Algorithmique:
Branch-and-Bound 1

Bonus :

Exercice : Seul l'exercice 1 de cette série (Taquin (15-puzzle)) fait l'objet du bonus.

Délai : mardi 26 novembre à 23h59 au plus tard

Fichiers : (avec votre nom et votre prénom à la place de 'NomPrenom')

- Un fichier NomPrenom.pdf répondant à toutes les questions de l'exercice.
- Un fichier python (NomPrenom.py) de votre implémentation de l'algorithme. N'envoyez pas le bytecode (fichier .pyc).

Rendu : Les fichiers doivent être soumis sur Moodle dans le devoir 'Bonus 5'.

Consignes : Le nom respect des consignes entraînera la non-évaluation de votre travail.

- L'implémentation et le rapport sont des travaux **individuels**.
- Respectez les spécifications de l'input et de l'output pour les fonctions demandées.
- Utiliser une version de Python > 3.0
- Rendez un rapport dactylographié au format pdf
- Mettez votre nom sur le rapport ainsi qu'un titre.

Conseil : Tester votre code avant de le rendre dans un nouvel environnement.

1 Taquin (15-puzzle)

Dans le jeu du Taquin, on dispose d'une grille de 16 cases (4×4) dont 15 cases numérotées et un trou, chaque case pouvant être coulissée si tant est qu'elle en ait la place, le but étant, à partir d'un état initial donné, de remettre la grille dans l'ordre de 1 à 15 (voir Figure 1). Nous avons vu en cours une description possible du problème, où une 16ème case fictive, la case vide, peut se déplacer dans au plus 4 directions à chaque tour. Le but de cet exercice est d'étudier une implémentation de stratégie Branch-and-Bound pour la résolution du Taquin.

1.1 Recherche Least Cost et fonction de coût (3 points)

1. Expliquez brièvement en quoi consiste une recherche Least Cost et quel est son rôle dans la méthode Branch-and-Bound.

FIGURE 1 – Exemple de jeu du Taquin

2. En cours, une fonction de coût $\hat{c}(x)$ associée à l'état x a été proposée pour le jeu du Taquin :

$$\hat{c}(x) = h(x) + g(x) \tag{1}$$

où $h(x)$ est la profondeur de x dans l'arbre de recherche et $g(x)$ le nombre de cases (hormis la case vide) mal placées. Quel est le rôle de $h(x)$ dans la fonction de coût ? Et celui de $g(x)$?

3. En se basant sur les critères discutés en cours concernant les propriétés de $\hat{c}(x)$ vis-à-vis d'une fonction de coût idéale $c(x)$, montrez que si l'on fait une recherche Branch-and-Bound avec la fonction de coût définie au point précédent et que l'on trouve une solution, cette solution est optimale.

1.2 Implémentation en python (3 points)

Implémentez en Python une méthode de résolution du Taquin en Branch-and-Bound, utilisant la fonction de coût $\hat{c}(x)$ (1).

Notez que ce qui nous intéresse ici n'est pas l'état final, puisque celui-ci est connu ; la solution est le chemin, c'est-à-dire la suite des directions prises par la case vide pour arriver à cet état-solution.

Implémenter la fonction de coût `c_hat(x)` où `x` est un E-node ainsi que la fonction donnant le chemin `solve(board)` où `board` est la liste contenant l'état initial.

Le format de ce chemin doit être une liste ordonnée de chaînes de caractères pris dans l'ensemble {'up', 'right', 'down', 'left'} : en lisant de gauche à droite cette liste, on obtient le chemin de l'état initial à l'état solution.

Bien que `solve` doive fonctionner pour tout état initial valide, on pourra au début prendre l'exemple donné en cours, avec l'état initial :

```
initial_board =  [1,   2,  3,  4,
                  5,   6, 16,  8,
                  9,  10,  7, 11,
                 13,  14, 15, 12]
```

La solution correspondante est le chemin : `[down, right, down]`

Si vous le désirez, vous pouvez vous aider du fichier *aidetaquin.py* disponible sur Moodle.

1.3 Efficacité de l'algorithme (3 points)

La fonction `gen_disorder(board, n)` désordonne un état donné 'board' (généralement l'état solution). Elle réalise `n` mouvements aléatoires afin de générer en output un état dont vous connaissez environ la distance à l'état caractérisé par 'board'. En effet, si `gen_disorder` prend soin de ne pas faire de cycles, alors on peut s'attendre à ce que la longueur du chemin-solution soit en général approximativement `n`, bien qu'inférieure parfois. Notez que le module `random` de Python permet de manipuler très simplement l'aléatoire.

Pour étudier l'efficacité de l'algorithme ou, plus précisément, de la fonction de coût utilisée, produisez un graphe du nombre de noeud k explorés par l'algorithme par rapport au paramètre n utilisé pour produire l'état initial avec la fonction `gen_disorder`. Incluez les graphiques produits dans votre rapport. Faites ceci pour :

1. Le cas où la fonction de coût est donnée par $\hat{c}(x) = h(x)$, et pour n de 1 à 6, en moyennant chaque valeur sur une vingtaine d'itérations au moins. À quel type de recherche cela correspond-il ?
2. Le cas de la fonction de coût (1), et pour n de 1 à 11, en moyennant chaque valeur sur une vingtaine d'itérations au moins. Que constatez-vous quant à l'efficacité de cette fonction de coût ? Que constatez-vous quant à la complexité en temps de l'algorithme ?