Algorithmique pour la résolution de problèmes

Le jeu du Sudoku

Bourdon Gabriel

13 décembre 2024

Table des matières

1 Description du problème

Le Sudoku est un jeu de réflexion logique qui consiste à remplir une grille de 9x9 cases avec des chiffres, de manière à ce que chaque ligne, chaque colonne, et chaque région de 3x3 cases contiennent tous les chiffres de 1 à 9 sans répétition. Les cases sont initialement remplies avec certains chiffres et le joueur doit remplir les cases restantes en respectant les règles du jeu. Le défi consiste à remplir la grille en appliquant les règles tout en respectant les contraintes de ligne, colonne et sous-grille.

2 Modélisation formelle initiale

La modélisation du problème de Sudoku peut être définie comme suit :

— **État initial** : Une grille de Sudoku 9x9, où certaines cases sont déjà remplies et d'autres sont vides, indiquées par des zéros.

— **Actions** : Les actions possibles sont de remplir une case vide avec un chiffre entre 1 et 9, à condition que le chiffre ne soit pas déjà présent dans la même ligne, colonne ou sous-grille 3x3.

— **Successeur** : Une grille résultant de l'application d'une action, c'est-à-dire, une case vide est remplie avec un chiffre valide.

— **Test de but** : La grille est complète, c'est-à-dire que toutes les cases contiennent un chiffre entre 1 et 9, et chaque ligne, chaque colonne, et chaque sous-grille 3x3 respectent les règles du Sudoku.

— **Coût** : Le coût de chaque action est constant et égal à 1, puisque chaque placement d'un chiffre est une action unique sans priorités spécifiques.

3 L'algorithme de recherche initial

Pour résoudre le problème du Sudoku, j'ai utilisé plusieurs algorithmes de recherche, parmi lesquels je présente ici l'algorithme de recherche en profondeur (DFS).

Algorithm 1 Algorithme DFS pour la résolution du Sudoku

```
Initialiser la grille de Sudoku
Ajouter la grille initiale dans la frontière
while la frontière n'est pas vide do
    Extraire le nœud de la frontière
    if l'état est un état but then
        Retourner l'état
    end if
    for chaque action valide do
        Générer un nouveau nœud avec l'action
        Ajouter le nœud à la frontière
    end for
end while
Aucune solution trouvée
```

L'algorithme DFS explore les nœuds les plus profonds en premier. À chaque itération, il remplit une case vide avec un chiffre valide et explore l'état résultant. Si la grille est remplie correctement, l'algorithme termine avec succès. Sinon, il continue jusqu'à ce qu'une solution soit trouvée ou que tous les chemins possibles aient été explorés.

4 Améliorations par rapport à la première approche

Dans la première approche, un simple algorithme de recherche non-informée tel que DFS ou BFS a été utilisé pour explorer l'arbre des états. Cependant, ces algorithmes peuvent être inefficaces dans des grilles complexes en raison de la taille importante de l'arbre de recherche.

Pour améliorer la performance, j'ai ajouté une heuristique pour guider la recherche. En particulier, l'algorithme A* a été intégré, qui combine à la fois le coût du chemin parcouru ($g(n)$) et une estimation du coût restant jusqu'à l'objectif ($h(n)$) pour choisir quel nœud explorer en priorité.

4.1 Utilisation d'une fonction heuristique

Les algorithmes A* et Greedy Best First Search (GBF) sont particulièrement efficaces car ils utilisent une fonction heuristique pour guider leur exploration. Cependant, la performance de ces algorithmes dépend fortement de la qualité de cette fonction heuristique. Voici comment une fonction heuristique peut être définie et optimisée dans le contexte du Sudoku :

— **Principe de la fonction heuristique** : L'heuristique estime le coût restant pour atteindre l'objectif. Dans le cas du Sudoku, une heuristique efficace peut être basée sur le nombre de cases vides restantes ou sur la difficulté à remplir certaines cases.

— **Exemple d'heuristique simple** : Une heuristique pourrait consister à compter le nombre de cases vides sur la grille. Cependant, cette méthode peut manquer de précision car elle ne tient pas compte des contraintes de remplissage (lignes, colonnes, sous-grilles).

— **Heuristique avancée** : Une heuristique plus performante pourrait calculer le nombre de valeurs possibles pour chaque case vide. Une case avec peu d'options disponibles est plus contraignante et devrait être priorisée. Par exemple, $h(n)$ peut être défini comme la somme des "degrés de contrainte" des cases vides.

— **Optimisation de l'heuristique** : Les heuristiques peuvent être ajustées en combinant plusieurs facteurs, comme le nombre de conflits potentiels dans une ligne, une colonne ou une sous-grille, pour fournir une estimation plus précise.

En utilisant une fonction heuristique bien conçue, les algorithmes A* et GBF peuvent réduire considérablement le nombre de nœuds explorés, augmentant ainsi l'efficacité globale de la recherche.

5 Résultats obtenus

Les résultats de l'algorithme ont montré une nette amélioration en termes de temps d'exécution avec l'utilisation d'A* comparé à DFS pur. En utilisant l'algorithme Greedy Best First Search, on obtient un meilleur temps qu'avec A*, et on explore plus de la moitié moins de nœuds.

Pour évaluer ces performances, plusieurs grilles de Sudoku de complexité croissante ont été utilisées. Les grilles avec peu de cases remplies à l'état initial sont particulièrement intéressantes car elles nécessitent d'explorer davantage de solutions potentielles, mettant ainsi en évidence les forces et faiblesses des différentes approches. Le tableau suivant résume les performances pour une grille moyenne :

Algorithme	**Temps d'exécution (en secondes)**	**Nœuds explorés**
DFS	1.2773	47997
A*	0.0397	1136
Greedy	0.0181	495

Table 1 – Comparaison des temps d'exécution pour différentes approches de recherche

Comme le montre le tableau, l'algorithme DFS est le moins performant en termes de temps d'exécution et de nœuds explorés. Bien qu'il garantisse d'explorer toutes les solutions possibles, cette approche exhaustive devient rapidement inefficace face à des grilles complexes. En revanche, A* introduit une heuristique qui permet de réduire considérablement le nombre de nœuds explorés, ce qui améliore nettement les performances.

Greedy Best First Search (GBFS) pousse cette optimisation encore plus loin en se concentrant uniquement sur l'heuristique pour guider la recherche. Bien que cela puisse, dans certains cas, conduire à explorer des solutions sous-optimales, il offre des résultats très rapides dans le cadre de la résolution du Sudoku, où chaque placement de chiffre réduit significativement les possibilités restantes.

Pour illustrer ces résultats, voici un exemple de grille initiale utilisée lors des tests :

2	0	0	0	0	9	0	0	0
0	9	0	0	0	0	0	6	0
8	0	0	0	7	0	9	0	0
0	0	0	0	6	0	0	9	0
9	0	0	0	5	0	0	0	2
0	3	0	0	0	0	0	0	0
0	0	0	4	0	0	0	2	0
4	0	0	0	0	0	0	1	0
0	5	0	2	0	0	0	0	7

TABLE 2 – Grille de Sudoku utilisée lors des tests - 20 cases remplies à l'état initial

Pour mieux comprendre l'impact de l'état initial sur les performances, des grilles avec un plus grand nombre de cases remplies ont également été testées. Les résultats montrent que, dans ces cas, les algorithmes A* et GBFS explorent encore moins de nœuds, ce qui réduit encore davantage le temps d'exécution. Par exemple, avec une grille de 30 cases initialement remplies, GBFS n'a exploré que 136 nœuds en moyenne, avec un temps d'exécution de 0.007 seconde en moyenne.

Ces résultats mettent en évidence l'importance d'utiliser des approches heuristiques adaptées pour résoudre efficacement le Sudoku. Les algorithmes basés sur des heuristiques, bien que légèrement plus complexes à implémenter, offrent des gains significatifs en termes de performances, rendant possible la résolution rapide même pour des grilles difficiles.

6 Références

— Colares, R., *Algorithmique pour la résolution de problèmes*, INP Clermont Auvergne - ISIMA, 2024

— https ://fr.wikipedia.org/wiki/Sudoku