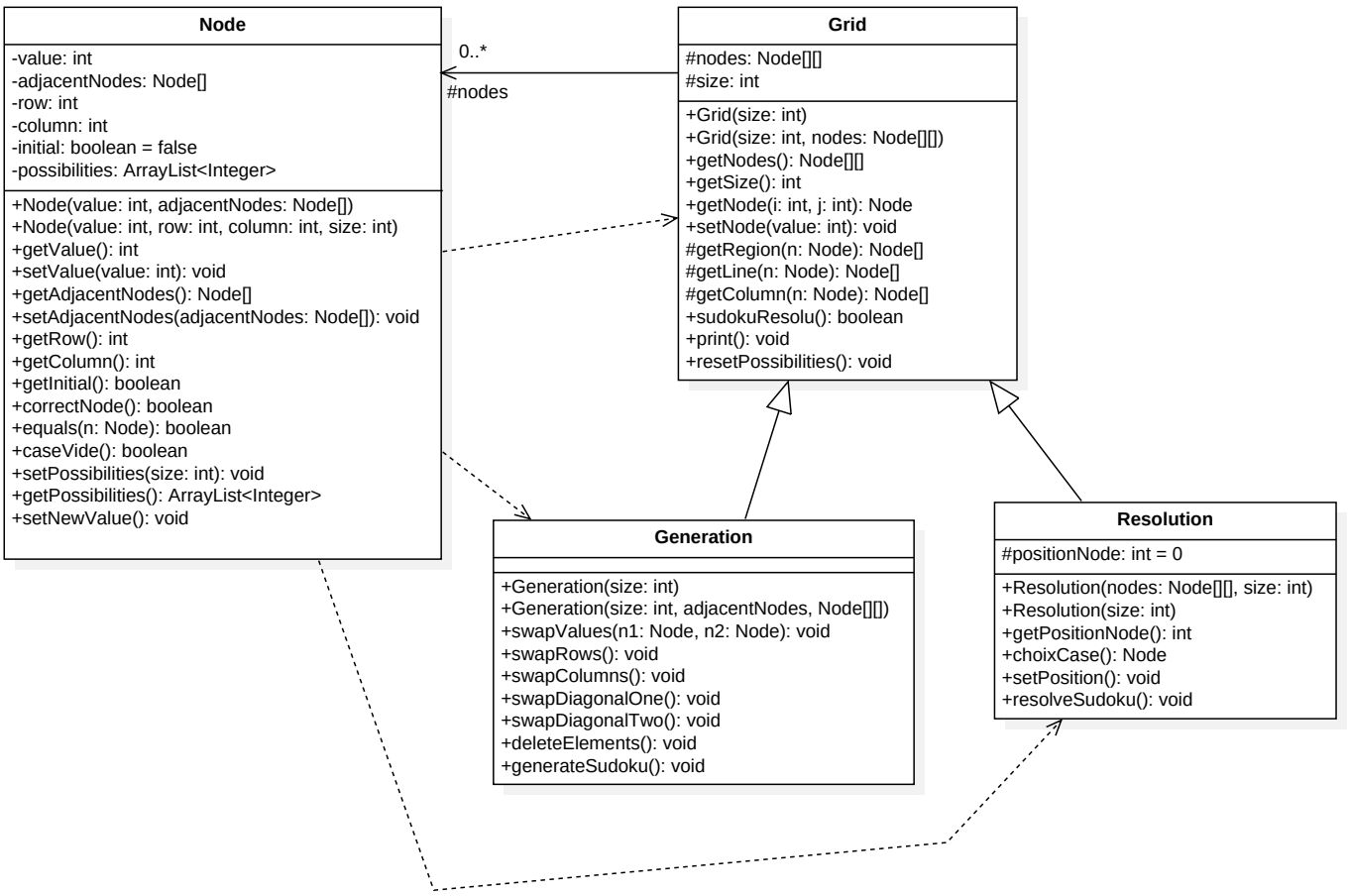


Voici ci-dessus un exemple de graphe qui représente un sudoku de taille 2×2, seul le premier élément a été relié pour éviter d'encombrer la figure.

2.2 Explication

Soit un graphe, noté G=(S,A), avec S l'ensemble de ses sommets et A l'ensemble de ses arêtes. Le sudoku, donc le graphe, est représenté dans un tableau à deux dimensions afin de représenter la grille d'un sudoku et contient tous les sommets de S.
Pour chaque sommet $s \in S$, s est relié au sommets de la même ligne, colonne et région que lui. Soit $a \in A$, tout a de s est représenté par un tableau qui contient les noeuds adjacents à s. Chaque sommet s possède également une liste de possibilités qui contient la valeur que peut prendre chaque noeud.

2.3 Modélisation objet UML


Node
-value: int
-adjacentNodes: Node[]
-row: int
-column: int
-initial: boolean = false
-possibilities: ArrayList<Integer>
+Node(value: int, adjacentNodes: Node[])
+Node(value: int, row: int, column: int, size: int)
+getValue(): int
+setValue(value: int): void
+getAdjacentNodes(): Node[]
+setAdjacentNodes(adjacentNodes: Node[]): void
+getRow(): int
+getColumn(): int
+getInitial(): boolean
+correctNode(): boolean
+equals(n: Node): boolean
+caseVide(): boolean
+setPossibilities(size: int): void
+getPossibilities(): ArrayList<Integer>
+setNewValue(): void
0..*
#nodes
Grid
#nodes: Node[][]
#size: int
+Grid(size: int)
+Grid(size: int, nodes: Node[][])
+getNodes(): Node[][]
+getSize(): int
+getNode(i: int, j: int): Node
+setNode(value: int): void
#getRegion(n: Node): Node[]
#getLine(n: Node): Node[]
#getColumn(n: Node): Node[]
+sudokuResolu(): boolean
+print(): void
+resetPossibilities(): void
Generation
+Generation(size: int)
+Generation(size: int, adjacentNodes, Node[][])
+swapValues(n1: Node, n2: Node): void
+swapRows(): void
+swapColumns(): void
+swapDiagonalOne(): void
+swapDiagonalTwo(): void
+deleteElements(): void
+generateSudoku(): void
Resolution
#positionNode: int = 0
+Resolution(nodes: Node[][], size: int)
+Resolution(size: int)
+getPositionNode(): int
+choixCase(): Node
+setPosition(): void
+resolveSudoku(): void

3 Résolution du sudoku

3.1 Principe

Pour résoudre le sudoku, et à l'aide de la modélisation réalisée, il faut effectuer un coloriage du graphe. Il ne faut pas qu'un sommet ait la même couleur qu'un de ses noeuds adjacents, ici il s'agit des numéros des sommets représentés allant de 1 à 9.
Pour se faire, nous allons utiliser le principe du retour-arrière. Le retour-arrière consiste à employer la stratégie du fil d'Ariane, pour savoir revenir à l'étape précédente, si besoin s'en fait. Ici ce fil conducteur est nécessaire, au cas où lorsqu'une valeur d'un noeud a été choisie et que celle-ci n'était pas la bonne. En effet si la mauvaise valeur a été choisie, à un moment donné lors de la résolution nous serions bloqués et ne pourrions donc plus continuer à résoudre le sudoku dû à l'absence de possibilité d'une valeur pour un noeud. Ainsi le fil d'Ariane nous permet de revenir à l'endroit voulu et de changer la valeur.
Grâce à cette stratégie, on trouve, si le sudoku est résolvable, une bonne valeur pour chaque noeud et on obtient donc bien un sudoku résolu.

3.2 Pseudo-code

3.2.1 Resoudre sudoku

```
précondition : le sudoku doit être résolvable
résoudre_sudoku(Sudoku) = Sudoku
résoudre_sudoku(s)
avec (s', noeud, pile, bool) = (s, choix_case(s), pilenouv, faux)
tant que (!sudoku_resolu(s')) faire
  si est_vide(liste_possibilités(noeud)) && !bool alors
  (s', nv_possibilités(noeud), pile, bool)
  fsi
  (s', nv_valeur(noeud), pile, bool)
  si est_correct(noeud) && !sudoku_resolu(s') alors
     (s', choix_case(s'), adjp(pile, noeud), faux)
    sinon
    si est_vide(liste_possibilités(noeud) alors
      (s', changer_valeur(noeud, 0), pile, vrai)
      (s', sommet(pile), retirer_sommet(pile), vrai)
    fsi
  fsi
ftantque
```

3.3 Explication des fonctions

résoudre_sudoku : Permet de résoudre le sudoku donné en paramètre

choix_case : Choisi le prochain noeud du sudoku, afin de continuer la résolution

sudoku_résolu : Vérifie si le sudoku est juste ou faux

liste_possibilités : Permet d'obtenir la liste des valeurs possibles pour un noeud

nv_possibilités : Donne une nouvelle valeur au noeud, basé sur la liste des possibilités

est_correct : Vérifie si un noeud n'a pas la même valeur qu'un des noeuds adjacents

changer_valeur : change la valeur du noeud avec la valeur donnée en paramètre

4 Génération de sudoku

4.1 Principe

4.1.1 Generer sudoku

Cette méthode permet de générer plusieurs sudoku aléatoires à partir d'un seul sudoku. Les modifications possibles sont swapRows, swapColumns, swapDiagonalOne et swapDiagonalTwo. En faisant des combinaisons de ces modifications on arrive à générer des sudoku corrects en utilisant la méthode nextInt de la classe Random, ce qui nous permet d'avoir des différentes possibilités de combinaison. A la fin des modifications, on applique la méthode deleteElements pour cacher certaines cases.

4.1.2 Cacher cases

Le principe de cette méthode est de cacher des cases du sudoku afin de pouvoir le résoudre plus tard. Au début on réinitialise les possibilités de chaque nud, au cas où des modifications comme swapRows, swapColumns, swapDiagonalOne, swapDiagonalTwo ont été appliquées sur le sudoku initial. Cela est nécessaire car sinon on ne peut pas assurer que la possibilité de mettre une chiffre dans une case du sudoku soit unique. La prochaine étape consiste à parcourir la grille (sudoku) et mettre les positions s[i][j] dans une liste d'entiers. Puis, avec l'aide de la méthode shuffle on mélange la liste. Dans une boucle qui tient jusqu'à ce que la liste ne soit pas vide et jusquà ce que l'intégrateur ne dépasse pas la taille de la liste, on regarde chaque fois le premier élément de la liste. Est-ce que la valeur du nud de cette position a une possibilité de 1 pour cette case ? Si oui, on peut enlever cet élément, donc in le supprime de la liste et on met sa valeur a 0 dans le sudoku. Mais en appliquant ce changement sur le sudoku, les possibilités des nuds adjacents ne sont plus les mêmes. Il faut donc le changer à l'aide de la méthode setPossibilities. Finalement, si on ne peut pas supprimer un élément, on passe au suivant de la liste.

4.2 Pseudo-code

4.2.1 Generer sudoku

```
generer_sudoku(Sudoku): Sudoku
generer_sudoku(s)

c = random(0,7)
pour i = 0 a c
```

```
  choice = random(0,4)
  si choice == 0
    alors echanger_lignes(s)
  sinon si choice == 1
    alors echanger_colonnes(s)
  sinon si choice == 2
    alors echanger_diagonale_une(s)
  sinon
    echanger_diagonale_deux(s)
  fsi
fpour
cacher_cases(s)
```

4.2.2 Cacher cases

```
cacher_cases(Sudoku) = Sudoku
cacher_cases(s)
l = listenouv()
/* REMPLIR LA LISTE AVEC LES POSITIONS DES ELEMENTS DU s */
pour i = 0 a taille(s)*taille(s)
  pour j = 0 a taille(s)*taille(s)
    pos = i*taille(s)*taille(s) + j
    adjl(l, pos)
  fpour
fpour

mélanger(l)
k = 0
ttq !est_vide(l) && k < taille(l)
rep
  pos = tete(l)
col = pos%(taille(s)*taille(s))
  ligne = pos/(taille(s)*taille(s))

  n = noeud(s, ligne, col)
  si taille(liste_possibilites(n) == 1) || est_vide(liste_possibilites(n))
    alors
      sup(l,k)
      changer_noeud(s, ligne, col, 0)
      tab = noeuds_adjacents(n)
      pour z = 0 a taille(tab)
        nouvelles_possibilites(s)
      fpour
  sinon
    k++
  fsi
frep
```

5 Phase de tests

Afin de tester notre programme, nous avons créé une méthode réalisant un sudoku vide, c'est à dire ne contenant aucune valeur dedans, puis nous résolvons ce sudoku afin d'obtenir un sudoku complet qui satisfait les besoins de génération du sudoku. Ensuite nous affichons le sudoku ainsi résolu, puis générons un nouveau sudoku, puis nous l'affichons à nouveau. Nous résolvons enfin le sudoku ainsi générer et nous l'affichons également.

Tous les sudokus sont ainsi résolvable afin de satisfaire les préconditions de résolution. Nous avons exécuter 20 fois le programme sur le serveur Turing afin d'obtenir une moyenne du temps d'exécution, qu'il soit réel, utilisateur ou bien encore système. Voici un tableau de ces différentes valeurs.

real	user	sys
0.395	0.380	0.1
0.435	0.450	0.06
0.483	0.520	0.05
0.467	0.480	0.06
0.456	0.480	0.06
0.395	0.410	0.06
0.376	0.380	0.08
1.083	1.270	0.07
3.345	3.510	0.09
0.554	0.630	0.07
0.425	0.450	0.05
1.138	1.320	0.07
0.390	0.410	0.06
0.348	0.390	0.03
0.543	0.600	0.12
0.367	0.390	0.05
0.444	0.490	0.04
0.469	0.450	0.090
0.447	0.450	0.07
0.433	0.460	0.06

6 Conclusion

Pour conclure, l'algorithme est fonctionnel et permet bien de résoudre les sudokus de taille 4x4 ainsi que ceux dont la taille est de 9x9. Cependant les sudokus de taille 16x16 ne semble pas résolvable dans l'état. Il peut en résoudre qui ne comporte que des valeurs vides, certes, mais pour en résoudre un, soit le temps pris lors de l'exécution est très long, soit il est coincé dans une boucle infinie et n'arrive pas à effectuer les bons choix. Mis à part ceci il n'y a aucun problème lors de la résolution de sudoku de taille normal, à savoir les 9x9, et l'algorithme arrive également à résoudre les sudoku dits diabolique.

Si l'algorithme de retour-arrière permet de résoudre les sudoku, il est légitime de se demander quel autre type de jeux de logique et de réflexion il permet également de résoudre.