## Les règles du jeu de Nim

Le jeu de Nim est un jeu de stratégie à deux joueurs qui se joue avec plusieurs tas d'objets (par exemple, des allumettes, des pierres ou tout autre objet similaire). Les règles classiques sont les suivantes :

```
1.  Configuration initiale : Il y a plusieurs tas, chacun
contenant un certain nombre d'objets. Par exemple, trois tas
contenant respectivement 3, 4 et 5 objets.
2.  Tour de jeu : Les joueurs jouent à tour de rôle. À chaque
tour, un joueur doit retirer au moins un objet d'un seul tas. Il
peut retirer autant d'objets qu'il le souhaite, mais uniquement
d'un seul tas à la fois.
3.  Condition de victoire : Le joueur qui retire le dernier objet
du dernier tas gagne la partie. (Note : Il existe une variante
appelée "jeu de misère" où le joueur qui prend le dernier objet
perd. Dans notre cas, nous utiliserons la règle où prendre le
dernier objet signifie gagner.)
```

### Structure de données appropriée en Python

Pour représenter l'état du jeu de Nim en Python, une structure simple et efficace est une liste où chaque élément représente le nombre d'objets dans un tas. Par exemple :

```
etat_jeu = [3, 4, 5]  # Trois tas contenant respectivement 3, 4 et
5 objets.
```

Cette représentation permet de :

```
• Accéder facilement au nombre d'objets dans chaque tas.
• Modifier le nombre d'objets dans un tas spécifique après un
mouvement.
• Générer les états successeurs en appliquant les mouvements
possibles.
```

**Série d'exercices menant à l'implémentation de l'algorithme A* en Python pour le jeu de Nim**

Pour vous guider vers l'implémentation complète de l'algorithme A* appliqué au jeu de Nim, voici une série d'exercices progressifs.

*Exercice 1 : Compréhension et représentation du jeu de Nim*

a. Écrire une fonction pour afficher l'état actuel du jeu :

```
• Implémentez une fonction afficher_etat(etat_jeu) qui prend en
entrée la liste représentant l'état du jeu et affiche le nombre
d'objets dans chaque tas de manière conviviale.
```

b. Tester la fonction avec différents états :

```
• Testez votre fonction avec plusieurs configurations, par exemple
[3, 4, 5], [1, 0, 2], etc.
```

*Exercice 2 : Génération des mouvements légaux*

a. Écrire une fonction pour générer tous les mouvements possibles à partir d'un état donné :

```
• Implémentez une fonction generer_mouvements(etat_jeu) qui
retourne une liste de tous les états possibles après un mouvement
légal.
```

b. Vérifier le fonctionnement de la fonction :

```
• Pour un état donné, assurez-vous que la fonction génère
correctement tous les états successeurs légaux.
```

*Exercice 3 : Application d'un mouvement à l'état du jeu*

a. Écrire une fonction pour appliquer un mouvement :

```
• Créez une fonction appliquer_mouvement(etat_jeu, tas_index,
nb_objets) qui retourne un nouvel état après avoir retiré
nb_objets du tas à l'index tas_index.
```

b. Gérer les mouvements invalides :

```
• Assurez-vous que la fonction vérifie la validité du mouvement
(par exemple, ne pas retirer plus d'objets qu'il n'y en a dans le
tas).
```

*Exercice 4 : Implémentation de la fonction heuristique*

a. Définir une fonction heuristique admissible :

```
• Implémentez une fonction heuristique(etat_jeu) qui estime le
coût restant pour atteindre l'état final (victoire).
• Suggestion : Dans le jeu de Nim, une heuristique simple pourrait
être le nombre total d'objets restants.
```

b. Justifier l'admissibilité de l'heuristique :

```
• Expliquez pourquoi votre heuristique ne surestime jamais le coût
réel jusqu'à l'objectif.
```

*Exercice 5 : Définition de l'état final et du coût des mouvements*

a. Définir l'état final :

```
• Déterminez comment identifier si un état est un état gagnant
(par exemple, tous les tas sont vides).
```

b. Définir le coût des mouvements :

```
• Dans le cadre de l'algorithme A*, définissez le coût pour passer
d'un état à un autre. (Par exemple, chaque mouvement a un coût de
1.)
```

*Exercice 6 : Implémentation de l'algorithme A**

a. Créer la classe Noeud pour l'algorithme A* :

```
• Implémentez une classe Noeud avec les attributs nécessaires
(état, parent, coût g(n), heuristique h(n), etc.).
```

b. Implémenter l'algorithme A* :

```
• Créez une fonction algorithme_a_star(etat_initial) qui utilise
les fonctions précédentes pour trouver le chemin optimal vers la
victoire.
• Utilisez une structure telle qu'une file de priorité (par
exemple, le module heapq en Python) pour gérer les nœuds à
explorer.
```

*Exercice 7 : Gestion des états déjà visités*

a. Éviter de revisiter les mêmes états :

```
• Implémentez un mécanisme pour stocker les états déjà visités
(par exemple, un ensemble set de tuples immuables représentant les
états).
```

b. Mise à jour de l'algorithme A* :

```
• Modifiez votre fonction algorithme_a_star pour vérifier si un
état a déjà été exploré avant de l'ajouter à la file des nœuds à
explorer.
```

*Exercice 8 : Reconstruction du chemin optimal*

a. Stocker le parent de chaque état :

```
• Assurez-vous que chaque nœud stocke une référence à son parent.
```

b. Écrire une fonction pour reconstruire le chemin :

```
• Implémentez une fonction reconstruire_chemin(noeud_final) qui
retourne la séquence de mouvements depuis l'état initial jusqu'à
l'état final.
```

*Exercice 9 : Interface utilisateur pour jouer contre l'IA*

a. Permettre à l'utilisateur de jouer :

```
• Créez une boucle de jeu où l'utilisateur peut entrer ses
mouvements.
```

b. Intégrer l'IA utilisant l'algorithme A* :

```
• Après chaque mouvement de l'utilisateur, utilisez l'algorithme
A* pour déterminer le meilleur mouvement de l'IA.
• Note : Comme l'algorithme A* est conçu pour trouver le chemin
vers la victoire, il peut être nécessaire d'adapter l'heuristique
ou la stratégie pour jouer de manière interactive.
```

*Exercice 10 : Amélioration de l'algorithme*

a. Optimiser la fonction heuristique :

```
• Expérimentez avec différentes fonctions heuristiques pour
améliorer les performances (chercher sur internet la stratégie
gagnante).
```