IFT3335 - TP 1

Jiadong Jin(20150692) Yuxiang Lin(20172116)

1 Description brève

Dans ce tp, on fait fonctionner le code de base avec les fichers donnés. Puis, nous implémentons une des heuristiques trouvée dans le site web `http://www.angusj.com/sudoku/hints.php`. Ensuite, nous implantons l'algorithme Hill-Climbing pour créer un autre moyen de recherche et puis le remplace avec le recuit simulé qui sert à améliorer la performance totale de recherche et résout les problèmes causés par l'algorithme dernier. Enfin, on compare les résultats apportés par les différentes méthodes de recherche et fait une conclusion sur leur performance.

2 Tester la méthode de Norvig

Nous ajoutons d'abord un autre méthode `solve` pour faire functionner le programme de base sur le fichier `1000sudoku.txt`. On le teste 10 fois sur les jeux et obtient le résultat suivant : la méthode de recherche de Norvig prend un temp moyen de 0.003 seconde avec une fréquence dans une échelle de 374 à 394Hz pour trouver la solution à chaque tour de jeu, dont le temp maximum varie de 0.008 à 0.012 seconde. Alors que pour le test du fichier `top95.txt`, on remarque que L'efficacité a considérablement diminué.

3 Petit changement de la méthode de Norvig

Ce qui rend l'algorithme de rechercher de Norvig spécial est qu'il ordonne les positions des carrés dans l'ordre croissant selon la taille de leur ensemble de chiffres candidats et chosit le carré vide qui a un ensemble de taille la plus petite pour commencer le remplissage(à l'aide de backtracking si les chiffres choisis ne fonctionneent pas). Donc, on fait un petit changement sur cette critère pour tester l'algorithme. Dans le code de source, on implante une autre méthode de recherche 'randomsearch' dont la première moitié ressemble à celle de Norvig, qui prend un dict de valeurs possibles de chaque carré en entrée et retourne le résultat avec tous les carrés remplis. Au lieu d'ordonner les positions et prendre celle dont l'ensemble est le plus petit, on chosit aléatoirement un carré et le remplit avec les chiffres possibles. Dans ce cas, après 10 tests aussi, l'algorithme a l'air un peu moins efficace : le temp moyen pris pour trouver une solution à

chaque tour est de 0.004 seconde avec une fréquence de 230 à 247Hz. Le temp maximum devient relativement élevé et se trouve entre 0.018 et 0.030 seconde. Ce résultat nous montre les avantages de l'algorithme de Norvig par rapport au celui de recherche aléatoire.

4 Implémentation des heuristiques

Ici, on implémente 2 heuristiques decrites dans le site web mentionné ci-dessus :

Hidden Single : L'algo "hidden single" vise à trouver le chiffre tel que l'ensemble de candidats du carré choisit est identique au celui des autres carrés non remplis dans 'peers' si le chiffre est enlevé, c'est-à-dire que ce chiffre est unique parmi les ensembles. Dans ce cas-là ce chiffre est la réponse finale pour le carré. On commence à l'aide de la fonction `find_hidden_singles` par parcourir l'ensembles des valeurs possibles de s et lister les peers y sont associés. Ensuite on compare l'ensemble de s avec celui des peers pour trouver le 'hidden_single'. On remplit s avec ce chiffre à la fin de cette fonction('values' est alors changé).

Naked pair : L'algo "naked pair" vise à trouver, dans 'peers', une paire de carrés qui ont les ensembles identiques qui contiennent seulement 2 condidats. Dans ce cas, aucun d'autres carrés dans le 'peers'(ligne, colonne ou carré de 3x3) ne peuvent être assignés avec les 2 chiffres. La fonction `find_naked_pairs` prend le dict 'value' et un carré s en entrée, parcourt ensuite le 'peers' de s afin de trouver la paire identique de taille de 2. Après ça, on enlève ces 2 chiffres de tous les autres ensembles de candidats.

Après l'implémentation des 2 heuristiques, on les met dans la fonction `randomsearch` pour tester leur performance. Cette fois-ci, on les teste à l'aide de trois fichiers : "1000sudoku.txt", "100sudoku.txt", "top95.txt". Voici les résultats :

Hidden Single : hidden_singles prend un temps moyen de 0.003 seconde avec une fréquence de 332 à 372Hz pour trouver une solution de `1000 sudoku`. Le temp max varie de 0.011 à 0.020 seconde. Quant au `100 sudoku`, le temps moyen reste 0.003 seconde mais la gamme de fréquence s'élargit, 318 à 383Hz, et le temps max passe de 0.007 à 0.011 seconde. Enfin pour `Top 95`, il faut un temps moyen entre 0.019 à 0.022 seconde pour réussir où la fréquence devient beaucoup plus faible(42 à 53Hz). Le temp max pris est moins stable que précédent(0.078 à 0.202 seconde).

Naked Pair : Naked_Pair aussi prend un temps moyen de 0.003 seconde avec une fréquence de 342 à 362Hz pour trouver une solution de `1000 sudoku`. Le temp max varie de 0.014 à 0.035 seconde. Quant au `100 sudoku`, le temps moyen et la fréquence restent les mêmes que ce qu'on à trouvés via hidden_single, et le temps max passe de 0.005 à 0.012 seconde. Enfin pour `Top 95`, il faut un temps moyen entre 0.018 à 0.025 seconde pour réussir où la fréquence devient beaucoup plus faible(40 à 55Hz). Le temp max pris est moins stable que précédent(0.129 à 0.234

seconde).

En comparant les statistiques des 2 heuristiques, on trouve que la différence entre leurs performance est très faible. On en déduit que le hidden single nécessite des recherches supplémentaires, même si on essaie de l'optimiser(lorsque le nombre d'occurrences d'un candidat atteint 2, ce tour de recherche se termine). Bien qu'un plus grand nombre de candidats puissent être éliminés lorsque naked pair est trouvé (par rapport à hidden single), la probabilité que naked pair se produise est plus faible et la recherche est plus coûteuse. Cependant, la probabilité que les deux situations se produisent n'est pas très élevée. L'optimisation provoquée par la suppression anticipée des candidats est presque la même que le coût provoqué par des recherches supplémentaires, il n'y a donc pratiquement aucune amélioration des performances.

5 Implémentation de Hill-Climbing

On remplit d'abord chaque carré 3x3 du Sudoku avec des chiffres au hasard, en veillant à ce que chaque carré respecte les contraintes du Sudoku en utilisant la fonction `initialize_grid`, puis on calcule le nombre initial de conflits sur les lignes et les colonnes pour la grille de départ à l'aide de `count_conflicts`.On trouve ensuite un échange de chiffres au sein d'un même carré 3x3 qui réduit le nombre total de conflits sur les lignes et les colonnes, réalise cet échange si et seulement si cela réduit le nombre de conflits et répète ce processus tant qu'une amélioration est possible. L'algorithme s'arrête lorsque aucun échange ne mène à une réduction des conflits, c'est-à-dire qu'un minimum local est atteint. Malheureusement, cette algorithme ne donne toujours pas de réponse. On assume que il est impossible de trouver la solution avec Hill-Climbing pour ce problem.

6 Implémentation du recuit simulé

Comme on n'arrive pas à trouver une solution via Hill-Climbing, on va l'améliorer en utilisant le recuit simulé.l'algorithme de recuit simulé trouve une configuration optimale de valeurs en minimisant les conflits. Il part d'une solution initiale et explore l'espace des solutions voisines, acceptant des changements basés sur une comparaison de "score" et la température actuelle, qui diminue progressivement. L'acceptation des solutions moins bonnes devient de plus en plus improbable à mesure que la température baisse, permettant une convergence vers une solution optimale. Un mécanisme de "reheat" est utilisé pour éviter les minima locaux en réinitialisant la température après un certain nombre d'itérations sans amélioration. Cette fois-ci, on arrive à trouver des solutions dans `1000 sudoku` avec un taux moyen de succès de 10.7%.(29.5% pour `top95.txt`) Le temp moyen pris passe de 1.057 à 1.409 seconde, alors le temp max varie de 3.084 à 3.585 seconde. La fréquence reste 0Hz tout le temps.

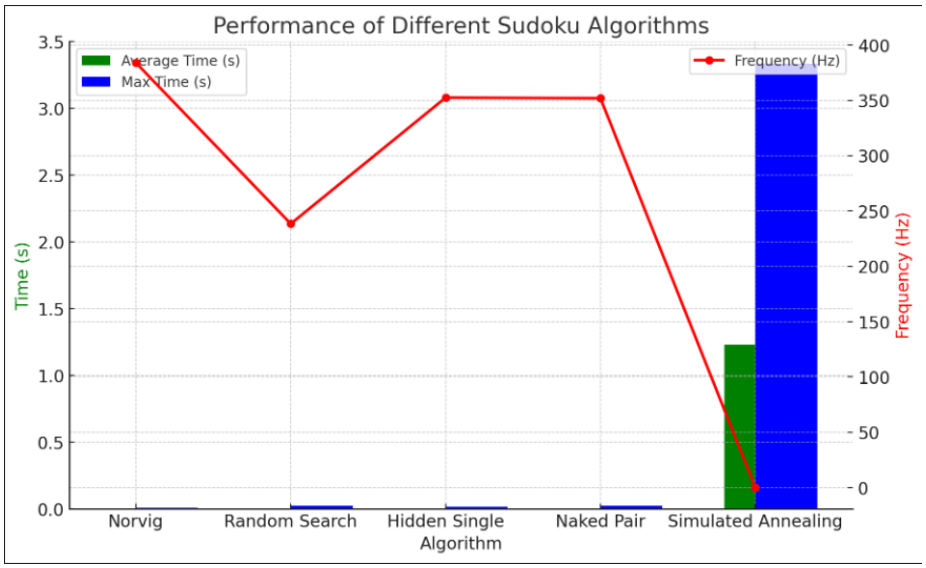

FIGURE 1 – Performance of Different Sudoku Algorithms

7 Comparaison des algorithmes

Temps moyen et maximal : L'algorithme de recuit simulé présente le temps moyen et maximal le plus élevé, dépassant nettement ceux des autres algorithmes. Le fait qu'il a un taux de succès plus élévé dans `top95.txt` que `1000 sudoku` indique que, bien qu'efficace pour trouver des solutions dans des scénarios complexes, il est évidement plus lent. Efficacité : Les algorithmes de Norvig, random serach, Hidden Single et Naked Pair montrent une grande efficacité avec des temps de résolution moyens très faibles (autour de 0.003 seconde) et des fréquences relativement élevées, ce qui les rend rapides et fiables pour la majorité des puzzles de Sudoku. Fréquence : La fréquence de réussite pour le recuit simulé n'est pas applicable (0Hz) en raison du temps de résolution(>1), tandis que les autres algorithmes présentent des fréquences significatives, démontrant leur capacité à résoudre rapidement les puzzles.