8INF846 - Intelligence Artificielle

UQAC

Université du Québec
à Chicoutimi

Travail Pratique n°2

Création CSP pour le jeu du Sudoku

Groupe 1 :

Mathias DURAND (DURM13099909)

Sophie GERMAIN (GERS09559909)

Pierre LACLAVERIE (LACP03119904)

Barnabé PORTIER (PORB26070006)

Remis pour évaluation le 07 mars 2022

I. Contextualisation

A. Rappel du problème

Le sujet consiste en le développement d'un programme capable de résoudre un sudoku généré aléatoirement ou fourni par l'utilisateur.
L'objectif de ce programme est d'utiliser deux algorithmes de résolution de Problèmes de Satisfaction de Contraintes (CSP) afin de répondre au sujet.
Il est important de noter que nous nous sommes tenus à la définition stricte d'un sudoku : table qui n'a qu'une seule et unique solution. Ainsi, toute table fournie en entrée pouvant avoir plusieurs solutions sera considérée comme insolvable avec AC3, Backtracking renvoyant la première solution trouvée.

B. Exécution de notre programme

Voici les caractéristiques d'exécution de notre programme :

- Version de Python > 3.8
- Fichier Python à exécuter : Main.py avec ou sans arguments (voir "Paramétrages optionnels")
- Utilisation de la bibliothèque argparse : pip install argparse (normalement installée par défaut dans les versions > 3.2 de Python)
- Arrêt du programme : après résolution du sudoku, puis input de l'utilisateur ou Ctrl+C

Paramétrages optionnels :

- Paramétrage optionnel : --backtracking permet d'utiliser l'algorithme de backtracking, plutôt que l'algorithme d'AC3, utilisé par défaut
- Paramétrage optionnel : --indices <entier> afin de définir le nombre de cases pré-remplies lors de la génération aléatoire d'une table de sudoku. Par défaut vaut 50.
- Paramétrage optionnel : --file <chemin/vers/le/fichier.txt> afin de fournir un sudoku personnalisé au programme (voir la partie "Exemple de Sudoku personnalisé")
- Paramètre -h pour revoir ces paramètres optionnels dans la console

II. Modélisation de la solution

Nous avons choisi comme algorithmes le ***Backtracking*** et ***AC3***.
Si dans le cas d'utilisation de l'AC3 avec le nombre de cases **pré-remplies** par défaut, la complétion de la table est **instantanée**, l'algorithme de backtracking est un peu plus long, mettant entre **1 et 15 secondes** à trouver une solution. En augmentant le nombre d'indices à 50 ce temps diminue à 4 secondes.
Nous avons modélisé la solution à l'aide des classes Python suivantes.

Sudoku :

Cette classe permet de représenter un **sudoku**. Elle définit donc un *board*, rempli **aléatoirement** ou **fourni** par l'utilisateur. La classe comprend également un **CSP**. L'affichage du Sudoku se fait 2 fois (avant et après résolution) dans la console tel qu'en bas de page.
Il est possible de fournir au programme un sudoku personnalisé. Pour cela il faut le décrire dans un fichier .txt que l'on fournit ensuite en entrée en ligne de commandes.
Le fichier .txt doit respecter la structure suivante :

```
790000003
000403000
003090586
370941852
085000070
920750000
630504190
007200400
000006300
```

avec les cases vides représentées par des 0. Vous pouvez utiliser le fichier "test.txt" à la racine du TP pour tester la fonctionnalité. Attention le backtracking est assez long sur celui-ci.

Exemple d'affichage de sudoku →

```
[AC3] AC3 en cours...

[Sudoku] SOLUTION TROUVÉE !

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

[Sudoku] SOLUTION TROUVÉE !
```

CSP :

Cette classe permet de définir les différentes contraintes de notre système. Ici les **contraintes** concernent les cases de notre table de sudoku qui sont dans la même ligne, même colonne ou même groupe de 3x3 cases (appelé *boîte*). Cette classe définit également les **domaines** (ensemble des valeurs acceptables) pour chacune des **variables** (ici les *cases*).

Backtracking :

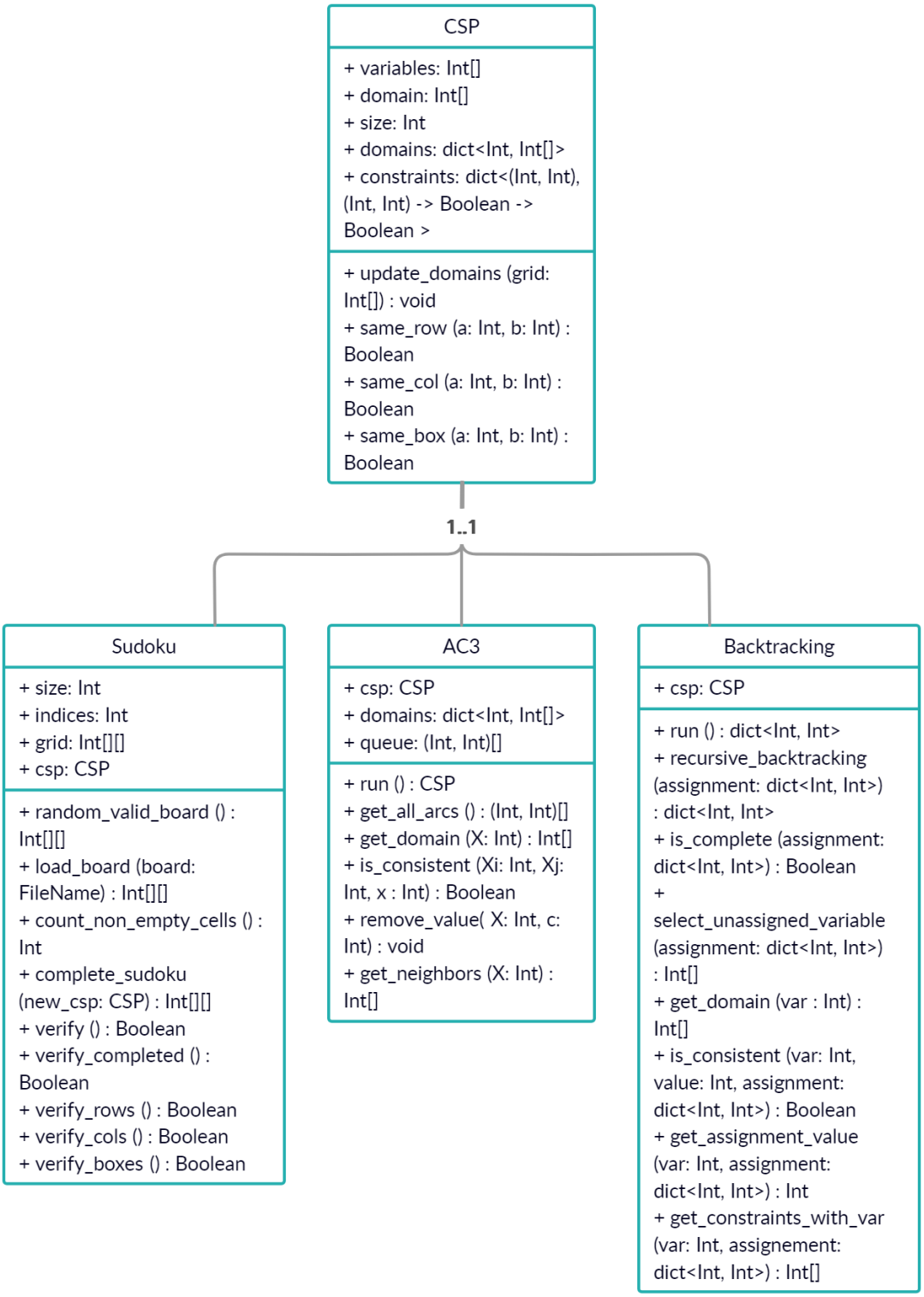
Développé à partir de l'**algorithme en pseudo-code** du cours, il permet de trouver une solution à notre problème avec contraintes en **revenant en arrière** dans le graphe des possibilités lorsqu'il est bloqué par une variable dont **aucune des valeurs** ne respecte les contraintes qui lui sont imposées. Le "backtracking" est la raison pour laquelle cet algorithme est **plus lent** que AC3.

AC3 :

Développé à partir de l'**algorithme en pseudo-code** du cours, il permet de trouver une solution à notre problème avec contraintes en utilisant la **propagation** des informations sur les valeurs prises par les variables du problème et en faisant des **vérifications locales** sur les arcs (représentant les deux cases en relation) sur la consistance d'une relation.
L'algorithme est bien plus rapide que le Backtracking dans la plupart des cas.

Relations :

Le diagramme de classes UML suivant fait état des relations liant les différentes classes de notre modèle :



III. Code et explications

Il est à noter que les captures d'écran ont été légèrement modifiées par rapport au code afin d'en simplifier la lecture. Des fonctions ont été renommées pour plus de cohérence et des commentaires ainsi que d'autres détails sont présents dans le code source mais pas ici, ces captures n'ayant pour vocation qu'à illustrer les propos tenus.

A. Définition du sudoku

La classe dispose de 10 fonctions dont 7 destinées à une utilisation interne uniquement. À la création d'un objet Sudoku, on lui fournit la **taille de la table** de jeu ainsi que le nombre de **cases pré-remplies** qu'on souhaite. On peut lui fournir une table déjà existante pour qu'elle soit **chargée** avec `load_board`. Si aucune table n'est précisée en entrée on en génère une **aléatoirement** (voir Génération aléatoire d'une table de sudoku). On crée ensuite un **CSP** (voir Collection de contraintes binaires) pour cette table, qu'on met à jour avec les cases déjà remplies.
Une fois **l'algorithme de calcul** terminé (*Backtracking* ou *AC3* dans notre cas), on fournit un **nouveau CSP** à l'objet dans la fonction `complete_sudoku` et on met à jour notre table avec les valeurs trouvées par l'algorithme. Si l'algorithme a trouvé une solution, toutes les **cases seront remplies correctement**, suivant les contraintes du CSP.
On peut alors faire une vérification avec la fonction `verify`, qui s'assure qu'il n'y a pas de **doublon** de chiffre dans les lignes, colonnes et "boîtes" de 3x3.

```python
class Sudoku:

  def __init__(self, size, board, indices):
    self.size = size
    self.indices = indices
    self.grid = self.random_valid_board() if board is None else self.load_board(board)
    self.csp = CSP(size)
    self.csp.update_domains(self.grid)

  def random_valid_board(self):
    [...]

  def load_board(self, board):
      with open(board, "r") as f:
          lines = f.readlines()
      return [[int(x) for x in line.strip()] for line in lines]

  def count_non_empty_cells(self):
      return sum(1 for row in self.grid for el in row if el != 0)

  def complete_sudoku(self, new_csp):
      for var in new_csp.variables:
          if len(new_csp.domains[var]) == 1:
              self.grid[var // self.size][var % self.size] = new_csp.domains[var][0]
              self.csp.domains[var] = [0]
      return self.grid

  def verify(self):
      return self.verify_completed() and self.verify_rows() and self.verify_cols() and self.verify_boxes()

  def verify_completed(self):
      return self.count_non_empty_cells() == self.size ** 2

  def verify_rows(self):
      [...]

  def verify_boxes(self):
      [...]

  def verify_cols(self):
      [...]

```

B. Génération aléatoire d'une table de sudoku

Il est primordial de noter que notre algorithme de création aléatoire de sudokus fonctionne mais il est possible que nos algorithmes ne parviennent pas à le résoudre. Une telle situation survient lorsque l'algorithme ne dispose, par exemple, **pas d'assez** de cases pré-remplies au lancement, ou que les cases vides restantes donnent lieu à **plusieurs solutions** possibles (voir Contextualisation).

Notre algorithme de génération aléatoire fonctionne en appliquant des transformations sur le **sudoku valide "de base"** suivant.

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

On charge donc cette table de sudoku ligne 2, puis on la fait "tourner" (transposée de la table) **aléatoirement entre 0 et 3 fois**.
Ensuite, pour **5 à 15 fois**, on **échange les valeurs** de certaines cases selon leur chiffre. Par exemple, tous les 2 deviennent des 9 pendant que tous les 9 deviennent des 2. Avec cette méthode, la table résultante reste un **sudoku valide**.
Nous avons calculé qu'il est possible d'avoir jusqu'à environ **10 000** tables de sudoku différentes avec cette technique.

Note : le seed du random est initialisé lors de la création de la classe Sudoku à la valeur de l'heure actuelle en millisecondes.

```python
def random_valid_board(self):
        board = self.load_board("BASE_SUDOKU.txt")
        for i in range(0, random.randint(0, 3)):
            board = [*zip(*board)]
        available_variables = [i for i in range(1, 9)]
        for m in range(random.randint(5, 15)):
            t_num1 = available_variables[random.randint(0, len(available_variables) - 1)]
            t_num2 = available_variables[random.randint(0, len(available_variables) - 1)]
            while t_num2 == t_num1:
                t_num2 = available_variables[random.randint(0, len(available_variables) - 1)]

            for k in range(len(board)):
                board[k] = [i if i != t_num1 and i != t_num2 else t_num1 if i == t_num2 else t_num2 for i in board[k]]

        for _ in range(self.size**2 - self.indices):
            r = random.randint(0, self.size**2 - 1)
            while board[r // self.size][r % self.size] == 0:
                r = random.randint(0, self.size**2 - 1)
            board[r // self.size][r % self.size] = 0

        return board
```

C. Collection de contraintes binaires

Le **Problème à Satisfaction de Contraintes** est modélisé avec une **liste de variables**, un **dictionnaire des variables et leurs domaines** et est généré à l'aide de la taille du sudoku passé en paramètre. Notre CSP est ici construit pour un problème lié à la résolution des sudokus, d'où les **contraintes symétriques** entre deux cases `a` et `b` d'une même ligne, même colonne ou même "boîte" de 3x3 (pas de doublon de chiffres).
La fonction `update_domains` est utilisée par la classe `Sudoku` pour mettre à jour les domaines des variables qui **possèdent déjà** une valeur à la création de la table (les "indices")

```python
class CSP:
  def __init__(self, size):
    self.variables = [i for i in range(size ** 2)]
    self.domain = [i for i in range(1, size + 1)]
    self.size = size
    self.domains = {}
    for var in self.variables:
      self.domains[var] = self.domain

    constraints = {}
    for a in range(size**2):
      for b in range(size**2):
        if (a != b and (self.same_row(a, b) or
           self.same_col(a, b) or self.same_box(a, b))):
          constraints[(a, b)] = lambda x, y: x != y
          constraints[(b, a)] = lambda x, y: x != y
    self.constraints = constraints

  def update_domains(self, grid):
    for i in range(self.size):
      for j in range(self.size):
        if grid[i][j] != 0:
          self.domains[i * self.size + j] = [grid[i][j]]
  [...]
```

D. Algorithme de backtracking

L'algorithme implémente la logique du **pseudo-code** du cours. Notre variable de **récursion**, `assignment`, est un dictionnaire des variables traitées et de la valeur choisie pour celle-ci. Une fois la valeur d'une variable choisie, on **propage l'information** sur la suite du trajet, et on cherche à déterminer si les variables "voisines" (qui ont un arc en commun avec la variable précédente traitée) sont **consistantes** avec le choix effectué. Si la consistance n'est pas respectée, on rebrousse chemin et on **supprime** la valeur choisie pour la variable précédente. Si elle est respectée, on ajoute une entrée dans notre dictionnaire `{variable:valeur}`.

```python
class Backtracking:
    def __init__(self, csp):
        self.csp = csp

    def Backtracking(self):
        r = self.RecursiveBacktracking({}, self.csp)
        return {i:[r[i]] for i in r}

    def RecursiveBacktracking(self, assignment, csp):
        if self.isComplete(assignment, csp):
            return assignment
        var = self.selectUnassignedVariable(assignment, csp)
        for value in self.orderDomainValues(var, assignment, csp):
            if self.isConsistent(var, value, assignment, csp):
                assignment[var] = value
                result = self.RecursiveBacktracking(assignment, csp)
                if result is not None:
                    return result
                assignment.pop(var)
        return None

    def isComplete(self, assignment, csp):
        return len(assignment) == len(csp.variables)

    def selectUnassignedVariable(self, assignment, csp):
        return list(csp.variables - assignment.keys())[0]

    def orderDomainValues(self, var, assignment, csp):
        return csp.domains[var]

    def isConsistent(self, var, value, assignment, csp):
        couples = self.getConstraintsWithVar(var, assignment, csp)
        for couple in couples:
            contrainte = csp.constraints[couple]
            [...]
            if not contrainte(value, nv):
                    return False
            [...]
        return True

    def getAssignementValue(self, var, assignment):
        return assignment[var]

    def getConstraintsWithVar(self, var, assignement, csp):
        return [i for i in csp.constraints if i[0] == var and i[1] in assignement
                or i[1] == var and i[0] in assignement]

```

E. Algorithme AC-3

L'algorithme implémente la logique du **pseudo-code** du cours. On commence par récupérer tous les arcs, qu'on place dans une queue, qu'on videra au fur et à mesure.
Pour chaque arc, on commence par supprimer les valeurs inconsistantes du domaine de la première variable de l'arc. Si on supprime certaines valeurs, on ajoute dans la liste des arcs à traiter les arcs dont le second membre est la variable qu'on étudiait à l'instant. On renvoie les domaines résultants lorsque la queue des arcs à traiter est vide, donc que tous les domaines ont été réduits au maximum.

```python
class AC3:
    def __init__(self, csp):
        self.csp = csp
        self.domains = csp.domains.copy()
        self.queue = self.get_all_arcs()

    def get_all_arcs(self):
        tmp = set([])
        for constraint in self.csp.constraints:
            i = constraint[0]
            j = constraint[1]
            if i != j:
                tmp.add((i, j))
                tmp.add((j, i))
        a = [i for i in tmp]
        return a

    def getDomain(self, X):
        return self.domains[X]

    def isConsistent(self, Xi, Xj, x):
        contrainte = self.csp.constraints[(Xi, Xj)]
        for y in self.getDomain(Xj):
            if contrainte(y, x):
                return True
        return False

    def removeValue(self, X, c):
        self.domains.get(X).remove(c)

    def getNeighbors(self, X):
        tmp = [i for i in self.csp.constraints if i[0] == X or i[1] == X]
        return [i[0] if i[0] != X else i[1] for i in tmp]

    def AC3(self):

        def removeInconsistentValues(Xi, Xj):
            removed = False
            D = self.getDomain(Xi)
            for c in D:
                if not self.isConsistent(Xi, Xj, c):
                    self.removeValue(Xi, c)
                    removed = True
            return removed

        print("[AC3] AC3 en cours...\n")
        while len(self.queue) > 0:
            p = self.queue.pop()
            Xi = p[0]
            Xj = p[1]
            if removeInconsistentValues(Xi, Xj):
                D = self.getDomain(Xi)
                if (len(D) < 1):
                    break
                else:
                    for Xk in self.getNeighbors(Xi):
                        self.queue.append((Xk, Xi))

        self.csp.domains = self.domains.copy()

        return self.csp
```

F. Programme principal

Le programme principal récupère les arguments en ligne de commande en entrée puis les traite (indices et fichier souhaité).
Un sudoku est alors créé et affiché. L'algorithme souhaité est lancé et les résultats sont traités par l'objet `sudoku`. On affiche alors un message et le sudoku final.

Note : le programme ne se ferme pas tout seul pour des raisons pratiques d'utilisation avec certaines consoles windows, qui ne laissent pas le temps à la personne de lire le texte à l'issue du traitement avant qu'elles se ferment.

```python
def main(args):
    sudoku = Sudoku(size=9, args.file, args.indices)
    print("\n------SUDOKU INITIAL------\n")
    print(sudoku)
    print("\n--------------------------\n")

    if args.backtracking:
        backtracking = Backtracking(sudoku.csp)
        r = backtracking.Backtracking()
        new_csp = backtracking.csp
        new_csp.domains = r
    else:
        ac3 = AC3(sudoku.csp)
        new_csp = ac3.AC3()

    new_grid = sudoku.complete_sudoku(new_csp)

    final_message = "SOLUTION TROUVÉE !" if sudoku.verify()
        else "SUDOKU INSOLVABLE"
    print(f"[Sudoku] {final_message}\n")
    print(sudoku)
    print(f"[Sudoku] {final_message}")
```