Création de CSP pour le jeu du Sudoku

Introduction

L'objectif de ce devoir est de concevoir un CSP, ou problème à satisfaction de contraintes, modélisant un problème bien connu : le jeu du Sudoku.

Le Sudoku est un jeu inventé en 1979 par Howard Garns. Sa version la plus connue est se compose d'une grille de 9x9 cases divisée en 9 sous-grilles de 3x3 cases. Le but est de remplir la grille en tenant compte des contraintes suivantes :

- Chaque ligne / colonne / sous-grille doit comporter tous les nombres allant de 1 à 9 ;
- Chaque ligne / colonne / sous-grille ne peut comporter qu'un exemplaire de chaque nombre.

Pour mener à bien la conception de ce CSP, deux notions fondamentales vues en cours seront utilisées :

- L'exploration de l'arbre des solutions possibles à l'aide de la méthode de *Backtracking Search* ;
- La propagation de contraintes grâce à l'utilisation de l'algorithme de *Forward Checking*.

Dans un premier temps, nous verrons l'architecture du programme proposé. Il sera ensuite question de présenter la propagation de contraintes *Forward Checking* et l'algorithme d'exploration *Backtracking Search*. Enfin nous verrons les heuristiques implémentées (*Minimum Remaining Values* et *Value with Maximum Constraints*) afin d'accélérer la résolution du problème.

Informations nécessaires à l'exécution du programme

Ce programme de modélisation de CSP a été réalisé en langage C# sous Visual Studio 2015 et nécessite .NET Framework 4.0 au minimum pour fonctionner. Le dossier remis comporte les éléments suivants :

- Le fichier de solution SudokuCSP.sln permettant d'ouvrir le code source sous Visual Studio (Windows) ou Monodevelop (Linux) et de le compiler. La compilation est à réaliser avec une plate-forme x64. Le code source est localisé dans le dossier SudokuCSP.
- Un fichier Run_SudokCSP.bat permettant de lancer l'exécutable SudokuCSP.exe localisé dans "SudokuCSP_Executable".
- Un fichier Open_Sudoku_Folder.bat permettant d'ouvrir le dossier localisé dans "SudokuCSP_Executable\SudokuGrid" et contenant les grilles de Sudoku. Quelques grilles sont déjà présentes (dont le nom commence par Sudoku9 pour les grilles 9x9 et par Sudoku16 pour les grilles 16x16) ainsi que leur solution.

Afin d'ajouter une grille, il suffit de la rajouter dans le dossier "SudokuCSP_Executable\SudokuGrid" sous la forme suivante et de la sauvegarder au format .CSV :

8,0,0,0,0,0,0,0,0
0,0,3,6,0,0,0,0,0
…

1. Organisation du projet

Le projet est décomposé en cinq classes qui vont être ici présentées avant de rentrer dans le détail du fonctionnement du CSP.

Concernant le vocabulaire employé, les termes suivants doivent être explicités :

- Cellule : case du Sudoku, *e.g.* un Sudoku 9x9 en comporte 81 ;
- Région : une des sous-grilles du Sudoku ;
- *Peers* : Les cases appartenant à la même ligne / colonne / sous-grille que la case considérée et contraignant ainsi ses valeurs.
- *Backtrack* : action de remonter d'une étape dans l'arbre de recherche lorsque l'algorithme est bloqué dans la résolution.

1.1. `MainProgram`

La classe `MainProgram` contient le `main` qui sera lancé lors du lancement de l'exécutable Sudoku.exe. Il permet de réaliser le bon déroulement du programme à savoir :

- Demander à l'utilisateur de saisir la taille et le nom de la grille de Sudoku à utiliser ;
- Importer le fichier .CSV associé et initialiser l'objet *Sudoku* associé ;
- Afficher le Sudoku de départ ;
- Lancer l'algorithme de recherche *BacktrackingSearch* ;
- Afficher le Sudoku résolu ou un message d'erreur si le Sudoku n'a pas de solution.

Elle comporte un `while(true)` afin de pouvoir lancer des résolutions de Sudoku en boucle. Il affiche également le temps et le nombre de *backtracks* nécessaires à la résolution du Sudoku.

1.2. `Coordinate`

La classe `Coordinate` est une simple classe utilitaire qui sert à formater les coordonnées des *peers* d'une cellule. Elle permet ainsi de faire référence au *peers* sans avoir une dépendance de pointeur d'objets avec ces derniers (utile pour réaliser une copie de la liste des *peers*).

1.3. `Cell`

L'objet `Cell` représente une des cellules du Sudoku. Elle est dotée de quatre attributs :

- `m_iCellValue`, un `int` qui représente la valeur courante de la cellule. Si la cellule est non-assignée, elle vaut 0.
- `m_liPossibleValues`, une liste d'`int` qui représente l'ensemble des valeurs possibles pour la cellule.
- `m_lcPeers`, une liste de `Coordinate` qui représente la liste des coordonnées des *peers* d'une cellule.
- `m_bAssigned`, un booléen qui permet de savoir si la cellule possède une valeur assignée.

1.4. `Sudoku`

Un `Sudoku` est une `array` carrée de `Cell`. Il possède deux attributs principaux :

- `m_iSudokuSize`, un int qui stocke la taille par défaut du Sudoku. Elle est décidée par l'utilisateur en début de programme et doit permettre de réaliser un Sudoku carré, *e.g.* 9 pour un Sudoku 9x9, 16 pour un 16x16 etc.

- `m_aiSudokuGrid`, une matrice de `Cell` qui constitue l'ensemble des cases du Sudoku.

Cette classe contient également les fonctions de lecture des fichiers .CSV `ReadCSV()` et d'affichage du Sudoku dans la console `PrintSudokuGrid()`. Elle comporte également la fonction `InitSudoku()` qui va trouver pour chaque cellule la liste de ses *peers* (à l'aide des fonctions `ComputeRowAndColumnPeers()` qui donne la liste des cellules de la même ligne et de la même colonne et `ComputeRegionPeers()` qui donne la liste des cellules de la même région n'étant pas sur la même ligne ou colonne) et appliquer les contraintes des éléments initiaux à l'ensemble des cases non-assignées du Sudoku.

1.5. `Solver`

Le `Solver` est une classe statique contenant l'ensemble des méthodes nécessaires à la résolution du Sudoku. Les diverses fonctions vont être détaillées dans les quatre parties suivantes.

2. Exploration

Le Sudoku est modélisé par un problème à satisfaction de contraintes. L'algorithme de recherche utilisé ici est l'un des plus populaires : la *backtracking search*. Ce dernier fonctionne sur le principe de la récurrence comme la plupart des algorithmes de construction d'arbre. Il est implémenté comme suit :

```
/// <summary>
/// Run a backtracking search on a given Sudoku.
/// </summary>
/// <param name="p_sSudokuToSolve"> The Sudoku to solve. </param>
/// <returns> The resulting Sudoku if solved, null otherwise. </returns>
private static Sudoku BacktrackingSearch(Sudoku p_sSudokuToSolve)
{
    // Handles the case of backtracking.
    if (p_sSudokuToSolve == null)
    {
        return null;
    }
    // Variable to store the backtracking result.
    Sudoku sBacktrackResultingSudoku = null;
    // If the Sudoku is solved, return the result.
    if (IsSolved(p_sSudokuToSolve))
    {
        return p_sSudokuToSolve;
    }
    // Find the cell with the smallest list of possible values.
    Coordinate cUnassignedVariableCoord = CellWithMinimumRemainingValue(p_sSudokuToSolve);
    // As long as we have some values in the list of possible values, we keep going deeper in the tree.
    while (p_sSudokuToSolve.SudokuGrid[cUnassignedVariableCoord.Row, cUnassignedVariableCoord.Column].PossibleValues.Count > 0)
    {
        // In the list of possible values of the chosen cell, get the one with the most constraint from peers.
        int iValue = ValueWithMaxConstraints(p_sSudokuToSolve, cUnassignedVariableCoord);

        // If the value with most constraints is consistent with the assignement, i.e. if not present in a peer's cell value.
        if (ComputePossibleValues(p_sSudokuToSolve, cUnassignedVariableCoord).Contains(iValue))
        {
            // Run a backtracking search with this new cell value and with the peers' value updated.
            sBacktrackResultingSudoku = BacktrackingSearch(AssignValueWithForwardChecking(Clone(p_sSudokuToSolve), cUnassignedVariableCoord, iValue));
            // Used to backtrack once the solution is found.
            if (sBacktrackResultingSudoku != null)
            {
                return sBacktrackResultingSudoku;
            }
        }

        // if the peers update failed, i.e. if one peer is not assigned and has no possible values remaining.
        // m_iBacktrackNumber is used to keep track of the count of backtracks.
        m_iBacktrackNumber++;
        // Remove the most constrained value from the list of possible values.
        p_sSudokuToSolve = RemoveValueFromPossibleValues(p_sSudokuToSolve, cUnassignedVariableCoord, iValue);
    }
    return null;
}
```

Figure 1 – Implémentation de l'algorithme de *backtracking search*

Son fonctionnement est le suivant :

- Dans le Sudoku, on choisit la cellule sur laquelle on va travailler (soit la première n'ayant pas de valeur, soit celle ayant le moins de valeurs possibles, nous verrons la distinction dans la partie 4).
- Tant que cette cellule possède des valeurs possibles, on réalise les actions suivantes :
  - On choisit une valeur dans la liste des valeurs possibles (soit la première disponible, soit celle ayant le maximum de contraintes, nous verrons cela dans la partie 5).
  - Si cette valeur n'est pas déjà prise par un de *peers* de la cellule considérée, on relance une *backtracking search* sur le Sudoku modifié comme suit :
    - On supprime de la liste des valeurs possibles des *peers* la valeur attribuée, c'est la propagation de contraintes.
    - On met à jour la valeur de la cellule.
  - Si la propagation de contraintes n'entraine pas d'erreur, *i.e.* si aucun des *peers* ne se retrouve sans valeur possible, on continue à approfondir l'arbre.
  - Dans le cas contraire, on supprime la valeur testée des valeurs possibles de la cellule et on recommence avec une autre valeur.
- La récurrence se termine sur un des 2 cas suivants :
  - Soit le Sudoku est complété (`IsSolved()` revoie `true`), on retourne le Sudoku résolu.
  - Soit le Sudoku ne comporte pas de solution, on retourne `null`.

On va maintenant présenter en détail le fonctionnement de la propagation de contraintes.

3. Propagation des contraintes

Le Sudoku pourrait être résolu en brute-forçant en choisissant pour chaque case non assignée une valeur satisfaisant les contraintes d'unicité de valeur. Néanmoins, il est possible de jouer bien plus finement en utilisant la notion de propagation de contraintes. Ici nous allons opter pour le *forward checking* qui est implémenté comme suit :

```
/// <summary>
/// Assign p_iValue to the cell with the coordinates p_cCoordinate and remove the value
/// with the ForwardChecking method from the peers' list of possible values.
/// </summary>
/// <param name="p_sSudoku"> The Sudoku to consider </param>
/// <param name="p_cCoordinate"> The coordinates of the cell considered. </param>
/// <param name="p_iValue"> The value to give to the cell and to remove from the peers' list of possible values </param>
/// <returns> The Sudoku with the cell value changed and the peers' list of possible values updated. </returns>
private static Sudoku AssignValueWithForwardChecking(Sudoku p_sSudoku, Coordinate p_cCoordinate, int p_iValue)
{
    for (int i = 0; i < p_sSudoku.SudokuGrid[p_cCoordinate.Row, p_cCoordinate.Column].Peers.Count; i++)
    {
        // We remove the given value from the peers' list of possible values.
        p_sSudoku = RemoveValueFromPossibleValues(p_sSudoku, p_sSudoku.SudokuGrid[p_cCoordinate.Row, p_cCoordinate.Column].Peers[i], p_iValue);
        // Check if a peer has its list of possible value empty and has no value assigned, i.e. need to backtrack.
        if (p_sSudoku.SudokuGrid[p_sSudoku.SudokuGrid[p_cCoordinate.Row, p_cCoordinate.Column].Peers[i].Row,
                p_sSudoku.SudokuGrid[p_cCoordinate.Row, p_cCoordinate.Column].Peers[i].Column].PossibleValues.Count == 0 &&
                p_sSudoku.SudokuGrid[p_sSudoku.SudokuGrid[p_cCoordinate.Row, p_cCoordinate.Column].Peers[i].Row,
                p_sSudoku.SudokuGrid[p_cCoordinate.Row, p_cCoordinate.Column].Peers[i].Column].Assigned == false)
        {
            return null;
        }
    }
    // If the value doesn't leave a peer without possible value, assign this value to the given cell.
    p_sSudoku.SudokuGrid[p_cCoordinate.Row, p_cCoordinate.Column].CellValue = p_iValue;
    p_sSudoku.SudokuGrid[p_cCoordinate.Row, p_cCoordinate.Column].Assigned = true;
    return p_sSudoku;
}
```

Figure 2 – Implémentation de la propagation de contraintes *Forward Checking*

Ce dernier fonctionne sur le principe suivant : lorsqu'on assigne une valeur à une cellule, on l'enlève de la liste de ses *peers*. Si après cette suppression, un des *peers* se retrouve avec une liste de valeurs possibles vides et n'a pas de valeur assignée, on retourne `null` afin de permettre à la *backtrack search* de remonter au nœud précédent de l'arbre et d'essayer une autre valeur.

4. Heuristique dans le choix de la cellule

Comme vu dans la partie 2, on a deux façons possibles de choisir les cellules : soit la première n'ayant pas de valeur, soit celle ayant le moins de valeurs possibles. La première façon est implémentée de la façon suivante :

```
/// <summary>
/// Look for the next unassigned cell.
/// </summary>
/// <param name="p_sSudoku"> The Sudoku to consider. </param>
/// <returns> The coordinates of the first unassigned cell. </returns>
private static Coordinate FirstUnassignedCell(Sudoku p_sSudoku)
{
    for (int i = 0; i < p_sSudoku.SudokuSize; i++)
    {
        for (int j = 0; j < p_sSudoku.SudokuSize; j++)
        {
            if (!(p_sSudoku.SudokuGrid[i, j].Assigned))
            {
                return new Coordinate(i, j);
            }
        }
    }
    return null;
}
```

Figure 3 – Implémentation de la recherche de cellule basique

Ici on prend simplement la première cellule n'ayant pas de valeurs et on part de celle-ci pour construire l'arbre des solutions.

On peut améliorer cette méthode en prenant comme cellule de travail la cellule possédant le moins de valeurs possibles afin de diminuer le nombre de *backtracks*, c'est la méthode de *Minimum Remaining Value* ou MRV :

```
/// <summary>
/// Look for the cell with a minimum number of possible values and return the first one found.
/// </summary>
/// <param name="p_sSudoku"> The Sudoku to consider. </param>
/// <returns> The first cell with a minimum number of possible values. </returns>
private static Coordinate CellWithMinimumRemainingValue(Sudoku p_sSudoku)
{
    int iCurrentMinRemainingValue = p_sSudoku.SudokuSize + 1;
    Coordinate cResult = new Coordinate();

    for (int i = 0; i < p_sSudoku.SudokuSize; i++)
    {
        for (int j = 0; j < p_sSudoku.SudokuSize; j++)
        {
            if ((p_sSudoku.SudokuGrid[i, j].PossibleValues.Count < iCurrentMinRemainingValue) && (!(p_sSudoku.SudokuGrid[i, j].Assigned)))
            {
                iCurrentMinRemainingValue = p_sSudoku.SudokuGrid[i, j].PossibleValues.Count;
                cResult.Row = i;
                cResult.Column = j;
            }
        }
    }
    return cResult;
}
```

Figure 4 – Implémentation de la recherche de cellule par MRV

Son fonctionnement est assez simple : on parcourt l'ensemble des cellules du Sudoku à résoudre. Pour une cellule donnée, si le nombre de valeurs possibles est inférieur à celui de la cellule ayant le plus petit nombre de valeurs possible, ses coordonnées sont stockées, sinon on passe à la cellule suivante et ainsi de suite. Si plusieurs cellules ont le même nombre minimal de valeurs possible, c'est la première rencontrée qui est conservée.

Ce choix s'est révélé payant comme le montrent ses résultats suivants (l'heuristique du choix de la valeur à assigner est le `ValueWithMaxConstraints()` de la partie 5) :

Sudoku choisi	`FirstUnassignedCell()`		`CellWithMinimumRemainingValue()`	
	Temps d'exécution /ms	*Backtracks*	Temps d'exécution /ms	*Backtracks*
9Easy	12	208	2	0
9Medium	15	294	4	17
9Hard	3300	83 436	30	541
9Hardest	18 000	449 355	420	9671
16Medium	210 000	911 013	120	100

Charte des couleurs :

- Vert : meilleure valeur pour un Sudoku donné ;
- Rouge : pire valeur pour un Sudoku donné ;
- Orange : égalité.

Le temps d'exécution et le nombre de *backtracks* est en effet énormément diminué ce qui confirme la pertinence de cette optimisation.

5. Heuristique dans le choix de la valeur à assigner

De la même façon que pour la partie 4, le choix de la valeur à donner en premier à la cellule sélectionnée se pose. Pour cela, on peut tout d'abord réaliser un choix simple : on prend la première valeur disponible dans la liste de valeurs possibles :

```
/// <summary>
/// Return the first item of the list of possible value for a given cell.
/// </summary>
/// <param name="p_sSudoku"> The Sudoku to consider. </param>
/// <param name="p_cCellCoordinate"> The cell coordinates.  </param>
/// <returns> The first item of the list of possible value for a given cell. </returns>
private static int FirstPossibleValue(Sudoku p_sSudoku, Coordinate p_cCellCoordinate)
{
    return p_sSudoku.SudokuGrid[p_cCellCoordinate.Row, p_cCellCoordinate.Column].PossibleValues[0];
}
```

Figure 5 – Implémentation basique de proposition de valeur

Néanmoins, on peut gagner en efficacité en sélectionnant la valeur ayant le plus (ou le moins) de contraintes venant de ses *peers*. Dans un cas, on fait le choix soit d'écarter au plus vite les chemins ne menant pas à la solution (maximum de contraintes), soit favoriser les chemins menant possiblement plus rapidement à la solution (minimum de contraintes).

On obtient les résultats suivants (l'heuristique du choix de la cellule est le `CellWithMinimumRemainingValue()` de la partie 4) :

Sudoku choisi	Valeur avec un minimum de contraintes		Valeur avec un maximum de contraintes	
	Temps d'exécution /ms	*Backtracks*	Temps d'exécution /ms	*Backtracks*
9Easy	2	0	2	0
9Medium	3	17	4	17
9Hard	254	5614	30	541
9Hardest	420	9204	420	9671
9NoSolution	165	202	120	100

On constate que sur le panel de Sudokus testés, la version avec un maximum de contraintes semble proposer un nombre de *backtracks* égal ou moins important dans la majorité des cas, elle est donc implémentée comme suit (en utilisant le `CellWithMinimumRemainingValue()` de la partie 5) :

```
/// <summary>
/// Return the item in the list of possible value which appear the most in the peers' possible values.
/// </summary>
/// <param name="p_sSudoku"> The Sudoku to consider. </param>
/// <param name="p_cCellCoordinate"> The cell coordinates. </param>
/// <returns> The item in the list of possible value which appear the most in the peers' possible values. </returns>
private static int ValueWithMaxConstraints(Sudoku p_sSudoku, Coordinate p_cCellCoordinate)
{
    int[] aiConstraintCountForValues = new int[p_sSudoku.SudokuGrid[p_cCellCoordinate.Row, p_cCellCoordinate.Column].PossibleValues.Count];

    for (int i = 0; i < p_sSudoku.SudokuGrid[p_cCellCoordinate.Row, p_cCellCoordinate.Column].PossibleValues.Count; i++)
    {
        for (int j = 0; j < p_sSudoku.SudokuGrid[p_cCellCoordinate.Row, p_cCellCoordinate.Column].Peers.Count; j++)
        {
            if (p_sSudoku.SudokuGrid[p_sSudoku.SudokuGrid[p_cCellCoordinate.Row, p_cCellCoordinate.Column].Peers[j].Row,
                p_sSudoku.SudokuGrid[p_cCellCoordinate.Row, p_cCellCoordinate.Column].Peers[j].Column].PossibleValues.Contains(
                p_sSudoku.SudokuGrid[p_cCellCoordinate.Row, p_cCellCoordinate.Column].PossibleValues[i]))
            {
                aiConstraintCountForValues[i]++;
            }
        }
    }
    return p_sSudoku.SudokuGrid[p_cCellCoordinate.Row, p_cCellCoordinate.Column].PossibleValues[aiConstraintCountForValues.ToList().IndexOf(
        aiConstraintCountForValues.ToList().Max())];
}
```

Figure 6 – Implémentation de recherche de valeur ayant un maximum de contraintes venant de ses *peers*

Son fonctionnement est le suivant : on parcourt l'ensemble des valeurs possibles des *peers* de la cellule considérée. Pour chacune des valeurs possibles de la cellule, on regarde combien de fois elle est présente dans les valeurs possibles de ses *peers*. La valeur retournée est celle étant présente un maximum de fois dans les valeurs possibles des *peers*.

On va maintenant comparer la version basique de recherche avec la version maximum de contraintes :

Sudoku choisi	`FirstPossibleValue()`		`ValueWithMaxConstraints()`	
	Temps d'exécution /ms	*Backtracks*	Temps d'exécution /ms	*Backtracks*
9Easy	2	0	2	0
9Medium	4	17	4	17
9Hard	140	3062	30	541
9Hardest	420	10 041	420	9671
9NoSolution	180	220	120	100

On constate qu'ici le choix est moins déterminant que dans le cas du choix de cellule. Néanmoins, dans le cas de Sudokus nécessitant une résolution plus complexe, le nombre de *backtracks* et le temps d'exécution est inférieur, la version avec `ValueWithMaxConstraints()` est dont préférée.

Conclusion

Au cours de ce devoir, on a pu modéliser le jeu du Sudoku sous la forme d'un problème à satisfaction de contraintes. En utilisant les notions vues en cours telles que l'algorithme d'exploration *Backtracking Search* et la propagation de contraintes de type *Forward Checking*, il a été possible de réaliser un programme capable de résoudre n'importe quel Sudoku carré jusqu'à 81x81, et de détecter si ce dernier possède une solution (dans le cas du Sudoku9NoSolution).

Il a également été question d'améliorer le temps de parcours de l'arbre des solutions en appliquant plusieurs techniques : tout d'abord le choix efficace de la cellule à traiter par l'algorithme de MRV, qui a permis de diminuer grandement le temps d'exécution et le nombre de *backtracks*, puis le choix plus discutable de prendre comme valeur celle ayant le maximum de contraintes dans les *peers*.

Les temps de résolution restent relativement contenus (quelques secondes pour les Sudokus les plus difficiles) mais il serait possible de les améliorer et surtout de diminuer le nombre de *backtracks* en implémentant la propagation de contraintes *Arc Consistency* également vue en cours.