Travail n°2

Création CSP pour le jeu du Sudoku

Groupe :
CAVANI Nicolas (CAVN06029909)
GRILLON Théo (GRIT04010107)
OECHSLIN Killian (OECK17039908)

I - Instructions pour le lancement de l'application

/!\ La seule chose à faire attention pour que l'application fonctionne correctement est de la lancer (fichier src/main.py) à partir du dossier ***src*** de notre arborescence de projet.

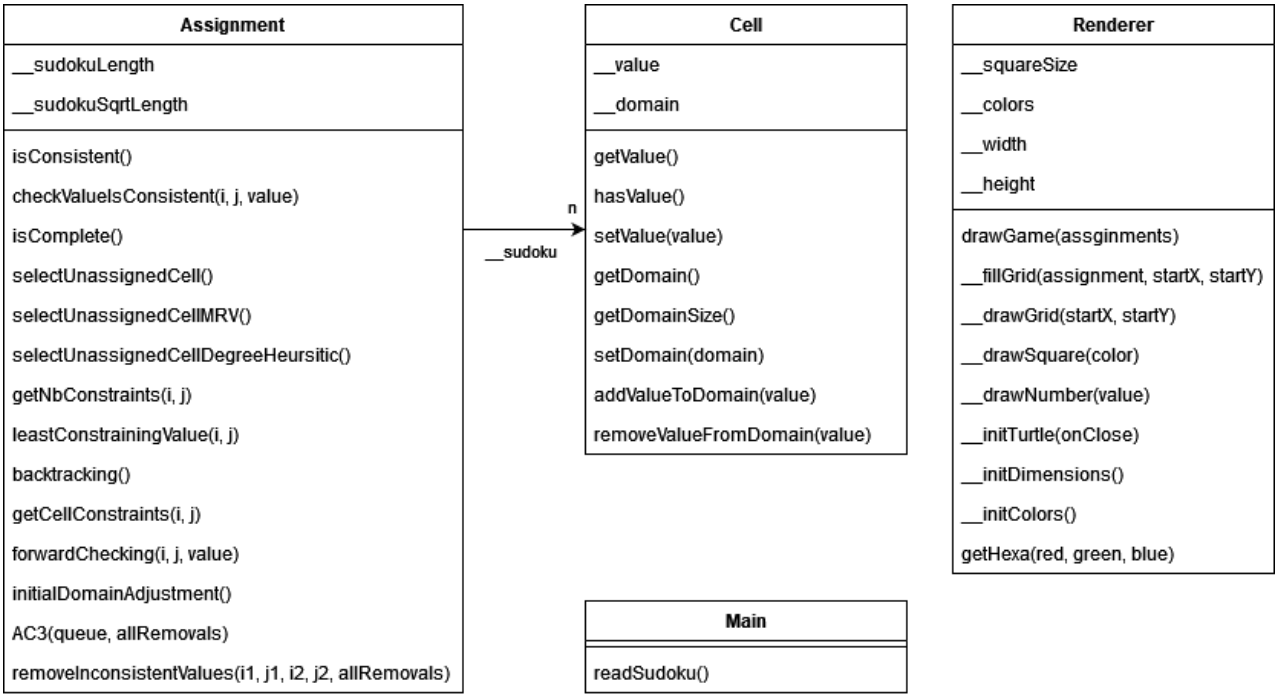
Il est possible d'éditer le sudoku à résoudre en modifiant le fichier "sudoku.txt" présent dans le dossier "resources". Il est nécessaire que les sudokus fournis soient de taille carrée. Le format à respecter est le suivant :

- La 1ère ligne doit indiquer la taille d'un côté du sudoku (Exemple : 9 pour un 9x9)
- Il doit y avoir ensuite N lignes (N étant la taille du côté) les unes à la suite des autres indiquant les numéros de départ du sudoku. Si la case est remplie par défaut, il faut indiquer le chiffre, et sinon il faut mettre un point (".") pour indiquer une case vide. Enfin, chaque chiffre/point doit être séparé par un espace.

Note : Nous avons placé quelques sudokus d'exemple dans le dossier "resources". Pour les utiliser il suffit de copier leur contenu dans le fichier "resources/sudoku.txt"

II - Explication de la modélisation

Avant de passer aux explications détaillées, nous pouvons déjà observer le diagramme UML de notre application :



Tout d'abord, par rapport à la modélisation d'un CSP nous avons créé les classes ***Assignment*** et ***Cell***.

La classe ***Cell*** modélise une variable de notre problème, qui est ici une case de la grille de sudoku, et elle possède pour cela un attribut __*value*, représentant la valeur actuellement assignée à la case, et un attribut __*domain* qui représente les valeurs légales pour cette case. Au niveau des méthodes, ce sont surtout des getters ou des setters qui permettent d'interagir avec les attributs de la classe. Nous pouvons parler un peu plus en détails de quelques méthodes : *hasValue* renvoie un booléen indiquant si la case possède une valeur non nulle ; *addValueToDomain* ajoute la valeur passée en paramètre au domaine de la case, si cette valeur n'est pas déjà dedans ; *removeValueFromDomain* retire la valeur passée en paramètre du domaine, et renvoie un booléen indiquant si la valeur était effectivement présente dans le domaine, ce qui peut être une information utile dans certains algorithmes.

Ensuite, la classe ***Assignment*** représente un état du CSP dans lequel une valeur a été associée ou non aux variables du problème. Cette classe possède 3 attributs : __*sudoku* qui est un tableau 2D de *Cell*, possédant les mêmes dimensions que le sudoku à résoudre ; __*sudokuLength* est un entier représentant la taille d'un côté du sudoku, et __*sudokuSqrtLength* représente la racine carrée de la taille du côté (nous avons ajouté cet attribut pour des raisons de performances car cette valeur est utilisée assez fréquemment et il est assez coûteux de calculer une racine carrée). Pour ce qui est des méthodes, nous retrouvons : *isConsistent* qui vérifie que l'assignement global du sudoku ne possède pas de conflits ; *checkValueIsConsistent* qui prend en paramètre les coordonnées (i, j) d'une case et une valeur, et qui vérifie simplement si le fait de mettre la valeur donnée dans la case indiquée introduit des conflits ou non (plus rapide que de vérifier tout le sudoku) ; *isComplete* indique si toutes les variables (cases) du CSP ont une valeur ; *selectUnassignedCell* parcours le sudoku et renvoie la première case n'ayant pas de valeur (sélection simpliste) ; *selectUnassignedCellMRV* renvoie une liste de cases (dans le cas où plusieurs cases ont la même valeur d'heuristique) choisies grâce à l'heuristique *Minimum Remaining Values* (MRV) ; *selectUnassignedCellDegreeHeuristic* renvoie une liste de cases (dans le cas où plusieurs cases ont la même valeur d'heuristique) choisies grâce à l'heuristique *Degree Heuristic* ; *getNbConstraints* prend en paramètre les coordonnées (i, j) d'une case et renvoie le nombre de contraintes qu'elle applique sur les cases restantes (utile pour Degree Heuristic) ; *leastConstrainingValue* prend en paramètre les coordonnées (i, j) d'une case et renvoie son domaine, trié par ordre de préférence grâce à l'heuristique *Least Constraining Value* ; *backtracking* est la méthode qui permet de rechercher récursivement la solution du sudoku (détails du fonctionnement expliqué dans la partie suivante) ; *getCellConstraints* prend en paramètre les coordonnées d'une case et renvoie un tableau de coordonnées indiquant les cases qui sont liées par des contraintes à celle passée en paramètre, c'est à dire les cases situées sur la même ligne, sur la même colonne et dans le même carré (nous les appellerons par la suite "cases voisines") ; *forwardChecking* prend en paramètre les coordonnées d'une case et une valeur, et elle va vérifier si en mettant la valeur dans la case spécifiée, et en mettant à jour les domaines des cases voisines, il n'y a pas de case encore non assignée qui ne possède plus aucune valeur légale ; *initialDomainAdjustment* est une méthode qui va mettre à jour les domaines des cases au début du problème, après avoir lu le sudoku d'entrée, afin de directement éliminer les valeurs illégales ; *AC3* correspond à l'implémentation de l'algorithme AC-3 et elle prend en paramètre une queue d'arcs (dans notre problème, un arc est une paire de coordonnées de 2 cases, sous la forme [(i1, j1), (i2, j2)]) ainsi qu'un tableau qui sert à retenir toutes les suppressions qui sont faites dans les domaines des cases pour pouvoir les remettre dans le cas où la valeur choisie pour une case ne conviendrait pas et où il serait alors nécessaire de remonter dans le backtracking ; et enfin *removeInconsistentValues* prend en paramètre un arc de contrainte (c'est-à-dire une paire [(i1, j1), (i2, j2)]) et un tableau de suppression (même rôle

qu'évoqué précédemment) et va enlever du domaine de la case (i1, j1) les valeurs pour lesquelles il n'y a pas de valeurs légales possibles pour la case (i2, j2).

Pour lancer le programme, nous avons une classe ***Main*** qui s'occupe de lire le sudoku en entrée, via la méthode *readSudoku*, puis de créer l'assignement initial, de lancer le backtracking, et d'afficher le résultat. Il est à noter que la méthode *readSudoku* permet de lire n'importe quelle taille de sudoku (tant que ce dernier est carré), à condition de bien respecter le format indiqué en première page. De plus, nous avons également implémenter un système de "benchmarking" permettant de lancer plusieurs fois à la suite la résolution du sudoku en mesurant le temps à chaque fois, puis d'afficher le temps moyen de résolution à la fin (il suffit de changer la variable "*nbRun*" dans main.py).

Finalement, nous avons créé une classe ***Renderer*** qui permet de gérer l'affichage graphique du sudoku. Dans cet affichage, nous pouvons voir le sudoku tel qu'il est donné en entrée (non rempli) sur la gauche de la fenêtre, ainsi que le sudoku résolu sur la droite. Les couleurs choisies pour afficher le sudoku sont calculées automatiquement en fonction de la taille du sudoku afin de sélectionner des couleurs autour du cercle chromatique les plus éloignées possibles, c'est-à-dire, pour que la nuance entre chaque couleur soit la plus grande possible. On affiche également la valeur numérique de la case. Le reste des attributs et méthodes de cette classe ne sont pas vraiment pertinents à détailler ici car, comme indiqué précédemment, ils s'occupent simplement de l'affichage et n'ont pas de lien avec la logique du CSP.

III - Explication de l'implémentation

a) Explication du backtracking

Comme indiqué précédemment, pour l'implémentation de ce CSP nous avons créé 2 classes principales : *Assignment* et *Cell*. Au lancement, le programme lit le sudoku fourni en entrée puis crée l'assignement initial. A partir de celui-ci, il lance la recherche récursive via la méthode *backtracking*. Le déroulement de cette recherche est le suivant :

- Nous regardons en premier si l'assignement est complet, ce qui signifie que la solution a été trouvée (car nous nous assurons que les contraintes sont respectées à chaque fois). Si c'est le cas, nous renvoyons True.
- Si l'assignement n'est pas complet, nous sélectionnons la prochaine case à visiter (que nous appellerons C) en utilisant d'abord MRV, qui peut renvoyer une liste de cases ayant la même valeur d'heuristique, puis en utilisant Degree Heuristic sur la liste de cases précédentes, et nous récupérons enfin la première case renvoyée par Degree Heuristic. Nous récupérons ensuite le domaine de la case choisie (que nous appellerons D), trié par ordre de préférence selon l'heuristique *Least Constraining Value*.
- Nous itérons alors sur chaque valeur du domaine D et nous vérifions dans un premier temps si placer la valeur actuelle dans la case C donne un assignement consistant ou non (via la méthode *checkValueIsConsistent*). Si l'assignement n'est pas consistant, nous passons à la prochaine valeur du domaine D.
- En revanche, si l'assignement est consistant nous plaçons la valeur dans la case C puis nous construisons la queue d'arcs pour la méthode AC3, et nous lançons cette dernière. Cette méthode va permettre de mettre à jour les domaines des cases en propageant les contraintes et si jamais une case n'a plus de valeur légale, alors nous renvoyons False et nous n'allons pas plus loin dans le backtracking.
- Dans le cas où AC3 renvoie False, nous enlevons la valeur de la case C et nous remettons les domaines des cases tels qu'ils étaient avant de lancer la vérification (car cette dernière a pu toucher aux domaines), puis nous passons à la prochaine valeur possible pour la case C.

- Dans le cas où AC3 renvoie True, nous appelons de nouveau la méthode backtracking, qui va permettre de continuer la recherche sur le nouvel assignement que nous venons de faire. Nous vérifions ensuite la valeur de retour du nouvel appel afin de savoir si la solution a été trouvée plus bas dans ce backtracking ou pas. Si la solution a été trouvée, nous renvoyons True, sinon nous enlevons la valeur de la case C, nous remettons les domaines des cases tels qu'ils étaient avant d'assigner cette valeur (comme pour AC3) puis nous passons à la prochaine valeur du domaine D.
- Finalement, si nous arrivons à la fin du domaine D et qu'aucune solution n'a été trouvée, nous renvoyons False.

b) Explication de l'heuristique MRV

La fonction *selectUnassignedCellMRV* implémente la méthode Minimum Remaining Values. Dans cette dernière, on parcourt le sudoku à la recherche des cases encore vides, et parmi elles des cases avec le moins de valeurs possibles restantes. Si plusieurs cases possèdent le même nombre (minimum) de valeurs restantes, elles sont ajoutées à une liste. C'est cette liste que la fonction retourne de manière à ensuite appliquer la méthode Degree Heuristic.

c) Explication de l'heuristique Degree Heuristic

Pour cette heuristique, nous cherchons la case qui possède le plus de contraintes sur les cases restantes. On utilise pour cela une méthode de calcul des contraintes, qui compte pour chaque case encore non assignée le nombre de cases voisines qui n'ont pas de valeur (il existe donc une contrainte entre les 2 cases considérées). Une fois la case la plus contraignante trouvée, nous la renvoyons. Il est à noter que si plusieurs cases possèdent le même nombre de contraintes, nous renvoyons la liste de ces cases.

Cette méthode demande potentiellement beaucoup de backtracking dans le cadre d'un sudoku, car commence souvent avec un champ de possibilité très large.

d) Explication de l'heuristique Least Constraining Value

Pour cette heuristique, nous prenons en paramètre les coordonnées (i, j) de la case C dont on souhaite récupérer le domaine trié.

Tout d'abord, nous récupérons les valeurs légales pour cette case ainsi que ses cases voisines (celles situées sur la même ligne, même colonne ou sur le même carré), qui sont liées par des contraintes à la case C.

Ensuite, pour chaque valeur légale V nous regardons les cases voisines qui n'ont pas de valeur (c'est-à-dire celles qu'il reste à assigner) et nous vérifions si V est présente dans le domaine de la case voisine. Si la valeur est présente, cela signifie qu'en assignant V à la case C cela réduira le domaine de la case voisine. Dans ce cas, nous incrémentons un compteur indiquant le nombre de cases voisines dont le domaine est réduit en assignant la valeur V.

Enfin, après avoir parcouru toutes les valeurs possibles pour la case C, nous avons des paires [valeur V ; nombre de cases voisines impactées], et nous trions ces paires par ordre croissant de nombre de cases voisines impactées. Nous renvoyons alors la liste des valeurs possibles pour la case C en respectant l'ordre indiqué précédemment. Ainsi, dans le backtracking les valeurs explorées en premier sont celles qui réduisent le moins les valeurs possibles des cases voisines.

e) Explication de la vérification par la méthode AC3

La méthode AC3 prend en paramètre une queue d'arcs de contraintes (dans notre cas il s'agit d'une paire des coordonnées de 2 cases, sous la forme [(i1, j1) , (i2, j2)]), ainsi qu'une liste permettant de garder en mémoire les suppressions qui sont faites dans les domaines des

cases. Cette dernière est surtout utile lorsqu'aucune solution n'a été trouvée et qu'on remonte dans le backtracking, ainsi nous ne détaillerons pas plus son utilité.

Nous itérons alors tant que la queue n'est pas vide, et à chaque itération nous prenons le premier arc de la queue. Grâce à cet arc, nous enlevons les valeurs inconsistantes (via la méthode *removeInconsistentValues*) : pour cela, nous parcourons le domaine de la case (i1, j1) et nous regardons pour chaque valeur V s'il y a au moins une valeur du domaine de la case (i2, j2) qui permet de satisfaire la contrainte. Concrètement, dans notre cas cela revient à regarder si le domaine de (i2, j2) a une taille supérieure à 1, et si jamais la taille est égale à 1, nous regardons si la seule valeur légale est la même que la valeur V. Si le domaine est de taille 1 et que la valeur est la même que la valeur V, alors nous retirons V du domaine de la case (i1, j1), puis nous continuons d'itérer sur les valeurs.

Si au moins une valeur a été enlevé du domaine de la case (i1, j1), alors pour chaque chaque case voisine C' nous ajoutons dans la queue l'arc [$(i_{C'}, j_{C'})$, (i1, j1)], ce qui permet ainsi de propager les contraintes (contrairement à *forward checking*).

Enfin, si une ou plusieurs valeurs sont enlevées des domaines, nous vérifions à chaque fois qu'il reste des valeurs légales dans le domaine. S'il en reste, nous continuons de propager les contraintes (tant que la queue n'est pas vide), sinon nous renvoyons False, ce qui permet d'indiquer que l'assignement actuel mènera à une inconsistance à un moment donné (une case avec plus aucune valeur légale).