IFT3335 - Introduction à l'intelligence artificielle

Travail complété par:
Eran TROSHANI 20187276

Question 1

Programme de Norvig

Le programme de Norvig utilise une méthode de recherche qui ordonne l'évaluation des cellules afin de minimiser le nombre d'évaluations nécessaires. Il commence par les cellules ayant le moins de possibilités, car la découverte de solutions au cours de la recherche peut réduire les valeurs possibles pour les cellules situées dans la même unité que la solution trouvée. Cette stratégie entraîne une réduction significative du temps de recherche. Nous avons testé ce programme avec quatre configurations différentes : 100sudoku, 1000sudoku, 1000random et 95top. Parmi celles-ci, 1000random est composé de 100 sudoku différents générés aléatoirement à l'aide de la fonction random_puzzle(), 100sudoku et 1000sudoku sont des sudoku "facile", et 95top est 95 sudoku "difficile" qui sont plus complexes à résoudre. Nous pouvons voir leur vitesse de résolution dans le tableau ci-dessous:

Pour *100sudoku* et *1000sudoku*, qui comprennent des sudokus faciles à résoudre, le niveau de difficulté est similaire. Par conséquent, le temps nécessaire pour trouver la solution est très proche dans les deux cas. De plus, la différence entre les fréquences est également très faible, ce qui permet de conclure que les deux configurations présentent pratiquement le même niveau de difficulté.

Pour *1000random*, le temps moyen de résolution est aussi très court (0,00 sec). Par contre, le taux de réussite est de 0,999%. Il y a un sudoku non soluble par la recherche de Norvig, ce qui s'explique par le fait que les sudokus sont générés de façon aléatoire, ce qui offre la possibilité d'avoir un sudoku qui est très difficile. La fréquence de cette configuration est de 454Hz, qui est un peu plus basse que les deux configurations de niveau "facile". Cette différence est probablement causée par des sudokus générés de niveau "difficile" qui prennent plus de temps à résoudre que ceux de niveau "facile".

	taux de résolution	temps moyen (sec)	fréquence(Hz)	temps max (sec)
100sudoku	100/100	0,00	529	0,00
1000sudoku	1000/1000	0,00	533	0,01
1000random	999/1000	0,00	454	0,30
95top	95/95	0,01	160	0,03

Le temps maximum de résolution est 0,30 secondes, ce qui est beaucoup plus long que les autres configurations, mais cela est raisonnable parce qu'on a le plus difficile sudoku qui exige un temps de recherche élevé.

Pour *95top*, qui est composé de sudokus considérés "difficiles", le temps moyen de résolution est de 0,01 seconde, la fréquence est de 160Hz, et le temps maximum moyen est de 0,03 seconde. On peut voir que tous les sudoku sont résolus. Donc il n'y a pas de sudoku aussi difficile que celui qui n'a pas été résolu dans 1000random. Par contre, avec un temps moyen de résolution et une fréquence plus bas que les autres fichiers et tests, on peut déterminer que le niveau de difficulté est plus élevé et que la recherche est moins efficace pour des sudokus plus difficiles.

Question 2

Pour désactiver le 3ème critère, on définit quelques fonctions supplémentaires, et les fonctions principales sont : *randomSearch(values)* et *worstSearch(values)*. Ce sont deux fonctions similaires à la fonction *search(values)*, mais on change l'ordre d'évaluation. *randomSearch(values)* vise à choisir aléatoirement l'ordre de recherche des cellules. *worstSearch(values)* vise à faire l'inverse du 3ème critère. Elle évalue d'abord les cellules qui ont plus de possibilités au lieu des cellules qui ont moins de possibilités. Dans les tests effectués par les fonctions *solve_all_using_random* et *solve_all_using_worst*, on voit que la fréquence de *randomSearch* est moins grande que celle de *search* (~10% pour "facile",57% pour "difficile"), et celle de *worstSearch* est beaucoup moins grande que celle de *randomSearch* et de *search*(~13% pour "facile",~76% pour "difficile"), et toutes ces trois recherches trouvent toutes les solutions. Cela veut dire que la performance de ces deux fonctions est pire que la recherche avec l'heuristique de Norvig. De plus, pour résoudre les sudokus "difficiles", l'impact d'une bonne heuristique est beaucoup plus grand. C'est parce que dans ce genre de sudokus, les cellules ont souvent plus de possibilités à déterminer. Donc, on peut conclure que l'ordre qui minimise le nombre de possibilités à évaluer peut trouver les solutions plus efficacement, mais l'ordre de recherche ne détermine pas la complétude de la recherche puisqu'on trouve toujours toutes les solutions.

Question 3

Pour analyser l'influence des autres heuristiques supplémentaires, on ajoute deux autres heuristiques : naked pair et naked triple qui élimine les paires et les triples de carrés qui ont exactement 2 et 3 possibilités de valeurs afin de minimiser le nombre de recherche à faire. Les fonctions, qui ont toujours un taux de réussite de 100%, pour les tests sont solve_all_using_nakedPair et solve_all_using_nakedTriple. Dans les tests avec les sudokus "faciles", les recherches avec l'heuristique supplémentaire sont moins efficaces(ce taux varie entre différentes machines). C'est probablement parce que les sudokus ont très peu de probabilités dans les carrés, donc les possibilités retirées par la nouvelle heuristique ne diminuent pas assez le temps de recherche. Le temps d'évaluation de l'heuristique en testant s'il existe des "naked pairs" et des "naked triples" est plus grand que le temps économisé dans la recherche après l'évaluation. Cependant, pour l'ensemble des sudokus "difficile", la fréquence des algorithmes avec heuristique supplémentaire sont meilleurs que celle de l'algorithme original(vers 5%). Avec les tests, on voit que la performance de "naked pair" est légèrement meilleure que celle de "naked triple"(vers 1%), on peut dire que "naked pair" domine "naked triple".

Question 4

Pour l'algorithme Hill-Climbing, nous avons implémenté 3 fonctions. La première, *hillclimbing(grid)* est la fonction principale qui applique la logique de l'algorithme hill-climbing, où les cellules sont interchangées dans les carrés 3x3, et ce changement est accepté si et seulement si l'inter changement réduit les contradictions totales du sudoku. Sinon, le changement est refusé et les cellules retournent à leurs valeurs initiales avant le changement. Ensuite, il y a la fonction *randomizedGridFill(grid)* qui prend le grid initiale de sudoku, où il y a des valeurs manquantes, et rempli les valeurs manquantes avec des valeurs au hasard, tout en respectant les contraintes d'un subgrid 3x3, et sans prendre en compte les contradictions des rows et columns. Enfin, il y a la fonction *contradictionReader(grid)* qui sert à compter le nombre de contradictions total dans le sudoku, afin que la fonction hillclimbing puisse déterminer s'il y a une amélioration lorsqu'un "switch" de cellules est appliqué.

Avec l'algorithme construit, **comme demandé par l'énoncé du devoir**, nous avons eu un taux de succès de près de 0% pour la résolution des sudoku, peu importe le fichier ou test. Nous réalisons

que la cause vient du fait que, lorsque le grid est rempli avec des valeurs aléatoires, la fonction ne prend pas en considération les contradictions causées dans les rows et columns des cellules en question. C'est à dire qu'avec un changement des contraintes de la fonction *randomizedGridFill(grid)* où elle remplie le grid en considérant les contradictions dans les subgrids autant que dans les rows et columns, l'algorithme hill-climbing pourrait bel et bien résoudre des sudokus. Elle ne pourrait pas résoudre tous les sudoku, dépendamment de la complexité du sudoku et du potentiel d'atteindre des minimums locaux, mais elle aurait un certain taux de réussite.

Question 5

Nous avons réussi à implémenter une version simplifiée du recuit simulé de Lewis(en utilisant solve_all_using_annealing). Après avoir testé l'influence de la température de départ et du taux de diminution alpha. Nous observons que l'augmentation d'alpha peut augmenter le taux de succès de façon significative, mais ceci prolonge le temps moyen pour résoudre un sudoku. Et l'augmentation de la température initiale influence la probabilité, au départ de l'algorithme, de "sauter" vers un autre état, ceci influence le taux de succès d'une façon non linéaire. Nous trouvons finalement qu'avec un alpha de 0,95 et une température initiale de 6,0, nous obtenons un taux de réussite assez bon (entre 18% et 25%), ce qui correspond bien à l'incomplétude du recuit simulé. De plus, on observe que le taux de succès ne varie pas trop pour des sudokus de différents niveaux de difficultés. Selon le pseudo-code fourni dans les slides du cours, le temps d'exécution doit être de 1 à l'infini, mais en réalité, si un sudoku ne peut pas être trouvé facilement dans les itérations, on doit l'arrêter pour ne pas perdre du temps à rechercher une solution qui a seulement une très faible probabilité d'être trouvée. On configure une température d'arrêt très petite (5e-100) qui permet d'obtenir un temps moyen et un taux de réussite acceptable sur l'ensemble des sudokus. Nous avons rencontré plusieurs problèmes dans l'implémentation, mais nous finissons finalement par trouver des solutions pour résoudre ces problèmes. Avec plusieurs tests basés sur des configurations différentes des paramètres, nous avons réussi à obtenir une calibration gagnante.

Analyse Finale

Pour résumer, l'efficacité du programme de Norvig démontre l'importance des heuristiques dans la résolution de problèmes complexes. Les résultats montrent clairement que le choix et l'optimisation des heuristiques sont cruciaux pour la performance des algorithmes afin de résoudre des problèmes sudoku.

Les expériences avec *RandomSearch* démontre l'efficacité de l'heuristique choisie par Norvig(entre recherche en prof pure et heuristique de Norvig). *WorstSearch* montre qu'une heuristique mal choisie influence négativement la performance de la recherche. *Naked Pair* et *Naked Triple*, montrent que les heuristiques supplémentaires en plus d'une heuristique existante peut varier selon différents contextes("difficile" et "facile"). Toutes ces expériences renforcent cette idée en montrant comment des modifications des critères de sélection peuvent avoir un impact significatif sur les performances.

L'échec de l'approche Hill-Climbing illustre les limites de cet algorithme de recherche locale pour ce type de problème, où la solution optimale peut être entourée de nombreux minimums locaux.

Le recuit simulé, malgré un taux de réussite inférieur, démontre l'utilité des méthodes probabilistes pour explorer l'espace des solutions d'une manière qui peut parfois surpasser les approches déterministes, surtout dans les cas où une solution parfaite n'est pas nécessaire. Elle permet de potentiellement sortir des minimums locaux, ce qui est une optimisation par rapport à l'algorithme hill climbing. Elle permet aussi d'ignorer la difficulté du problème dans certains cas, il est très flexible

de choisir les configurations de température et de taux de diminution afin de manipuler le temps d’exécution et le taux de succès selon notre besoin du problème.