IFT3335
TP1

Remis par
Alexandre Brilhante, matricule 862236
Yan Lajeunesse, matricule 20013795

17 mars 2017

Formulation du problème :

Notion d'état : grille carrée de 81 cases composé de 9 sous-grilles composé aussi de 9 cases contenant un chiffre entre 1 et 9 ou une valeur nulle.

État initial : grille carrée initialisée avec certaines valeurs entre 1 et 9 ou nulle.

État but : grille carrée contenant aucune valeur nulle respectant les contraintes (une ligne ne peut pas contenir 2 fois le même nombre., une colonne ne peut pas contenir 2 fois le même nombre et un sous carré de 3x3 ne peut pas contenir 2 fois le même nombre)

Successeur : une grille temporaire dont une des cases, excluant les cases contenant une valeur donnée au départ, peut être modifiée pour prendre une valeur entre 1 et 9.

Coût d'étape : 1 par étape. Non pertinent dans ce problème.

Description de notre implantation :

Nous avons modifié le code Python fournit. Afin de simplifier le traitement et l'affichage, nous avons eu recourt à la librairie numpy qu'il faut installer à partir de la ligne de commande "pip install numpy".

Nous avons créé sudoku.py qui contient le gros de notre code. Nous avons fait quelques modifications mineures à search.py :

- ajout de quelques lignes à chaque algorithme qui nous permettent d'afficher le nombre de nœuds explorés.
- modification de hill_climbing de manière à utiliser la valeur minimale au lieu de maximale, uniquement dans le but de simplifier, car nous travaillons directement avec le nombre de conflits et ainsi on souhaite le diminuer.
- modification des paramètres pour simulated_annealing.
- ajout d'une borne (paramètre "bound") pour depth_first dans le but de limiter la computation afin d'obtenir des statistiques plus claires et plus rapidement.

Pour sudoku.py, le point important à noter est que nous utilisons deux classes différentes, soit *Sudoku* et *SudokuHillClimbing*, dû au fait que pour hill_climbing et simulated_annealing, nous devons initialiser la grille de manière différente (tel que spécifié dans l'énoncé).

Pour *Sudoku*, nous implémentons des fonctions qui nous donnent les chiffres possibles à placer dans une case (en vérifiant si cela crée des conflits sur la ligne, la colonne ou le sous-carré). Nous implémentons aussi la fonction heuristique qui nous donne la case la plus contraignante pour l'algorithme best first.

Pour *SudokuHillClimbing*, nous implémentons des fonctions qui nous permettent de savoir quelle case nous pouvons échanger (afin de s'assurer que nous ne modifions pas une case donnée au début). Nous implémentons aussi une fonction qui nous permet de calculer le nombre total de conflits et ainsi permettre aux algorithmes de procéder.

Utilisation :

Recherche en profondeur d'abord : python q2.py
Recherche locale hill climbing : python q3.py
Recherche locale avec recuite simulée : python q4.py
Algorithme best first: python q5.py

Analyse :

Recherche en profondeur d'abord :
Dû au nombre très élevé de nœuds explorés, nous avons choisi d'ajouter une borne autrement l'algorithme prenait beaucoup de temps. Malgré cela, il ne trouve pas systématiquement de solutions sur les problèmes ayant trop de difficultés (donc de 0 dans la grille initiale). Cela étant, lorsque nous lui donnons un Sudoku relativement facile, il trouve une solution en explorant une quantité très élevée de nœuds (autant qu'avec l'algorithme best first).

Recherche locale Hill Climbing :
Hill Climbing ne trouve jamais de solutions sur les problèmes donnés, puisqu'il reste pris dans un maximum locale (techniquement un minimum local suivant notre implémentation). Cependant, comme recherche en profondeur, il arrive à trouver la bonne solution pour des problèmes plus simples. Le nombre de nœuds explorés est beaucoup plus petit car dès qu'il trouve un minimum local l'exécution se termine.

Recherche locale avec recuite simulée :
Contrairement à hill climbing, il ne reste pas pris au premier minimum local trouvé. Ainsi, cet algorithme parvient à trouver des solutions si et seulement si les paramètres fournissent lui permettent suffisamment de liberté (tel qu'avec une limite de 5000). Le nombre de nœuds explorés et donc beaucoup plus élevé qu'avec hill climbing, cependant les résultats en sont grandement améliorés.

Algorithme Best First :
Le choix de l'heuristique est important pour l'algorithme Best First. Nous avons choisi l'heuristique de la cellule la plus contraignante. Ceci donne donc un algorithme très vorace qui ne permet pas systématiquement de trouver les solutions puisqu'il doit faire certains choix lorsque la cellule la plus contraignante contient plus d'une possibilité. Ainsi, nous n'avons pas eu des performances satisfaisantes sur les problèmes Sudoku données. En utilisant des problèmes plus simples, il parvient tout de même à trouver des solutions. Aussi, le nombre de nœuds visités est donc plus élevé qu'en recherche locale mais moins qu'avec profondeur d'abord.

Conclusion :

Somme toute, les quatre algorithmes utilisés n'ont pas pu trouver de solutions pour les problèmes dans 100sudoku.txt. Nous avons testé ces algorithmes sur des problèmes plus simples afin d'en tirer des conclusions et de s'assurer que cela correspondait bien à la conception de ces algorithmes. Par exemple, sur un sudoku avec 2 zéros dans chaque sous-grille, recherche en profondeur trouve la solution en explorant 6161 nœuds en 2.89 secondes. Hill Climbing quant à lui trouve la solution en explorant 3 nœuds en 8.33 secondes. Pour la recuite simulé, il trouve la solution rapidement, cependant dû aux paramètres (la fonction exp_schedule) il continue de chercher une meilleure solution. Ainsi le nombre de nœuds explorés est grand et le temps aussi (ceci pourrait être corrigé simplement en vérifiant si on a la solution en cours de route et s'arrêter sur le champs). Pour Best First, avec le même problème Sudoku, l'algorithme n'a toujours pu pas le résoudre en moins de 2 heures.