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. Dans le cas de la recherche par Hill-Climbing, l'état initial est une grille remplie. La seule restriction est que chaque boite 3x3 ne doit contenir chaque chiffre qu'une seule fois.
- L'état but est un état dans lequel il n'y a plus aucun 0 dans la grille et où il n'y a aucun conflits. Nous décrivons plus en détail dans la section 2 le fonctionnement de la vérification de l'état-but.
- 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 (utilise Python 3, fichier depth-first.py)

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. Cela a fait en sorte que le développement de cet algorithme a été extrêment rapide, comparativement aux deux autres, qui ont posé plus de difficultés (voir les sections 2.2 et 2.3)

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 plus petit nombre qui ne cause pas de conflits dans cette case. Cette étape est cruciale, puisque lors de la vérification de l'état-but, on ne vérifie pas la validité de la grille. Tout ce que nous vérifions, c'est que le grille ne contient aucun 0. Si c'est le cas, la grille est implicitement valide, car nous ne donnons que des actions valides dans le problème d'exploration. Par exemple, si la case (x,y) contient un 2, il est impossible que l'action « placer un 2 » se retrouve dans une case qui se trouve dans la même boite, ligne ou colonne que la case (x,y).

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, cela signifie que nous avons fait une erreur plus tôt dans l'arbre. 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 (Utilise Python 2, fichier Search.py)

En raison de difficultés d'implémentations et de l'utilisation des packages fournis, la section qui suit décrit le comportement théorique du programme et ne reflète pas ce qui arrive réellement lors de l'éxécution. La section 3 détaille les problèmes rencontrés et les tentatives de solutions.

Pour cette méthode, la grille initiale est remplie par des nombres au hasard dans la classe ProblemHC, qui hérite de la classe Problem, afin de redéfinir les actions. 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 et de mémoire, 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 (Utilise Python 2, fichier Search.py)**

Comme pour la partie sur le Hill-Climbing, la partie heuristique décrite dans cette section est plutôt théorique en raison de bugs rencontrés lors du développement. La section 3 détaille se qui arrive lors de l'éxécution de cet algorithme, ainsi que nos pistes de solution.

Dans cette partie, l'heuristique utilisée est celle du nombre de possibilités dans chaque case. Pour un état donné, chaque case vide peut avoir entre 1 et 9 possibilités. Nous calculons la somme de toutes les cases et nous utilisons cette mesure comme heuristique. Nous allons donc explorer le nœud qui réduit le plus le nombre de possibilités globales sur la grille en premier. Cela est similaire à la manière dont un humain pourrait tenter de résoudre un puzzle de Sudoku. En effet, il n'est pas rare que certaines cases ne puissent contenir qu'un seul nombre. Il ne sert donc à rien d'explorer les autres nombres. Lorsque ces nombres sont placés, cela créée de nouvelles contraintes qui peuvent faire en sorte que de nouvelles cases, qui précédemment pouvaient avoir 2 possibilités, n'en n'ont plus qu'une seule. En enchaînant ces déductions, nous pouvons trouver la solution plus rapidement qu'avec la recherche en profondeur d'abord. Si nous nous trompons, nous pouvons reculer dans l'arbre, comme expliqué dans la section sur le backtracking.

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, pour les 100 puzzles.

Un autre aspect de l'implémentation qui peut causer des pertes de performances est l'ordre dans lequel les enfants d'un nœuds sont créés. Puisqu'il s'agit d'une recherche non-informée, il se peut que nous devions explorer les 9 enfants d'un nœud avant de trouver celui qui conduit à la solution.

3.2 Hill Climbing

La principale difficulté dans l'implémentation du Hill-Climbing a été de trouver un algorithme qui trouve efficacement toutes les inversions possibles de nombres dans une même boite. Pour chaque boite, il y a 8! possibilités et il a 9 boites à calculer, ce qui fait 9! possibilités pour l'ensemble d'une configuration du Sudoku.

C'est à cet endroit que notre méthode de recherche semble éprouver des difficultés. Les quatre boucles imbriquées (2 pour les coordonées des cases individuelles et 2 pour les coordonées des boites 3x3) ne parviennent pas à trouver toutes les combinaisons d'inversions et cela fait en sorte que la méthode Hill-Climbing ne fonctionne pas. En effet, l'algorithme n'arrive à résoudre aucun des 100 puzzles et retourne toujours des grilles remplies qui contiennent des conflits. Les conflits sont une chose normale, puisque lors du remplissage initial, nous ne spécifions pas que les nombres dans les lignes et les colonnes doivent être uniques.

Le temps d'éxécution est extrêmement rapide, mais pour la mauvaise raison. Au lieu de trouver des voisins qui ont des valeurs plus ou moins élevées, la méthode de Hill-Climbing nous indique que tous les nœuds ont la même valeur. Cela peut être dû au problème de permutations énoncé plus haut. Notre hypothèse est que puisque l'algorithme de calcul de permutations ne retourne pas l'ensemble de résultats, les voisins qui auraient une valeur plus petite, en termes de nombres de conflits, ne sont pas considérés et donc, la méthode de recherche de hill-climbing agit comme si elle avait trouvé un plateau.

Dans un cas où l'algorithme aurait été fonctionnel, il nous aurait été possible d'implémenter un « restart » lors de ce genre de comportement. Le « restart » est en fait tout simplement une ré-initialisation de la grille, avec des nouveaux nombres générés au hasard. Cela peut être fait dans l'espoir d'arriver sur une nouvelle partie de la fonction qui ne vas pas conduire au même plateau. Par contre, il nous faudrait limiter le nombre de « restarts », car l'algorithme pourrait ne jamais retourner de résultat, si la fonction à optimiser comporte trop de plateaux ou de minimums locaux.

3.3 Recherche heuristique

Notre algorithme de recherche heuristique semble ne pas tenir compte de la limite de nœuds imposée. Nous avons tenté de bloquer l'exécution de l'algorithme après que la limite ait été atteinte, mais sans succès. Soit la méthode ne s'arrête tout simplement jamais, ou alors une exception est levée, car le format de retour (Node, valeur de l'heuristique) n'est pas respecté. Dans la version remise, la condition d'arrêt n'est pas respectée. C'est pourquoi le code qui fait la recherche heuristique est commenté, pour pouvoir tester le Hill-Climbing qui se situe plus loin dans le code.

La principale difficulté a été dans l'implémentation de l'heuristique. En effet, dans l'énoncé du TP, il est mentionné que l'heuristique est calculée sur chacune des 81 cases du Sudoku. Or, dans le package du livre, la méthode recursive_best_first_search() s'attend à pouvoir calculer le maximum ou le minimum (selon le type de problème) entre deux nodes. Nous avons donc du trouver une solution de contournement et c'est de là qu'est née notre idée de faire une somme.

Notre analyse de l'efficacité de cette fonction est donc limitée au domaine théorique. Nous pensons que l'efficacité de cette méthode sera généralement plus grande que la recherche en profondeur d'abord. En effet, il s'agit de la même idée qui guide les deux recherches, mais dans le cas heuristique, nous utilisons l'information contenue dans la grille, pour guider nos choix de nœuds à explorer. Dans le pire des cas, il n'y a aucun nœud qui a une seule possibilité et beaucoup de nœuds ont deux possibilités. Dans ce cas précis, il nous faut trancher arbitrairement pour savoir quel nœud explorer. La probabilité de choisir le bon nœud, si nous avons n choix, est de 1/n.

Comme pour la recherche en profondeur d'abord, la recherche heuristique donnera toujours une solution, si la limite de nœuds à explorer est assez grande. Il est aussi fort probable que cette solution requière beaucoup moins que 3.7 millions de nœuds à visiter, en raison du fait que nous favorisons les cases ayant le moins d'enfants en premier.