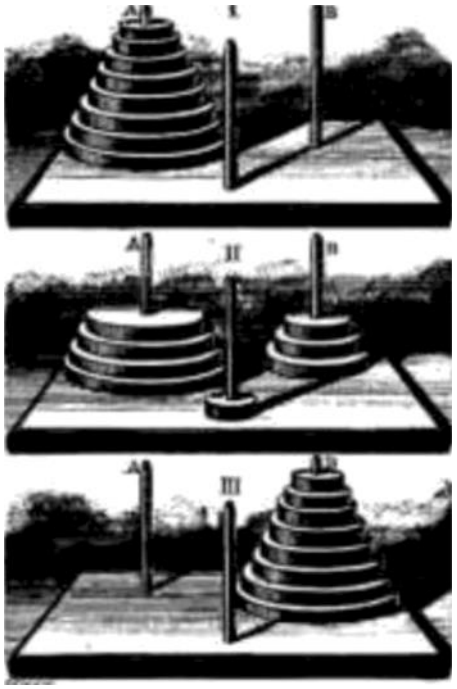
Rapport Montée en compétences

Système expert : application à la résolution des tours de Hanoï

(Date : 18/10/24)



Walid EL MEJJAD

2024/2025

SOMMAIRE

Table des matières

1 - Introduction :

Dans ce projet, l'objectif est de résoudre le problème des tours de Hanoï en utilisant un système expert basé sur des règles. Le but est d'améliorer l'efficacité du moteur d'inférence en optimisant le nombre de coups nécessaires pour déplacer les disques d'une tige à une autre, tout en respectant les règles du jeu.
Dans un premier temps, nous présenterons la classe et les méthodes développées pour réaliser ce projet. Ensuite, nous analyserons la performance de l'algorithme sur un cas standard avec 3 disques, en comparant le nombre de coups observés avec le nombre optimal théorique. Enfin, nous discuterons des améliorations possibles et proposerons une extension du programme pour traiter des cas avec un plus grand nombre de disques et de tiges.

2- Classe et méthodes :

Le moteur d'inférence pour résoudre le problème des tours de Hanoï est implémenté à l'aide de la classe **Jeu_Hanoi.** Cette classe nous permet de modéliser le jeu et de gérer les déplacements de palets entre les pics, tout en respectant les règles du jeu. Voici une description détaillée des principales méthodes de cette classe :

A) Principales méthodes

- **__init__(self, nb_pic, nb_palet):**

Il s'agit du constructeur de la classe. Initialement, l'idée était d'avoir deux constructeurs, comme en Java : un par défaut qui crée un jeu avec 3 pics et 3 palets, et un autre qui prend en paramètre le nombre de pics et de palets souhaités. Cependant, Python ne permet pas d'avoir plusieurs constructeurs pour une classe. Nous avons donc opté pour une seule méthode avec des paramètres personnalisables, ce qui laisse la porte ouverte à une future extension de la classe :

- nb_pic : Le nombre de pics sur lesquelles les palets seront déplacés.
- **nb_palet** : Le nombre de palet à déplacer.

Lors de l'initialisation, les disques sont placés en ordre décroissant sur le premier pic.

```
10          for i in range(nb_palet):
11              self.pic[0, i] = nb_palet - i
```

Les autres pics sont initialisés comme des listes vides. On a ainsi en attribut de la classe :

- pic : Un tableau à deux dimensions représentant les pics et les palets dessus. Le plus petit palet est représenté par le nombre 1, le deuxième plus grand par le nombre 2, etc.
- **nombre_palet** : Un tableau à une dimension qui représente le nombre total de palets sur chaque pic.

- **situation ()**

Cette méthode renvoie la configuration actuelle des pics sous forme de chaîne de caractères (string). Chaque pic est représenté par une liste d'entiers correspondant aux palets qu'elle contient. L'état général du jeu est alors retourné sous forme d'une liste imbriquée. La situation initiale pourrait se présenter comme suit :

```
Situation initiale :
[[3 2 1]
 [0 0 0]
 [0 0 0]]
```

Ici, le premier pic contient les palets 3, 2 et 1, tandis que les autres pics sont vides.

- **pic_vide(indice_pic)**

Cette méthode vérifie si le pic spécifié par **indice_pic** est vide. Elle se base sur le tableau **nombre_palet** : si l'élément à l'index donné vaut 0, cela signifie que le pic est vide.

```
self.nombre_palet[indice_pic] == 0
```

- **regle_jeu(indice_pic1, indice_pic2)**

Cette méthode vérifie si un déplacement de palet est autorisé entre une pic source **(indice_pic1)** et une pic destination **(indice_pic2).** Un déplacement est valide si :

- Le pic de destination est soit vide

```
(self.pic_vide(indice_pic2))
```

- Soit le pic de destination contient un disque plus grand que celui qu'on veutdéplacer

```
self.pic[indice_pic1, self.nombre_palet[indice_pic1] - 1] <
self.pic[indice_pic2, self.nombre_palet[indice_pic2] - 1]:
```

On peut voir que l'on cède de l'attribut **nombre_palet qui** nous permet d'avoir directement l'indice du plus petit palet sur le pic dans le tableau.

- **effectue_deplacement(indice_pic1, indice_pic2)**

Cette méthode gère le déplacement d'un palet du pic source **(indice_pic1)** source vers le pic destination **(indice_pic2)** sans prendre en compte les règles.

```
self.pic[indice_pic2, self.nombre_palet[indice_pic2]] = self.pic[indice_pic1, self.nombre_palet[indice_pic1] - 1]
self.pic[indice_pic1, self.nombre_palet[indice_pic1] - 1] = 0
```

Une fois le déplacement effectué, le tableau nombre_palet est mis à jour: le nombre de palets sur le pic source est décrémenté de 1, tandis que celui sur le pic destination est incrémenté.

```
34        self.nombre_palet[indice_pic1] = self.nombre_palet[indice_pic1] - 1
35        self.nombre_palet[indice_pic2] = self.nombre_palet[indice_pic2] + 1
```

- **situation_non_vue(indice_pic1, indice_pic2, situation_etudiee)**

Cette méthode vérifie si la configuration résultant d'un déplacement entre indice_pic1 et **indice_pic2** a déjà été rencontrée. Elle est essentielle pour éviter de répéter les mêmes configurations et se retrouver dans des boucles inutiles ou avoir une boucle infinie. Chaque configuration est représentée comme une liste de listes de palets, et elle est comparée aux situations déjà stockées dans **situation_etudiee.**

```
        self.effectue_deplacement(indice_pic1, indice_pic2)

        for situation in situation_etudiee:
            if np.array_equal(self.pic, situation):
                self.effectue_deplacement(indice_pic2, indice_pic1)
                return True

        self.effectue_deplacement(indice_pic2, indice_pic1)
        return False
```

Après avoir testé si la situation est déjà connue, il est essentiel de remettre les palets dans leur configuration d'origine afin de ne pas altérer l'état du jeu lors de la vérification.

B) Moteur d'inférence

Le moteur d'inférence est la méthode principale responsable de la résolution du problème des tours de Hanoï. Elle fonctionne en itérant sur les différentes configurations du jeu et en appliquant les règles de déplacement des palets de manière systématique. Voici un aperçu détaillé du fonctionnement de ce moteur d'inférence :

Cette méthode fait appel à plusieurs sous-méthodes pour choisir, à chaque étape, la règle à appliquer, en vérifiant si le déplacement est valide selon les règles du jeu. Une fois la règle choisie, elle simule l'application de celle-ci et vérifie si la situation résultante a déjà été rencontrée ou non.

- Le moteur commence par ajouter la configuration initiale à une liste des situations étudiées, ce qui permet de garder une trace des états du jeu déjà explorés. Ensuite, il entre dans une boucle qui se poursuit jusqu'à ce que tous les paletssoient déplacés vers le dernier pic.

```
def moteur_inference(self):
    situation_etudiee = []
    coups = 0

    # Ajout de la situation initiale
    situation_etudiee.append(np.copy(self.pic))

    # Tant que la situation finale n'est pas atteinte c'es
    while self.nombre_palet[2] != self.pic[0].size:
```

- À chaque itération, le moteur parcourt les différentes possibilités de déplacement et prend le premier qui valide les conditions. Si toutes ces conditions sont remplies, le moteur effectue le déplacement du palet, enregistre cette nouvelle situation et incrémente le compteur de coups.

```
elif not self.pic_vide(1) and self.regle_jeu(1, 0) and not self.situati
    self.effectue_deplacement(1, 0)
    coups += 1
    print(f"Déplacement de 1 à 0 (Coup {coups})")

# Afficher la situation actuelle
print(self.situation())
situation_etudiee.append(np.copy(self.pic))
```

En suivant les règles, on obtient une résolution du problème en 26 coups. Cela résulte à une solution qui n'est pas du tout optimale.

```
Déplacement de 0 à 1 (Coup 25)
[[0 0 0]
 [1 0 0]
 [3 2 0]]
Déplacement de 1 à 2 (Coup 26)
[[0 0 0]
 [0 0 0]
 [3 2 1]]
Problème résolu en 26 coups !
PS C:\Informatique-Fac\Master2_Informatic
```

Cependant, cette solution pourrait être davantage optimisée, car la solution idéale ne nécessite que 7 coups. Nous aborderons cette optimisation dans la partie suivante.

- Problème rencontré : En Python, la notion d'affectation se fait par référence à l'objet concerné. Ainsi, lorsque nous ajoutions une nouvelle situation à celles déjà étudiées, nous passions à chaque fois le pic qui était directement modifié à chaque itération dans le moteur. En conséquence, nous obtenions une liste avec les mêmes valeurs pour les situations étudiées, ce qui entraînait une boucle infinie. La solution a donc été de copier le tableau des pics avant de l'ajouter auxsituations étudiées.

```
situation_etudiee.append(np.copy(self.pic))
```

3- Optimisation du moteur d'inférence :

Le moteur d'inférence peut être optimisé en ajoutant d'autres règles ou en modifiant l'ordre des règles existantes. Par exemple, l'ajout de la règle de déplacement direct du pic 0 au pic 2 permet de réduire le nombre total de coups à 9. Bien que cette solution ne soit pas encore optimale, c'est déjà une amélioration significative.

```
# ajout de regle pour optimiser
if not self.pic_vide(0) and self.regle_jeu(0, 2) and not self.situation_non_vue(0, 2, situation_etudiee):
    self.effectue_deplacement(0, 2)
    coups += 1
    print(f"Déplacement de 0 à 2 (Coup {coups})")
```

```
 [3 2 0]]
Déplacement de 1 à 2 (Coup 9)
[[0 0 0]
 [0 0 0]
 [3 2 1]]
Problème résolu en 9 coups !
PS C:\Informatique-Fac\Master2 Informatiq
```

Pour parvenir à la solution idéale de 7 coups, il faudrait ajuster l'algorithme afin qu'il suive précisément la séquence minimale de déplacements des palets pour une résolution optimale. Cela pourrait être réalisé en :

- Priorisant les déplacements entre le pic 0 et le pic 2 dès que possible, puisque l'objectif est de transférer tous les palets directement au dernier pic.
- Utilisant une stratégie plus systématique, comme l'algorithme mathématique de résolution des tours de Hanoï, qui prescrit les mouvements dans un ordre spécifique pour atteindre l'optimalité (0 -> 2, 0 -> 1, 2 -> 1, 0 -> 2, 1 -> 0, 1 -> 2, 0 ->2 pour trois palets).

Ainsi, une meilleure planification des déplacements permettrait d'atteindre les 7 coups minimum.

4- Discussion sur l’extension du programme:

Pour étendre le moteur d'inférence, au lieu d'utiliser des conditions "if" rigides où l’on suit un ordre prédéfini de règles, on pourrait envisager de mettre en place une double boucle qui itère sur le nombre de pics. Cette boucle remplirait un tableau avec tous les mouvements possibles à chaque étape. À la fin de chaque itération, on choisirait le premier déplacement valide de la Iiste des mouvements possibles et l’on effectuerait ce déplacement. Ensuite, la boucle se répéterait jusqu'à ce que la solution soit trouvée.

Cette approche permettrait de rendre le moteur d'inférence plus flexible et d'adapter facilement le programme à un nombre variable de pics et de palets, tout en améliorant potentiellement l'efficacité de la résolution du problème.

5 – Conclusion

Le moteur d'inférence développé pour les Tours de Hanoï a été optimisé pour résoudre le cas classique avec 3 disques, atteignant le nombre minimal de 9 coups par rapport à la solution théorique (7 Coups). Nous avons concentré nos efforts sur l'optimisation de ce scénario en envisageant des améliorations futures, telles que l'ajout de règles pour anticiper les coups ou éviter les cycles. Toutefois, en raison de contraintes de temps, nous n'avons pas pu étendre le programme à des configurations avec plus de disques ou de tiges, ce qui reste une perspective d'évolution intéressante.