IFT3335 — Sudoku

Vincent Antaki & Guillaume Poirier-Morency

9 octobre 2015

Numéro 1

L'état est représenté par une grille carrée de 81 cases pouvant contenir un chiffre entre 1 et 9 ou une valeur nulle sujette aux contraintes suivantes:

- une ligne ne peut contenir deux fois le même nombre
- une colonne ne peut contenir deux fois le même nombre
- une sous-grille 3 par 3 aux positions 0, 4 et 7 ne peut contenir deux fois le même nombre

L'état initial est une grille initialisée avec certaines valeurs.

L'état final est une grille contenant aucune valeur nulle.

Un successeur constitue une grille dont l'une des cases nulle prend une valeur entre 1 et 9 qui n'entre pas en conflit avec un autre élément sur la même ligne, colonne ou dans le même carré. 1Le coût d'étape est le même pour toute les cases (et il ne nous est pas vraiment utile).

Il y a donc une borne supérieure de $9 * n$ successeurs possible pour une grille de n cases nulles.

De manière interne, l'état est représenté par un tuple de 81 entiers hashable. Il est manipulé à comme une matrice NumPy afin de faciliter les traitements sur les carrés.

Le programme est séparé en trois modules principaux:

- `sudoku` contient les descriptions de problèmes
- `heuristics` contient les heuristiques
- `utils` contient quelques fonctions utilitaires pour manipuler les états des grilles

Les autres modules réalisent des parties spécifique de l'énoncé et peuvent être exécutés avec comme argument un fichier d'exemple contenant des grilles de Sudoku.

Des modifications ont été apportés sur le code fourni afin de pouvoir compter le nombre de noeuds explorés. Le nombre de noeuds explorés est défini par la taille de l'ensemble fermé pour les algorithmes de recherche dans un graphe et *best first*.

Il n'y a pas de borne sur Hill-Climbing, mais le code a été modifié pour compter le nombre de noeuds uniques explorés.

Tous les résultats des bancs d'essais se trouvent dans le répertoire `results`.

Numéro 2

Trois approches ont été utilisée pour définir l'ordre d'exploration dans la recherche en profondeur:

- en itérant sur les possibilités d'une case et sur les cases vides.
- ordonné par une heuristique
- ordonné aléatoirement

Le code correspondant est compris dans le module numero2 et il applique les trois approches de profondeur d'abord avec une borne de 10 000 explorations.

Lorsque testé sur 40 cas, aucune implémentation n'a réussi à terminer en moins de 10 000 itérations.

```
python numero2.py examples/100sudoku.txt
```

Numéro 3

En remplissant chaque case vide en respect avec les carrés et en définissant les actions comme des permutations de 2 éléments dans un carré qui n'étaient pas placé initialement, il nous est possible d'appliquer un algorithme de hill-climbing sur ce problème. Pour ce faire, un score est associé à chaque configuration. Dans notre implémentation, le score d'une configuration est l'inverse du nombre de conflicts (jusqu'à 2 par cellule; une pour la ligne et une pour la colonne) dans celle-ci car hill-climbing maximise et on souhaite minimiser les conflits. Après l'avoir fait rouler sur 1000 exemples, aucun Sudoku n'était résolu.

Nous avons aussi fait une autre variante qui a une définition d'action un peu différente. Nous faisons référence à celle-ci en temps que "super-branchement".

Une action est une permutation de 2 éléments dans un carré qui n'étaient pas placé initialement ou 3 permutations de 2 éléments non-placé initialement (dans 3 carrés différents). Il est intéressant de remarquer que toutes les actions de la première définition sont inclues dans cette variante. Par contre, un noeud aura beaucoup plus de noeuds adjacents. L'algorithme prendra donc beaucoup plus de temps de calcul pour faire une itérations mais pourra possiblement se rendre plus loin en moyenne.

Malheureusement, même après l'avoir rouler sur 1000 exemples, celle-ci n'a pas réussi non plus à résoudre un seul Sudoku.

```
python numero3.py examples/1000sudoku.txt
```

Numéro 4

Voici les heuristiques implémentées :

- *most constrained cell* consiste à favoriser le choix des cases qui ont le plus petit nombre de possibilités.
- *conflicts* compte le nombre de conflits afin de diriger la recherche dans un cadre d'un espace d'états défini par permutation sur les carrés.
- *remaining possibilities* compte la somme des possibilités restantes pour chaque case. Elle priorise le choix des configurations pour lequel cette valeur est la plus petite. Celle-ci s'est avérée peu efficace.

Aucune des heuristique n'a réussi à résoudre un Sudoku. Nous avons roulé sans succès un Sudoku sur *most constrained cells* pendant plus de deux heures.

```
python numero4.py examples/100sudoku.txt
```

Numéro 5

La recherche informée *greedy best first* était encore en exécution au moment de mettre ce rapport sous presse.

Les résultats prélimitaires pour les heuristiques on été calculés avec une borne de 100 noeuds explorés.

L'algorithme depth first search a été testé avec une borne de 10 000 itérations sur les 40 premiers exemples du fichier de 100 exemples avec les trois approches : séquentielle, aléatoire et triée.

L'approche séquentielle a pris en moyenne 27,36 secondes pour accomplir les 10k itérations avec déviation standard de 2,06 secondes alors que l'approche aléatoire

a pris en moyenne 24,62 secondes avec déviation standard de 1,62 secondes. L'approche triée a pris en moyenne 38,04 secondes avec déviation standard de 1,49 secondes.

L'algorithme Hill-Climbing a été testé sur 1000 exemples avec les deux stratégies.

Avec le branchement régulier, l'algorithme a atteint une moyenne de 14,964 conflits avec une déviation standard de 3,06 conflits. Il a exploré en moyenne 139 noeuds en 0,18 seconde.

Avec le super branchement, l'algorithme a atteint en moyenne 14,208 conflits avec une déviation standard de 2,86 conflits. Il a exploré en moyenne 404 noeuds en 1,25 seconde.

L'heuristique *most constrained cell* a exploré 100 noeuds avec une moyenne de 163 secondes et une déviation standard de 43 secondes.

L'heuristique conflicts a exploré 100 noeuds avec une moyenne de 125 secondes et une déviation standard de 138 secondes sur la représentation de grille remplie aléatoirement.

Notre stratégie d'élimitation de possibilités, décrite dans le numéro 6, a résolu tous les puzzles pour les 100 Sudokus avec une moyenne de 1,55 seconde et un pire cas de 3 noeuds explorés lorsque appliqué avec l'heuristique remaining possibilities. En roulant les mêmes 100 cas avec l'algorithme de profondeur d'abord, on obtient un temps moyen 11,27 secondes et un pire cas de 14 noeuds explorés.

```
python numero5.py examples/100sudoku.txt
```

Numéro 6 (bonus) Pour le bonus, nous avons implanté une recherche dans l'espace d'état avec élimitation de possibilités.

Dans cette représentation, l'état est une grille carrée de 81 case où chaque cellule correspond à un ensemble d'entiers de 1 à 9. Chaque ensemble représente les valeurs possibles que peut prendre la case. L'ensemble qui possède un seul élément représente la cellule qui contient cet élément.

Une réduction consiste à considérer les cases d'un seul élément et pour chaque case correspondante par ligne, colonne et carré, retirer cet élément de l'ensemble. On applique la réduction tant que le nombre de cases d'un élément ne se stabilise pas.

Une amélioration possible que nous avons envisagé pour rendre la représentation plus compacte aurait été d'utiliser un codage de Gödel pour coder les ensembles. Un nombre serait dans l'ensemble si l'entier qui le code est divisible par le nième nombre premier correspondant. Il aurait fallu encoder au plus 223092870 (2 * 3 * 5 * ... * 23) sur 27 bits. On applique les réductions en divisant les lignes, colonnes et grilles correspondantes au nombre placé dans une case.