TP1 – Jeu de Sudoku

Travail présenté à

M. Jian-Yun Nie

Par

Émile Trottier (p1054384)

François Poitras (p1053382)

IFT3335 – Introduction à l’intelligence artificielle

Université de Montréal

9 octobre 2015

1. Pour définir la résolution du jeu de Sudoku comme un problème de recherche dans l’espace d’états, il faut définir les éléments suivants :

* Un état correspond à un arrangement des chiffres dans la grille de Sudoku. On peut la voir comme une matrice 9x9 où chaque élément est un chiffre entre 0 et 9 inclusivement. Un zéro à la case (i,j) indique qu’elle est vide. Dans notre programme, nous utilisons la notion de nœud. Un nœud comprend un état, donc un tableau 9x9 indiquant une certaine disposition des chiffres dans une grille de Sudoku, ainsi que son coût, son nœud parent et une action. On peut voir le nœud comme une dérivation de son nœud parent lorsqu’on fait cette action.
* L’état de départ est la configuration initiale telle qu’elle nous est donnée dans le fichier 100sudoku.txt ou encore 1000sudoku.txt, mais évidemment sous forme de tableau 9x9 et non d’une ligne de 81 chiffres. Même si ces deux représentations ont le même sens, la première est beaucoup visuelle et beaucoup plus facile à lire et à comprendre.
* On ne peut pas définir l’état but puisque le problème est de le trouver (on ne le connaît pas au début). Par contre, on peut très bien définir une fonction de vérification de but. Il s’agit de vérifier que chaque ligne, chaque colonne et chaque boîte de la disposition X contienne chacun des chiffres entre 1 et 9 inclusivement. La méthode goal\_test(self, state) de la classe Problem de notre programme est une implantation de la fonction de vérification de but. Elle est trop longue pour être affichée dans le rapport malheureusement, il vous faudra regarder notre code.
* Nous avons déjà parlé en quelque sorte de la relation de successeur lorsque nous avons défini la notion de nœud. Une action se résume à transformer un zéro d’un état X en un certain chiffre entre 1 et 9 inclusivement pour se retrouver dans l’état Y. Dans cet exemple, le nœud correspondant à l’état Y stockerait un lien vers le nœud correspondant à l’état X (puisque c’est son parent) et l’action puisqu’elle est le lien entre ces deux nœuds. On dit donc que le nœud Y est un successeur du nœud X. L’ensemble de tous les nœuds B qu’on peut atteindre en une seule action à partir d’un nœud A, donc en changeant n’importe quel des zéros de la disposition A en un chiffre entre 1 et 9, est l’ensemble de tous les successeurs de A.
* Le coût d’étape est de 1 peu importe l’action qu’on fait. Cette caractéristique n’est pas très importante pour le problème de résolution du Sudoku vu qu’on sait d’avance combien nous coûtera la solution (ce sera le nombre de cases vides de l’état initial) et que nos décisions ne se baseront jamais là-dessus

**2. Description de l’implantation**

**2.1 Recherche en profondeur d'abord**

Il faut noter que la résolution du problème par profondeur d'abord n'a pas été implémentée en utilisant le code Python du livre. Il ne fait donc pas appel aux classes Node et Problem, mais la logique utilisée est similaire, puisque le pseudo-code du livre à servi de modèle pour la réalisation.

La recherche en profondeur d'abord est un algorithme de backtracking classique. En effet, nous commençons par localiser une case vide (définie comme contenant un zéro) et nous la remplissons par le premier nombre qui ne cause pas de conflits dans cette case. Puis, récursivement, on recommence le processus. Si, au cours de l'exploration des nœuds, on découvre un état qui fait en sorte qu'aucun nombre n'est possible pour une case, on cesse l'exploration et on remonte dans l'arbre. Si la limite de nœuds explorés est assez grande, cet algorithme trouve toujours la solution au puzzle. Par contre, il se peut que cela prenne du temps (voir section 3).

**2.2 Recherche par Hill-Climbing**

Pour cette méthode, la grille initiale est remplie par des nombres au hasard. La seule contrainte qui doit être respectée est celle de l'unicité des nombres dans les boîtes 3x3. Par la suite, on calcule il y a combien de conflits dans les lignes et les colonnes du puzzle. La fonctions à minimiser est ce nombre de conflits. Pour ce déplacer d'un voisin à l'autre, nous inversons deux nombres dans une même boîte 3x3. Les actions du problème sont donc différentes de la recherche heuristique et de la recherche en profondeur d'abord. Cette méthode peut être extrêmement efficace en termes de temps, mais elle ne conduit pas toujours à la solution optimale. En effet, il se peut que la fonction à minimiser atteigne un minimum local et ne cherchera pas à améliorer cet objectif. Aussi, il se peut que la fonction amène des voisins qui ont tous le même nombre de conflits. Cela signifie que nous avons atteint un plateau et l'algorithme ne peut plus progresser.

2.3 **Recherche heuristique**

...

**3. Comparaison des algorithmes sur les 100 configurations**

**3.1 Profondeur d'abord**

L'algorithme de profondeur d'abord n'est pas très efficace, mais il est plutôt simple à implémenter. En effet, sur les 100 configurations, il n'en résout que 3 avec une limite de 1000 nœuds explorés. Avec une limite de 5000, on a 13 configurations résolues. On passe à 20 résolutions avec une limite de 10 000. À partir de ce point, l'exécution du programme commence à être plus lente. La prochaine limite que nous avons testée est 50 000 nœuds. Cela nous a permis de résoudre 52 puzzles en un peu plus d'une minute. Pour résoudre l'intégralité des puzzles, il nous faut avoir une limite d'environ 3.71 millions de nœuds explorés, ce qui prends un temps considérable, soit environ 9 minutes.