IFT 3335 - H21 -Devoir 1

Nom: Nicolas Sabourin (1068459)
Nom: Marc-André Chabot (20104579)
Nom: Alexandre Benguerel (20083359)

Tâche 1:

Pour faire fonctionner le programme de Norvig sur les tests fournis, il a fallu faire les mises à jour du code pour que ça fonctionne dans Python3. Pour les besoins du devoir nous avons ajouté un argument “heuristique” à la fonction solve_all() pour choisir dans la fonction search quelle heuristique sera utilisé pour résoudre le sudoku. Ce qui est fait alors c’est appeler dans le main : solve_all(from_file(“top95.txt”), “------”, None, heuristique). La valeur défaut de heuristique est 0. Pour le programme de Norvig original c’est l’heuristique “0”. En faisant un run de chaque programme il fera le search pour chaque heuristique pour 100 puzzles et pour hard1 deux fois.

Tâche 2:

Heuristique = “1”
Nous avons enlevé la partie du code qui choisissait la case avec le moins de possibilité et choisi avec random une case dans la liste en vérifiant que ce n’est pas une case qui est déjà fixée. Le temps double presque, pour trouver une solution sauf pour le “hard1” ou il y a une amélioration considérable vue qu’on choisit la case d’une autre façon.

Tâche 3:

Heuristique = “2”
Jusqu’ici nous avons choisi le numéro à tester dans la case en ordre croissant dans les choix du dictionnaire “values”. Nous avons plutôt opté pour le choisir au hasard dans les choix. C’est ici un peu moins rapide que Norvig sauf encore une fois dans le cas de “hard1”.

Heuristique = “3”
La case avec heuristique de Norvig. On ordonne les chiffres à explorer avec leur nombre de récurrence dans les peers (ligne, colonne, carré). Celui qui se répète le moins souvent ailleurs a plus de chance d'aller dans cette case. La vitesse est sensiblement supérieur à l'heuristique 2, mais meilleur pour le “hard1”

Heuristique = “4”
Nous explorons en utilisant l'heuristique de base de Norvig, excepté qu'à chaque node on vérifie s' il y a des “hidden singles” et on les assigne avant de continuer. La vitesse est semblable à Norvig vu que c’est très semblable, même pour la lenteur avec le “hard1”

Heuristique = “5”
Nous voulions vérifier si nous gagnons à mélanger les heuristiques 3 et 4

Tâche 4

L'algorithme de Hill-Climbing est implanté dans le fichier sudoku_hill_climbing.py.

Tâche 5

L'algorithme de recuit simulé est implanté dans le fichier simulated_annealing.py. Le paramètre alpha = 0.99 comme stipulé dans les notes et le t = 3 est utilisé en premier. Nous avons remarqué que notre algorithme roulait jusqu'à ce qu'il y est plus que 2 erreurs et et continuait indéfiniment pour trouver les bon deux chiffres à échanger pour finir si c'était possible. Nous avons alors décidé de l'aider en essayant directement les 4 permutations possibles s'il reste 2 erreurs sur 2 rangs ou 2 colonnes. Nous avons donc réussi à avoir un algorithme qui finit dans au moins 75% des cas.

Tâche 6

100 puzzle	Moyenne	Pire	% réussite	**Hard1**	% réussite
0	0.00363	0.00798	100	102.55	100
1	0.00549	0.01514	100	0.78	100
2	0.00427	0.00821	100	0.00798	100
3	0.00492	0.01215	100	0.00698	100
4	0.00435	0.00698	100	109.52	100
5	0.00434	0.00627	100	0.00600	100
Hill-Climbing	0.02	0.08	46	0.30	100
Simulated annealing	1.47	33.19	96	0.38	100

Comparaison de performances:

Au niveau de la performance des heuristiques 1 à 5. On se rend compte que toutes les heuristiques basées sur le depth first search finissent éventuellement par trouver un résultat, puisqu'en depth first search, toutes les possibilités sont éventuellement testées jusqu'à ce qu'on trouve le résultat (bien sur le backtracking permet d'éviter d'avoir à vérifier des solutions qui sont déjà disqualifiées d'avance). Dans la plupart des cas, l'heuristique de Norvig fonctionne très bien. Comme il est écrit dans sa documentation, le fait de prendre les cases qui ont le moins de valeurs possible, permet de limiter la recherche dans l'arbre. En effet, si on commence par prendre les cases qui ont le moins de valeurs possibles, on a une plus grande chance de tomber sur la vraie solution (par exemple si on prend une case qui a comme valeur candidate 1 et 2, on a 50% de chance de tomber sur la vraie valeur). Donc dans notre backtracking, on ne reviendra jamais plus haut dans l'arbre que cette valeur. Cependant, on voit que même si cette approche donne des bons résultats, comme elle est déterministe dans ses choix, il est possible qu'elle fonctionne très mal si on tombe sur une configuration qui fera que les choix développés seront systématiquement mauvais comme avec la grille "hard1", cela explique le fait que l'approche ou l'on prend une valeur aléatoire au lieu de la case dont il y a le moins de valeurs peut être meilleure dans certains cas, même si en moyenne l'algorithme de Norvig est environ 2x plus rapide.

Si on compare les heuristiques à base de backtracking vs les approches recherche locale, on voit beaucoup de différences de performances entres ses méthodes et dépendamment du sudoku choisi. En effet sur les problèmes simples de 100sudoku.txt, les heuristiques sont beaucoup plus efficaces en moyenne et en temps maximum que Hill Climbing ou simulated annealing. Dans ces cas là, les heuristiques trouvent une solution 100% du temps avec une très bonne moyenne et en temps max, tandis que les méthodes de recherche locale prennent beaucoup de temps et souvent reste bloqué. Simulated annealing a une meilleure performance au niveau du pourcentage de réussite que Hill-Climbing dû au fait qu'on introduit de l'aléatoire et qu'on choisit des cas qui n'améliorent pas le nombre d'erreurs, ce qui permet de se sortir d'une impasse, mais en général les deux peuvent tomber sur des problèmes simples et ne pas trouver de solutions, ou alors en temps vraiment énorme pour simulated annealing, dû au fait d'un manque de température qui engendre une faible probabilité de se sortir d'une impasse.

En revanche, sur des problèmes difficiles comme la grille hard1, le programme de Norvig est très mauvais (+/- 100 sec), mais dans le cas de simulated annealing et hill climbing, on a environ 0.3 sec. On peut expliquer cela simplement, depth first search avec peu d'heuristique doit rechercher énormément de nodes. hard1 possede seulement 17 valeurs fixés (minimum de valeurs théoriques pour un sudoku) de maniere a laissé beaucoup de valeurs possibles pour les cases restantes, ceci engendre donc un très grands nombre de nodes a vérifier et beaucoup de nodes à placés avant d'arriver à une collision. Ceci implique donc un grand nombre de calculs, puisque la fouille, sans être exhaustive, est beaucoup plus longue que sur des sudokus plus facile, chaque valeurs possible dans la dernière case d'une node engendre elle aussi une autre node à vérifier. Cela amène un algorithme qui grossit en temps exponentiel (même si en pratique le backtracking permet d'éviter une fouille exhaustive). Dans le cas du simulated annealing il fait simplement ajouter des valeurs a des cases de manière à diminuer le nombres de collisions, ce qui se fait rapidement et permet de converger vers une grille qui peut être proche de la solution. Bien entendu il peut bloquer comme dit précédemment, mais dans le cas où il ne bloque pas, il converge beaucoup plus rapidement vers une solution, car dans le principe même d'une recherche locale, il n'a pas à vérifier un arbre dont le nombre de feuilles augmente de manière exponentiel avec la hauteur dudit arbre.

En conclusion, nous avons testé plusieurs heuristiques avec plus ou moins de succès et tester d'autres heuristiques utilisant le principe de recherche locale. Ce qui en ressort, c'est le fait que certains problèmes se résolvent bien avec un algo depth first search, et d'autre avec un algorithme de recherche locale et qu'il est important de peser le pour et le contre. En gros les algorithmes depth-first-search nous garantissent une solution, au prix parfois d'un coût en temps très élevé si l'espace de recherche est très grand, tandis qu'un algorithme de recherche locale, ne nous garantit pas nécessairement une solution optimale (ou une solution tout court dans le cas du sudoku), mais on peut converger rapidement vers un résultat même si l'espace de recherche est grand. Dans un cas, le hill-climbing, c'est tres rapide et ca assume naivement une heuristique et peut rester bloquer si on est incapable de trouver une meilleure solution suivant cette heuristique. Simulated annealing vient pallier ce défaut et permet potentiellement de trouver une solution si nous sommes pris dans une impasse, mais est un peu plus lent dû à l'overhead de l'aléatoire, et au choix d'un critère d'arrêt si il reste bloqué.