**Aide à jouer au Sudoku**

Ren Shen

Alexandre Fernandes-Bartolomeu

8 mai 2017

Présenté à Pr. Roger Nkambou dans le cadre du cours INF4230 (Hiver 2017)

**1 Introduction**

Notre projet consiste à résoudre des jeux de Sudoku, tout en fournissant l’aide au joueur pour le guider dans ces choix de jeu. Ainsi, si nous avons un Sudoku au début avec aucune case de jouée sauf celles qui sont fournies au début, il sera possible d’afficher la solution, mais aussi de dire au joueur, par exemple, quand il se trompe et quel coup il devrait effectuer à la place pour réussir la grille. Notre Sudoku consiste en une grille de 9x9 dont le but est de remplir la grille de chiffres de 0 à 9 de façon à ce que :

1. Toutes les lignes possèdent tous les chiffres
2. Toutes les colonnes possèdent tous les chiffres
3. Toutes les carrés 3x3 dans la grille possèdent tous les chiffres

**2 Problématique**

Le problème que nous tentons de résoudre est d’implémenter une intelligence artificielle qui sera en mesure de compléter la grille, mais aussi de fournir et d’expliquer les coups que devrait prendre l’utilisateur en cas d’erreur. Il s’agit d’un problème de satisfaction de contraintes. Ainsi, si l’utilisateur entre le chiffre 4 dans la première ligne, il faudra pouvoir lui dire que le chiffre ne peut aller à cet endroit et expliquer pourquoi, soit en montrant le fil d’exécution, soit en montrant le chiffre qui empêche ce chiffre d’être placé à cet endroit à ce moment. Dans le cas de notre projet, l’environnement est entièrement observable, car l’agent a accès à toute l’environnement à chaque instant. L’environnement est aussi déterministe, car l’état suivant est connu, donc le prochain coup au Sudoku va dépendre de la grille courante. L’environnement est épisodique, car il s’agit d’observer la grille et de n’effectuer qu’une seule action par après. L’environnement est dynamique. Comme il s’agit d’un joueur qui joue au Sudoku et que l’intelligence artificielle ne fait qu’aider le joueur, il doit s’adapter aux coups que le joueur a effectués. L’environnement est discret, car il existe un nombre limité et distincts de données sensoriels et d’actions. En fait, le coup joué par l’utilisateur et l’état de la grille est tout ce dont à de besoin l’IA pour pouvoir décider de sa propre action. Finalement, il s’agit d’un environnement à agent unique, ou seule notre IA se déplace dans cet environnement.

La problématique nécessite une intelligence artificielle pour résoudre le problème, car au vu des possibilités d’une grille de 9x9, il pourrait être très long pour cette approche pour trouver la solution à une grille en particulier, surtout au début quand la grille est vide.

Les métriques suivantes seront utilisées pour observer la validité de notre algorithme

**1**- Le temps de calcul de la solution ne devra pas prendre plus de 50 ms

**2**- La solution devra être trouvée si elle existe

La première métrique concerne la rapidité de notre algorithme, car il ne faudrait pas qu’à chaque fois qu’un utilisateur rentre un chiffre il ait à attendre que la solution se calcule. La deuxième métrique est une condition obligatoire dans un jeu pour que l’utilisateur ne perde pas son temps à jouer à un jeu avec aucune solution.

**3 Technique de IA retenue**

Le premier algorithme possible pour résoudre le problème de résolution du Sudoku est celui du *Backtracking Search*. Dans le cas du Sudoku, nous partons d’une case qui consiste au nœud parent, puis on passe au premier nœud fils qui satisfait les contraintes, énumérées dans l’introduction. Si tous les nœuds fils satisfont les contraintes, alors l’algorithme a trouvé la solution. Si on arrive au dernier fils du premier nœud parent et que celui-ci ne satisfait pas les contraintes, alors il n’existe pas de solution.

Le deuxième algorithme possible est l’algorithme stochastique. Avec cet algorithme il faut tout d’abord remplir la grille, puis regarder tous les doublons qu’il y a pour chaque lignes et colonnes. Une fois que l’on obtient les doublons pour chacun, il faut regarder quel carré 3x3 possède le plus de doublons, en combinant les doublons des lignes et des colonnes pour le carré 3x3. Il faut ensuite prendre ce bloc et le modifier de façon aléatoire jusqu’à obtenir moins de doublons qu’avant pour la grille. Lorsque cela est réussi, il faut recalculer les doublons pour choisir quelle case 3x3 modifier, jusqu’à arriver à zéro doublons pour toute la grille, ce qui voudra dire que la solution a été trouvée.

Un troisième algorithme qui existe est celui des *Dancing Links*, un algorithme de couverture exacte. Il s’agit de créer une matrice qui ne, représentée à l’aide de listes doublement chaînées de façon à ce que chaque ligne, colonne ou bloc contienne exactement chacun des n^2 nombres une seule fois. Il s’agit de déplacer les chaînons jusqu’à obtenir la solution voulue.

Un quatrième algorithme de résolution du Sudoku serait celui de programmation par contrainte et de propagation des contraintes. Comme le dit le nom, il s’agit de trouver toutes les contraintes possible pour le sudoku (27 pour un Sudoku 9x9), pour lesquelles chacune d’entre elles sait que chaque case d’une ligne, colonne ou sous grille sont différentes. Cela permet de réduire l’espace des solutions pour la grille et pour les cases. Ensuite, il s’agit de remplir la grille et avec toutes ces conditions nous aurons le résultat.

**Algorithme choisi** : *Backtracking Search*

**Raison** : La première raison est que de tous les algorithmes vus en classe, parmi les techniques énumérées ci-dessus, il s’agit de la technique vu la plus en profondeur. Au niveau de temps de programmation il s’agit de la méthode la plus rapide pour nous d’arriver à résoudre notre problématique. Toutes les solutions énumérées sont capables de résoudre un Sudoku de 9x9 dans un temps négligeable, donc le temps de calcul ne devrait pas être un facteur pour le choix de notre algorithme. Comme un Sudoku est un environnement où il y a des contraintes à respecter pour trouver la solution, backtracking search est un très bon algorithme dans les cas de programmation par contraintes. C’est ses contrainte qui permettront l’efficacité de l’algorithme. DE plus, il s’agit d’un algorithme déterministe, ce qui est le cas du problème du Sudoku, donc encore une fois bien adapté au problème.

**4 Particularités et Implémentation**

**4.1 Détails de l’implémentation**

L’implémentation est divisée en deux grandes parties. La première est l’implémentation de l’algorithme de *backtracking search* pour trouver la solution d’une grille de Sudoku en particulier. La deuxième partie concerne l’interface utilisateur pour jouer au jeu et afficher la solution.

En ce qui concerne l’algorithme de *backtracking search*, l’implémentation va comme suit:

**1.** L’algorithme trouve une case vide (0 représente une case vide dans notre

tableau), en y allant colonne d’abord, puis ligne ensuite.

**2.** Lorsque l'algorithme trouve une case vide, il essaye de mettre les chiffres

de 1 à 9 jusqu’à ce que les 3 conditions de la section 1 soient satisfaites.

**3.** Si, pour une case, l’algorithme ne trouve pas un chiffre satisfaisant aux

trois conditions de la section 1 pour les chiffres de 1 à 9, il efface le résultat de la case d’avant avec un autre chiffre.

**4.** L’algorithme répète les étapes 1 à 3 récursivement jusqu’à obtenir la

solution finale

En ce qui concerne l’interface, il s’agit d’une interface faite avec le *toolkit* Swing the Java. Il s’agit d’une interface minimaliste qui permet de jouer de façon efficace au Sudoku et qui apporte facilement au joueur l’état de la partie. Il est possible de cliquer sur chacune des case pour faire apparaître une mini grille pour choisir les chiffres de 1 à 9. Le clic droit sur une case va effacer le chiffre sur la case.

**4.2 Code externe**

Aucun code externe n’a été utilisé. Seules des librairies standards de java ont été utilisées, comme la librairie Swing ou bien AWT pour l’interface graphique.

**5 Expérimentations, résultats et limites**

L’algorithme de backtracking search fonctionne bien pour un problème comme le Sudoku. Dans un Sudoku standard le temps de calcul, même pour un grille considérée comme difficile par un utilisateur humain prends un temps négligeable pour calculer la solution de la grille. De plus, advenant que la grille ne contiendrait pas de solution valide, l’algorithme serait capable de trouver qu’il n’existe pas de solution et de le dire à l’utilisateur. En ce sens, c’est parfait dans le cas du Sudoku. En ce qui concerne les limites, dans le cas de notre algorithme pour le Sudoku la complexité est de l’ordre de O(n^m), où n est le nombre de chiffres que peut avoir une case (1 à 9 dans notre cas) et m le nombre de cases qui peuvent être vides (variable selon le nombre de chiffres donnés au début du Sudoku et dépendant de la taille de la grille. Au vu de la complexité algorithmique exponentielle de notre algorithme, il est sensé de dire que nous arriverions à une limite en ce qui concerne la taille de la grille assez rapidement si nous voulons que les calculs de la solution se fassent en un temps raisonnable.

Le temps d’exécution est très raisonnable. Dans notre test la méthode *chercherSolution()* ne prends que 9,88 ms à s’exécuter et l’ensemble du main, qui comprends la construction du tableau et l’affichage de la solution 176 ms (voir Annexe). Si un joueur veut avoir la solution du Sudoku il s’agit là d’un très bon résultat qui ne risque pas de faire s’impatienter le joueur en question. La première métrique est donc satisfaite. La solution est aussi trouvée, donc la deuxième métrique est satisfaite. Ainsi, notre algorithme répond aux attentes demandées et résous bien la problématique.

La grande limite concerne le nombre de solutions pour un Sudoku en particulier. Pour un Sudoku présentant une seule solution, *backtracking search* va trouver la solution et l’utilisateur pourra se fier aux erreurs que l’algorithme trouve lorsque l’utilisateur entre de mauvais coups. Toutefois, plus d’une solution, cela peut devenir beaucoup plus compliqué pour l’utilisateur et cela est extrêmement plus compliqué pour l’IA de trouver toutes les solutions selon les entrées de l’utilisateur. Prenons le cas à 2 solutions. Deux coups peuvent donner deux bonnes grilles à la fin, mais ensemble donnent une mauvaise grille. Comment décider quel coup est mauvais pour l’utilisateur? Le problème est plus grand plus il y a de solutions et plus l’utilisateur est avancé dans la grille, s’il a avancé plusieurs solutions en parallèle. C’est pour cette raison que pour le bouton “Afficher mauvais coups” et “Afficher solution” prennent pour hypothèse une et unique solution, soit celle du *backtracking search*. Optimalement, un bon Sudoku devrait avoir une seule et unique solution. Peut-être que nos niveaux sont uniques, peut-être pas. Ce n’est pas le but du travail pratique donc nous assumerons l’hypothèse pour nos résultats.

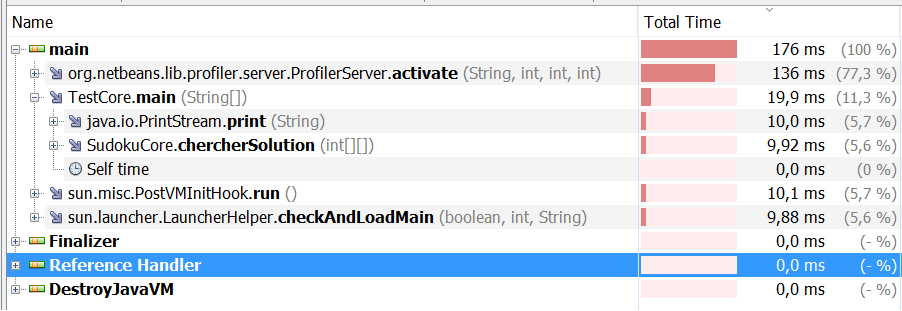
**6 Conclusion**

En conclusion, l’algorithme de backtracking search est un algorithme très bon pour résoudre un Sudoku standard de 9 lignes par 9 colonnes. Son temps de calcul et sa recherche exhaustive des possibilités lui permet de trouver une solution à chaque fois ainsi que de trouver l’absence de solution. Plusieurs solutions pour une grille ne peuvent que compliquer la tâche pour un algorithme dans un environnement à contraintes, car le domaine des solutions est beaucoup plus important s’il existe plus d’une solution pendant le fil d’exécution.

**7 Répartition des tâches**

Les tâches ont été réparties de façon équitables et tous les membres de l’équipe ont contribué à toutes les parties du projet.

**8 Annexe**



Temps d’exécution du main avec TestCore.java