Algorithmique 2022 - TP 5

Brian Pulfer
Brian.Pulfer@unige.ch

30.11.2022

Remarques:

Veuillez suivre attentivement les spécifications de l'énoncé.

- **Réponses** - Essayez d'être exhaustif dans vos réponses. Les réponses comme "oui" et "non" ne permettent pas d'évaluer vos connaissances.
- **Points extra** - Le nombre maximum de points pouvant être marqués pour un TP est de 5, ce qui conduit à une note de 6. Vous pouvez effectuer les exercices 2 et 3 pour obtenir des points supplémentaires. Marquer plus de 5 points est toujours arrondi à la note 6.
- **Tracés** - Mettez toujours des étiquettes d'axe et des titres pour les tracés.
- **Implémentation** - Veuillez utiliser exactement les noms fournis pour les fonctions. Vous pouvez utiliser *pytest* avec le script de test donné pour vérifier votre implémentation.
- **Soumission**
  Merci de téléchargé vos devoirs sur moodle:
  - **NomPrenom.pdf** avec votre rapport pour tout les exercices
  - **tp5.py** avec votre implementations
  - *Pas d'autres fichiers* (.pyc, .ipynb, DS_Store, __pycache__, ...)

  Délai - 13.12.2022, 21:00 CET
- Pour toute questions et remarques, merci d'utiliser le mail Brian.Pulfer@unige.ch.

1 Taquin (5 Points)

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).

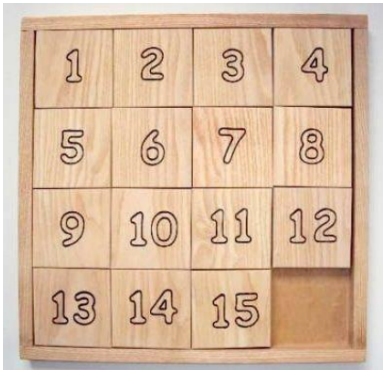

Figure 1: Example de jeu du Taquin

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 Fonction de coût

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 $g(x)$?
- A quel type de recherche correspond la fonction de coût (Equation 1)? Et dans le cas d'une fonction de coût $\hat{c}(x) = h(x)$?
- En se basant sur les critères discutés en cours concernant le propriétés de $\hat{c}(x)$ (Equation 1) 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

Implémentez en Python une méthode de résolution du Taquin en Branch-and-Bound, utilisant la fonction de coût $\hat{c}(x)$ (Equation 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(board)` où `board` est un E-node, node, ainsi que la fonction donnant le chemin `solve_taquin(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_taquin` doive fonctionner pour tout état initial valide, on pourra au début prendre l'exemple donné en cours, avec l'état 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"]`.

1.3 Efficacité de l'algorithme

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:

- 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 cinquantaine d'itérations au moins.
- Le cas de la fonction de coût (Equation 1), et pour n de 1 à 11, en moyennant chaque valeur sur une cinquantaine d'itérations au moins.

Et répondez aux questions suivantes:

- Que constatez-vous quant à l'efficacité relative de ces deux fonctions de coût ?
- Que constatez-vous quant à la complexité en temps de l'algorithme ?

2 Assignation de tâches (Extra 1 Point)

Dans ce problème, on veut assigner n tâches à n agents, sachant que chaque agent est rémunéré et chaque tâche est assignée à un agent différent.

C'est un problème que l'on peut couramment rencontrer: assigner des emplacements à des bâtiments devant être construits, assigner des évènements à des organisteurs candidats, etc... La rémunération des agents peut être représentée par une matrice des coûts, où l'élément (i, j) représente le coût de l'exécution de la tâche i par l'agent j.

Le but de l'exercice est d'implémenter une stratégie Branch-and-Bound pour résoudre ce problème, avec la matrice des coûts comme paramètre. Pour ce faire, nous proposons de décrire un état, ou solution partielle, par la liste des tâches déjà assignées et des agents déjà assignés (on commence donc avec des listes vides). De cette manière, on peut définir le coût d'une solution partielle par la somme des coûts minimaux pour chaque tâche restante, augmentée du coût effectif des assignations effectuées jusque là. Prenons par exemple la matrice des coûts ci-dessous:

```
1  #example : cost of task 1 by agent 2 is 17 (numbering starts at index 0)
2  cost_matrix = [
3  [11, 14, 11, 17],
4  [12, 15, 17, 14],
5  [18, 13, 19, 20],
6  [40, 22, 23, 28]
7  ]
8  #the optimal assignement for this cost matrix is :
9  #agent0-task0, agent1-task2, agent2-task3, agent3-task1.
```

Dans ce cas, le coût de la solution partielle 'agent0-task0' vaut 11 + min(task2) + min(task3) + min(task4) = 11 + 12 + 13 + 22 = 58, où '*min(task i)*' dénote la coût minimal pour la tâche i, en ignorant les agents déjà assignés (dans l'exemple, l'agent 0).

2.1 Optimalité

Une telle fonction de coût mène-t-elle à une solution optimale?

2.2 Implémentation

Implémentez l'algorithme pour une matrice des coûts de taille arbitraire n. La fonction `solve_tasks (cost_matrix)` doit retourner la liste des assignations (par exemple, `[0, 3, 1, 2]` dans le cas de la matrice des coûts cidessus).

3 Plus court chemin (Extra 1 Point)

Nous allons dans ce problème étudier un algorithme Branch-and-Bound pour trouver le chemin le plus court entre deux points dans un espace bidimensionnel discret ($n \times m$ cellules). Pour rendre ce problème intéressant, on place des obstacles entre ces deux points. On peut représenter cet espace par une matrice où chaque cellule peut prendre la valeur 0 (libre) ou 1 (obstacle). Le chemin ne peut évidemment pas passer par une cellule dont la valeur est égale à 1. Les mouvements autorisés sont gauche, droite, haut et bas (pas de mouvements en diagonale).

3.1 Fonction de coût

Trouvez une fonction de coût $\hat{c}(x)$ qui remplisse les critères nécessaires pour garantir l'optimalité d'une solution avec la méthode Branch-and-Bound.

3.2 Implémentation

Implémentez l'algorithme. La fonction `solve_shortest_path(domain, a, b)` retourne la liste des positions correspondant au chemin entre les points `a` et `b` du domaine.