Projet Taquin

Aix-Marseille Université - L3 Mathématiques - UE Programmation 3

October 19, 2022

1 Modalités de rendu

Ce projet est à rendre pour le lundi 14 novembre 2022 à 23h59 et peut être fait seul ou en binôme. Le rendu se fait sur Ametice en déposant le code que vous aurez développé ainsi qu'un léger compte-rendu précisant comment expliquer votre programme ainsi que les réponses aux questions théoriques. *Nous vous rappelons que tout plagiat de code d'autres groupes ou de code trouvé en ligne impliquera une note de 0 pour tous les groupes impliqués. Tout code copié doit faire référence à la source, que ce soit un autre élève, un site web, ou autre.* **Vous pouvez néanmoins utiliser sans les citer les corrections des TDs et TPs mises à disposition sur Ametice, et vous entraider tant que le code que vous rendez est bien le fruit de votre travail personnel.**

2 Introduction

Le jeu du Taquin est composé de $n^2 - 1$ petits carreaux numérotés qui glissent dans un carré de côté n de sorte qu'il y a une case vide à chaque instant. Il consiste à remettre dans l'ordre les $n^2 - 1$ carreaux dans la configuration finale donnée $(1, 2, 3, ..., n^2 - 1,$ *case vide*$)$ à partir d'une configuration initiale quelconque. A chaque coup, on peut déplacer un carreau situé sur une case voisine de la case vide vers celle-ci.
Vous pouvez vous familiariser avec le jeu en ligne : https://taquin.net/fr/taquin_3x3_std/i47.html?s=135874629
En pratique, si trouver une solution au jeu du Taquin est facile (vous pouvez vous amuser à trouver une méthode de résolution systématique), trouver la solution demandant le moins d'échange est un problème de complexité exponentielle (partie 1). C'est pour cela que l'on a recourt à une heuristique (parties 2 et 3).
N'oubliez pas de tester votre programme au fur et à mesure.
Je vous encourage à utiliser etulab (https://etulab.univ-amu.fr/, accessible avec votre compte amu) pour la gestion de votre projet. Vous trouverez une aide sur le fonctionnement de Git et d'etulab sur la page Ametice du cours.

3 Approche naïve

Dans cette première partie, nous nous proposons de résoudre le problème en explorant systématiquement toutes les combinaisons possibles, comme vu en cours avec l'agorithme de Dijkstra. **Dans cette partie, vous ferez attention à ne pas tester votre code pour $n > 3$ car l'execution de l'algorithme prendrait trop longtemps.**
Vous écrirez les programmes de cette partie dans un module *naif.py*.

3.1 Construction du graphe (2 points)

Pour un taquin de côté n, combien y a-t-il d'état possible ? Quel est le nombre maximal de voisins pour un sommet donné ?
En vous aidant de la classe graphe fournit dans le module *graph.py*, écrivez une fonction prenant en entrée un entier n, et renvoyant un graphe dont les sommets représentent les états possibles du taquin, et les arrêtes reflètentles transitions possibles.

Vous pourrez représenter les états par des tuples à n^2 éléments, comprenant les entiers entre 0 et $n^2 - 1$, le 0 représentant la case vide. L'élément de la ligne i, colonne j serait donc stocké dans la case d'indice $n * i + j$. **Indication :**

- envisager toutes les positions possible pour la case vide (en itérant sur i et j),
- générer toutes les permutations des chiffres de 1 à 8 pour compléter le tuple autour du 0 correspondant à la case vide.
- considérer les échanges possibles entre la case vide et ses cases voisines
- ajouter les arcs en conséquence

Pour obtenir la liste de toutes les permutations d'un tuple en python : module itertools. Ex : list(itertools.permutations((1, 2, 3)))

3.2 Détermination de la solution (2 points)

On considère que la configuration où les chiffres sont rangés de gauche à droite, de haut en bas par ordre croissant, et où la case vide est en bas à droite correspond à la configuration finale.
En vous appuyant sur l'algorithme de Dijkstra vu en cours, écrivez une fonction permettant de résoudre le taquin à partir d'une configuration initiale donnée en entrée. La fonction renverra le nombre de coupz minimal et un chemin de longueur minimal entre la configuration de départ et la configuration d'arrivée.
Notez que selon la configuration initiale, le taquin n'a pas forcément de solution (l'algorithme de Dijkstra renvoie alors une distance minimale infinie). Dans ce cas, la fonction renverra $False$.

3.3 Complexité (Bonus : 2 points)

A l'aide du module *time* de Python, mesurez le temps d'execution de l'initialisation du graphe et de l'algorithme de Dijkstra pour $n = 2$. Que remarquez-vous pour $n = 3$? Mesurer le temps d'execution de l'algorithme à l'aide du module *time*.
Quelle est la complexité temporelle de l'algorithme ? Justifiez.
Quelle est la complexité spatiale de l'algorithme ? Donnez une approximation de l'espace mémoire nécessaire pour n = 4 sachant qu'on peut stocker un état sur quelques octets.
Que pensez-vous de cette approche naïve ?

3.4 Un test rapide pour savoir si il existe une solution (2 points)

Tester votre programme pour l'exemple de Loyd [1] : $debut = (1, 2, 3, 4, 5, 6, 8, 7, 0)$.
Lisez cet extrait [1] : *"Il est possible de dire à l'avance si le problème posé est soluble ou non. En effet, la configuration initiale d'un taquin est une permutation de sa configuration finale. Cette permutation est dite paire si elle peut être obtenue par un nombre pair d'échanges successifs de deux cases, adjacentes ou non, vide ou non, appelés également transpositions. On montre que cette notion ne dépend pas du choix de la suite des échanges. Elle est impaire sinon. On associe également à la case vide une parité : la case vide est paire si l'on peut se rendre de la position initiale de la case vide à la position finale en un nombre pair de déplacements, impair sinon."*

En vous basant sur l'extrait précédent, écrivez un algorithme qui, étant donné une configuration initiale, réalise ce test.
Vous pourrez tester la parité d'une configuration en testant sa parité à l'aide du code ci-dessous (tiré de [5]), puis en comparant le résultat avec la parité de la configuration finale (Si deux permutations sont de même parité, alors les combinaisons allant de l'une à l'autre seront des permutations paires. A l'inverse, si elles n'ont pas la même parité, les permutations permettant d'aller de l'une à l'autre seront impaires).

```
def parity(p):
    return sum(
        1 for (x,px) in enumerate(p)
          for (y,py) in enumerate(p)
          if x<y and px>py
        )%2==0
```

Quel est la complexité de ce test ?

4 La classe Taquin (9 points)

On a vu dans la partie précédente que l'approche basée sur l'algorithme de Dijkstra n'était pas envisageable pour des questions de complexité temporelle. On se propose d'adopter une solution heuristique. Commencez par créer le module *taquin.py* contenant la classe $Taquin$. Cette classe a pour but de représenter l'état du taquin à un instant donné. N'oubliez pas de tester votre classe au fur et à mesure que vous la programmez.

- Le constructeur de la classe $Taquin$ initialisera l'attribut $configuration$ (reflétant la position courante) avec une position initiale passée en argument. Vous representerez la position dans une matrice carrée de côté n. Vous pourrez par exemple utiliser une liste de liste ou les tableaux définis dans le module *numpy*. Le constructeur créera également deux attributs *vide_i* et *vide_j* permettant de stocker la position de la case vide.

- Définir une méthode *horizon* renvoyant $\Sigma_{\alpha=1}^{n^2-1}|c_\alpha^i - \lfloor\frac{\alpha-1}{n}\rfloor| + |c_\alpha^j - ((\alpha-1) \mod n)|$, où c_α^i est le numéro de ligne de l'entier α dans la matrice $configuration$ et c_α^j est son numéro de colonne. Notez que c_α^i et c_α^j sont dans des entiers compris entre 0 et $n-1$. Nous expliciterons l'intérêt de la fonction *horizon* dans la partie 3. Notez que *horizon* renvoie 0 si et seulement si la configuration courante est la configuration finale.

- Définir une méthode *deplacer* prenant en argument deux indices i et j déplaçant l'entrée située en dans la case i, j vers la case vide. (On suppose que la case vide est adjacente à la i, j). N'oubliez pas de modifier les attributs en conséquence.

- Définir quatre méthode *haut* (resp. *bas*, *gauche*, *droite*) basée sur la méthode *deplacer*, déplaçant un carreaux en dessous (resp. au dessus, à droite, à gauche) de la case vide vers la case vide. Ces méthodes ne prennent pas d'argument.

- Définir quatre méthode *essaie_gauche*, *essaie_droite*, *essaie_haut*, *essaie_bas*. Chacune de ces méthodes vérifie que le déplacement effectué est possible. Si le déplacement réussi, la méthode renvoie $True$, sinon elle renvoie $False$.

- Définir une méthode *traduction*, qui renvoie un tuple correspondant à l'attribut $configuration$, sur le même format que dans la partie 1.

- Définir une méthode *lire* qui prend un tuple en entrée et met à jour l'attribut $configuration$ en conséquence. Cette méthode fait l'opération inverse de la précédente.

- En réutilisant le code de la question 1.4, définir une methode *est_soluble* retournant $True$ si la configuration initiale admet une solution et $False$ sinon.

- Définir une méthode $affficher$ affichant la matrice configuration. Vous pourrez utiliser le module matplotlib.pyplot [2] ou encore surcharger la méthode *__str__* pour qu'elle affiche la matrice ligne par ligne en sortie à l'aide la fonction *print*. Cette méthode sera très utile pour tester votre programme. Prenez soin de choisir judicieusement les couleurs d'affichage afin d'en garantir la lisibilité. (cf. figure 1)

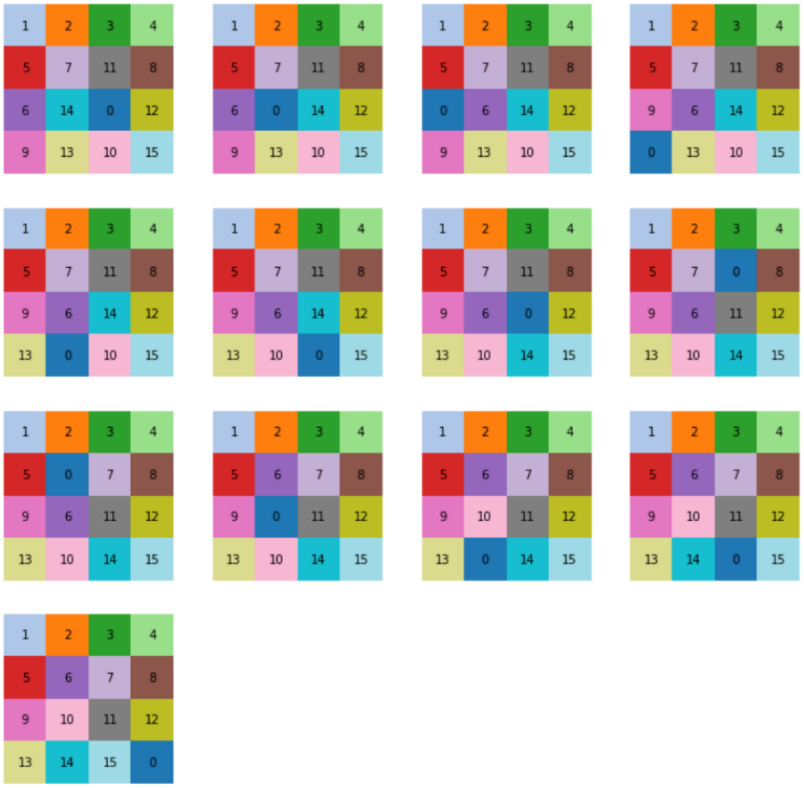

Figure 1: Exemple d'affichage d'une solution de longueure minimal, affichée de gauche à droite, de haut en bas.

5 Résolution à l'aide de l'algorithme IDA*

L'algorithme IDA* [3] est un algorithme de recherche en profondeur. Contrairement à l'algorithme de Dijkstra qui explore en priorité les sommets les plus proches du sommet initial, IDA* cherche en priorité à compléter un chemin en s'enfonçant le plus loin possible dans le graphe. L'algorithme abandonne cependant une branche si son coût $f(k) = g(k) + h(k)$ à un instant donné dépasse un certain seuil, où k est la configuration courante, $g(k)$ est le coût pour aller de la configuration initiale à k, et $h(k)$ est l'"horizon" de k (définit à la partie 2 dans notre cas). Pour tout k, $h(k)$ est inférieur au coût nécessaire pour aller de la configuration k à la configuration finale. A chaque itération, on enregistre la coût minimal de toutes les valeurs qui dépassent la valeur seuil. Ce coût minimal consitute le seuil pour l'itération suivante.

5.1 Questions préliminaires (Bonus : 2 points)

- Donnez une interprétation succinte de l'horizon défini à la partie 2.
- Estimez l'espace mémoire nécéssaire pour l'algorithme IDA*. Comparez le avec le résultat de la même question à la partie 1.

5.2 Implémentation (5 points)

Implémentez l'algorithme IDA* dans une méthode IDA en adaptant le pseudo-code ci-dessous pour qu'il fonctionne avec la classe *Taquin*.
Écrivez une méthode testant si la configuration initiale admet une solution, et le cas échéant, qui affiche un parcours optimal.

```
procedure ida_star(configuration):
  bound := horizon(configuration)
```

```
path := [configuration]
stack := liste vide
ajouter l'element [path, 0 , bound] a la liste stack
mini := 0
tant que mini est strictement inferieur a l'infini faire
  mini := infini
  tant que stack n'est pas vide faire
    p,g,b := stack[-1]
    retirer le dernier element de stack
    node = p[-1]
      f = g + horizon(node)
      si horizon(node) == 0, alors retourner (p,b)
      sinon si f est strictement superieur a b, alors
        si f est strictement inferieur a mini alors
          mini = f
      sinon
        si il y a un carreau en dessous de la case vide, alors
          deplacer ce carreau sur la case vide
          si la configuration obtenue (notee c) n'est pas dans p, alors
            ajouter [p + [c] , g+1, b] a la fin de stack
          reechanger le carreau deplace et la case vide
        si il y a un carreau en dessus de la case vide, alors
          deplacer ce carreau sur la case vide
          si la configuration obtenue (notee c) n'est pas dans p, alors
            ajouter [p + [c] , g+1, b] a la fin de stack
          reechanger le carreau deplace et la case vide
        si il y a un carreau a gauche de la case vide, alors
          deplacer ce carreau sur la case vide
          si la configuration obtenue (notee c) n'est pas dans p, alors
            ajouter [p + [c] , g+1, b] a la fin de stack
          reechanger le carreau deplace et la case vide
        si il y a un carreau a droite de la case vide, alors
          deplacer ce carreau sur la case vide
          si la configuration obtenue (notee c) n'est pas dans p, alors
            ajouter [p + [c] , g+1, b] a la fin de stack
          reechanger le carreau deplace et la case vide
  si mini est strictement inferieur a l'infini, alors
    ajouter l'element [path, 0 ,mini] a la fin de stack
retourner("pas de solution")
```

5.3 Test (Bonus : 2 point)

- Testez votre algorithme. Mesurer son temps d'execution pour $n = 2, 3$ et 4. (Prenez soin de vérifier que la configuration initiale admet bien une solution). Comparez aux résultats de la question 1.3.
- Expliquez pourquoi l'algorithme IDA* donne bien un chemin optimal (dans le cas où un tel chemin existe).

Grâce à l'heuristique, l'on parvient à trouver des solutions en moins d'une minute pour la plupart des configurations initiales et en quelques heures pour les configurations les plus difficiles.

6 Références

[1] D'après wikipédia : "Loyd affirma qu'il avait « *rendu le monde entier fou* » avec un taquin modifié. Dans la configuration proposée, les carreaux 14 et 15 étaient inversés, l'espace vide étant placé en bas à droite. Loyd prétendait avoir promis 1 000 USD à celui qui remettrait les carreaux dans l'ordre, mais la récompense n'aurait jamais été réclamée. La résolution de ce problème est impossible. D'une part, il faut en effet échanger les places des carreaux 14 et 15, et l'on peut montrer que cette opération nécessite un nombre impair de glissements. D'autre part, il faut que la case vide retrouve sa place initiale, opération qui, quant à elle, nécessite un nombre pair de glissements. Il est toutefois possible d'ordonner les chiffres de 1 à 15 si la case vide est initialement en haut à gauche." https://fr.wikipedia.org/wiki/Taquin

[2] Comment afficher une matrice avec les valeurs à l'intérieur : https://stackoverflow.com/questions/20998083/show-the-values-in-the-grid-using-matplotlib

```
plt.matshow(data,cmap = 'hsv')
for (x, y), value in np.ndenumerate(data):
    plt.text(y, x, f"{value:.2f}", va="center", ha="center")
```

[3] https://en.wikipedia.org/wiki/Iterative_deepening_A*

[4] Sujet adapté en partie de https://www.ens.psl.eu/sites/default/files/2017_Info_sujet_infoA.pdf

[5] https://stackoverflow.com/questions/1503072/how-to-check-if-permutations-have-equal-parity