LAB 6 - Building Games with AI

> **Auteur :** Badr TAJINI - Introduction à l'IA - ESIEE-IT - 2024/2025

Dans ce chapitre, nous allons apprendre à construire des jeux en utilisant une technique d'intelligence artificielle appelée **recherche combinatoire**. Dans sa forme la plus basique, elle peut être considérée comme une approche *brute-force*. Nous explorons chaque solution possible. Plus tard dans le chapitre, nous deviendrons plus intelligents et trouverons un moyen de raccourcir la recherche sans avoir à essayer toutes les possibilités. Nous apprendrons à utiliser des algorithmes de recherche pour élaborer efficacement des stratégies afin de gagner une série de jeux. Nous utiliserons ensuite ces algorithmes pour construire des bots intelligents pour différents jeux.

À la fin de ce chapitre, vous aurez une meilleure compréhension des concepts suivants :

- Algorithmes de recherche dans les jeux
- Recherche combinatoire
- L'algorithme Minimax
- L'élagage Alpha-Beta
- L'algorithme Negamax
- Construction d'un bot pour jouer à Last Coin Standing
- Construction d'un bot pour jouer au Tic-Tac-Toe
- Construction de deux bots pour jouer l'un contre l'autre à Puissance 4
- Construction de deux bots pour jouer l'un contre l'autre à Hexapawn

Utilisation des algorithmes de recherche dans les jeux

Les algorithmes de recherche sont couramment utilisés dans les jeux pour élaborer une stratégie. Les algorithmes recherchent à travers les mouvements de jeu possibles et choisissent le meilleur. Plusieurs paramètres doivent être considérés lors de la mise en œuvre de ces recherches : vitesse, précision, complexité, etc. Ces algorithmes considèrent tous les mouvements de jeu possibles à partir d'un état actuel et évaluent chaque mouvement possible pour déterminer le meilleur. L'objectif de ces algorithmes est de trouver le mouvement optimal qui mènera finalement à la victoire. De plus, chaque jeu a un ensemble différent de règles et de contraintes. Ces algorithmes tiennent compte de ces règles et contraintes lors de l'exploration des meilleurs mouvements.

Les jeux sans adversaires sont plus faciles à optimiser que les jeux avec adversaires. La jouabilité devient plus compliquée avec des jeux qui impliquent plusieurs joueurs. Considérons un jeu à

deux joueurs. Pour chaque mouvement effectué par un joueur pour essayer de gagner le jeu, l'adversaire effectuera un mouvement pour empêcher le joueur de gagner. Ainsi, lorsque un algorithme de recherche trouve l'ensemble optimal de mouvements à partir de l'état actuel, il ne peut pas simplement faire des mouvements sans tenir compte des contre-mouvements de l'adversaire. Cela signifie que les algorithmes de recherche doivent constamment être réévalués après chaque mouvement.

Discutons de la façon dont un ordinateur perçoit un jeu donné. Nous pouvons considérer un jeu comme un **arbre de recherche**. Chaque nœud de cet arbre représente un état futur. Par exemple, si vous jouez au **Tic-Tac-Toe** (Morpion), vous pouvez construire un arbre pour représenter tous les mouvements possibles. Nous commençons à la racine de l'arbre, qui est le point de départ du jeu. Ce nœud aura plusieurs enfants représentant divers mouvements possibles. Ces enfants, à leur tour, auront plus d'enfants représentant les états du jeu après d'autres mouvements de l'adversaire. Les nœuds terminaux de l'arbre représentent les derniers mouvements du jeu. Le jeu se terminera soit par une égalité, soit par la victoire de l'un des joueurs. Les algorithmes de recherche parcourent cet arbre pour prendre des décisions à chaque étape du jeu. Nous allons maintenant apprendre une variété de techniques de recherche sur la façon d'inclure une recherche combinatoire exhaustive pour nous aider à ne jamais perdre au Tic-Tac-Toe et résoudre une variété de nombreux autres problèmes.

Recherche combinatoire

Les algorithmes de recherche semblent résoudre le problème d'ajouter de l'intelligence aux jeux, mais il y a un inconvénient. Ces algorithmes utilisent un type de recherche appelé recherche exhaustive, également connue sous le nom de recherche *brute-force*. Elle explore essentiellement tout l'espace de recherche et teste chaque solution possible. Cela signifie que l'algorithme devra explorer toutes les solutions possibles avant d'obtenir la solution optimale.

À mesure que les jeux deviennent plus complexes, la recherche brute-force peut ne pas être la meilleure approche, car le nombre de possibilités devient énorme. La recherche devient rapidement informatiquement intractable. Pour résoudre ce problème, on peut utiliser la recherche combinatoire. La recherche combinatoire fait référence à un domaine d'étude où les algorithmes de recherche explorent efficacement l'espace des solutions en utilisant des heuristiques pour réduire la taille de l'espace de recherche. Cela est utile dans des jeux tels que les Échecs ou le Go.

La recherche combinatoire fonctionne efficacement en utilisant des stratégies d'élagage. Ces stratégies évitent de tester toutes les solutions possibles en éliminant celles qui sont manifestement incorrectes. Cela permet d'économiser du temps et des efforts. Maintenant que nous avons appris la recherche combinatoire exhaustive et ses limitations, nous allons commencer à explorer des moyens de prendre des raccourcis, d'**élaguer** l'arbre de recherche et

d'éviter d'avoir à tester chaque combinaison. Dans les sections suivantes, nous explorerons quelques algorithmes spécifiques qui nous permettent d'effectuer une recherche combinatoire.

L'algorithme Minimax

Maintenant que nous avons brièvement discuté de la recherche combinatoire en général, parlons des heuristiques employées par les algorithmes de recherche combinatoire. Ces heuristiques sont utilisées pour accélérer la stratégie de recherche et l'algorithme Minimax est une telle stratégie utilisée par la recherche combinatoire. Lorsque deux joueurs s'affrontent, leurs objectifs sont diamétralement opposés. Chaque joueur essaie de gagner. Donc, chaque côté doit prédire ce que l'autre joueur va faire pour gagner le jeu. En gardant cela à l'esprit, Minimax essaie d'atteindre cet objectif par la stratégie. Il essaiera de minimiser la fonction que l'adversaire essaie de maximiser.

Comme discuté précédemment, la brute-force ne fonctionne que dans des jeux simples avec un petit nombre de mouvements possibles. Dans des cas plus compliqués, l'ordinateur ne peut pas parcourir tous les états possibles pour trouver le gameplay optimal. Dans ce cas, l'ordinateur peut essayer de calculer le mouvement optimal basé sur l'état actuel en utilisant une heuristique. L'ordinateur construit un arbre et commence par le bas. Il évalue quels mouvements pourraient bénéficier à son adversaire. L'algorithme sait quels mouvements l'adversaire va faire en se basant sur le principe que l'adversaire fera les mouvements qui leur seraient les plus bénéfiques et, par conséquent, les moins bénéfiques pour l'ordinateur. Ce résultat est l'un des nœuds terminaux de l'arbre et l'ordinateur utilise cette position pour remonter en arrière. Chaque option disponible pour l'ordinateur peut être attribuée à une valeur, puis il peut choisir la valeur la plus élevée pour prendre une action.

Élagage Alpha-Beta

La recherche Minimax est une stratégie efficace, mais elle explore encore des parties de l'arbre qui sont non pertinentes. Lorsqu'un indicateur sur un nœud est trouvé montrant qu'une solution n'existe pas dans ce sous-arbre, il n'est pas nécessaire d'évaluer ce sous-arbre. Mais la recherche Minimax est trop conservatrice, donc elle finit par explorer certains de ces sous-arbres.

L'algorithme Alpha-Beta est plus intelligent et évite de rechercher des parties de l'arbre qu'il découvre ne contiendront pas la solution. Ce processus s'appelle **l'élagage** et l'élagage Alpha-Beta est un type de stratégie utilisée pour éviter de rechercher des parties de l'arbre qui ne contiennent pas la solution.

Les paramètres Alpha et Beta dans l'élagage Alpha-Beta font référence aux deux bornes utilisées pendant le calcul. Ces paramètres font référence aux valeurs qui restreignent l'ensemble des

solutions possibles. Cela est basé sur la section de l'arbre qui a déjà été explorée. Alpha est la borne inférieure maximale sur le nombre de solutions possibles et Beta est la borne supérieure minimale sur le nombre de solutions possibles.

Comme discuté précédemment, chaque nœud peut être attribué à une valeur basée sur l'état actuel. Lorsque l'algorithme considère un nouveau nœud comme un chemin potentiel vers la solution, il peut déterminer si l'estimation actuelle de la valeur du nœud se situe entre Alpha et Beta. C'est ainsi qu'il élague la recherche.

L'algorithme Negamax

L'algorithme **Negamax** est une variante de Minimax qui est fréquemment utilisée dans les implémentations réelles. Un jeu à deux joueurs est généralement un jeu à somme nulle, ce qui signifie que la perte d'un joueur est égale au gain de l'autre joueur et vice versa. Negamax utilise cette propriété de manière intensive pour élaborer une stratégie afin d'augmenter ses chances de gagner le jeu.

En termes de jeu, la valeur d'une position donnée pour le premier joueur est la négation de la valeur pour le second joueur. Chaque joueur cherche un mouvement qui maximisera les dégâts infligés à l'adversaire. La valeur résultant du mouvement devrait être telle que l'adversaire obtienne la valeur la plus basse possible. Cela fonctionne de manière fluide dans les deux sens, ce qui signifie qu'une seule méthode peut être utilisée pour évaluer les positions. C'est là qu'il a un avantage par rapport à Minimax en termes de simplicité. Minimax nécessite que le premier joueur sélectionne le mouvement avec la valeur maximale, tandis que le second joueur doit sélectionner un mouvement avec la valeur minimale. L'élagage Alpha-Beta est également utilisé ici. Maintenant que nous avons examiné quelques-uns des algorithmes de recherche combinatoire les plus populaires, installons une bibliothèque afin que nous puissions construire une IA et voir ces algorithmes en action.

Installation de la bibliothèque easyAI

Nous allons utiliser une bibliothèque appelée `easyAI` dans ce chapitre. C'est un cadre d'intelligence artificielle qui fournit toute la fonctionnalité nécessaire pour construire des jeux à deux joueurs. Vous pouvez en savoir plus ici :

http://zulko.github.io/easyAI

Installez-la en exécutant la commande suivante :

```
pip install easyAI
```

Certains fichiers doivent être accessibles pour utiliser certaines routines préconstruites. Pour plus de facilité, le code fourni avec ce livre contient un dossier appelé `easyAI`. Assurez-vous de placer ce dossier dans le même répertoire que vos fichiers de code. Ce dossier est essentiellement un sous-ensemble du dépôt GitHub `easyAI` disponible ici :

https://github.com/Zulko/easyAI

Vous pouvez parcourir le code source pour vous familiariser davantage avec.

Construction d'un bot pour jouer à Last Coin Standing

Dans ce jeu, il y a un tas de pièces et chaque joueur prend à tour de rôle un certain nombre de pièces du tas. Il y a une limite inférieure et supérieure sur le nombre de pièces qui peuvent être prises du tas. L'objectif du jeu est d'éviter de prendre la dernière pièce du tas. Cette recette est une variante de la recette du Game of Bones donnée dans la bibliothèque `easyAI`. Voyons comment construire un jeu où l'ordinateur peut jouer contre l'utilisateur.

Créez un nouveau fichier Python et importez les packages suivants :

```
from easyAI import TwoPlayersGame, id_solve, Human_Player, AI_Player
from easyAI.AI import TT
import numpy as np
import math
import random
```

Créez une classe pour gérer toutes les opérations du jeu. Le code hérite de la classe de base `TwoPlayersGame` disponible dans la bibliothèque `easyAI`. Il y a quelques paramètres qui doivent être définis pour que le code fonctionne correctement. Le premier est la variable `players`. L'objet `player` sera discuté plus tard. Créez la classe en utilisant le code suivant :

```
class LastCoinStanding(TwoPlayersGame):
    def __init__(self, players):
        # Définir les joueurs. Paramètre nécessaire.
        self.players = players

        # Définir qui commence le jeu. Paramètre nécessaire.
        self.nplayer = 1

        # Nombre total de pièces dans le tas
        self.num_coins = 25

        # Définir le nombre maximal de pièces par mouvement
        self.max_coins = 4
```

Définissez tous les mouvements possibles. Dans ce cas, les joueurs peuvent prendre 1, 2, 3 ou 4 pièces à chaque mouvement :

```
    # Définir les mouvements possibles
    def possible_moves(self):
        return [str(x) for x in range(1, self.max_coins + 1)]
```

Définissez une méthode pour retirer les pièces et suivre le nombre de pièces restantes dans le tas :

```
    # Retirer les pièces
    def make_move(self, move):
        self.num_coins -= int(move)
```

Vérifiez si quelqu'un a gagné le jeu en vérifiant le nombre de pièces restantes :

```
    # Quelqu'un a-t-il pris la dernière pièce ?
    def win(self):
        return self.num_coins <= 0
```

Arrêtez le jeu après que quelqu'un a gagné :

```
    # Arrêter le jeu lorsqu'un joueur gagne
    def is_over(self):
        return self.win()
```

Calculez le score basé sur la méthode `win`. Il est nécessaire de définir cette méthode :

```
    # Calculer le score
    def scoring(self):
        return 100 if self.win() else 0
```

Définissez une méthode pour afficher le statut actuel du tas :

```
    # Afficher le nombre de pièces restantes dans le tas
    def show(self):
        print(self.num_coins, 'pièces restantes dans le tas')
```

Définissez la fonction `main` et commencez par définir la table de transposition. Les tables de transposition sont utilisées dans les jeux pour stocker les positions et les mouvements afin

d'accélérer l'algorithme.

```
if __name__ == "__main__":
    # Définir la table de transposition
    tt = TT()

    # Définir la méthode ttentry pour obtenir le nombre de pièces
    LastCoinStanding.ttentry = lambda self: self.num_coins

    # Résoudre le jeu
    result, depth, move = id_solve(LastCoinStanding,
                                   range(2, 20),
                                   win_score=100,
                                   tt=tt)
    print(result, depth, move)

    # Démarrer le jeu
    game = LastCoinStanding([AI_Player(tt), Human_Player()])
    game.play()
```

Le code complet est disponible dans le fichier `coins.py`. C'est un programme interactif, il attend donc une entrée de l'utilisateur. Si vous exécutez le code, vous jouerez contre l'ordinateur. Votre objectif est de forcer l'ordinateur à prendre la dernière pièce afin de gagner le jeu. Si vous exécutez le code, vous verrez initialement la sortie suivante :

```
d:2, a:0, m:1
d:3, a:0, m:1
d:4, a:0, m:1
d:5, a:0, m:1
d:6, a:0, m:1
d:7, a:0, m:1
d:8, a:0, m:1
d:9, a:0, m:1
d:10, a:100, m:4
1 10 4
25 coins left in the pile

Move #1: player 1 plays 4 :
21 coins left in the pile

Player 2 what do you play ? 1

Move #2: player 2 plays 1 :
20 coins left in the pile

Move #3: player 1 plays 4 :
16 coins left in the pile
```

Figure 1 : Sortie initiale du jeu Last Coin Standing

Si vous faites défiler, vous verrez la sortie suivante vers la fin :

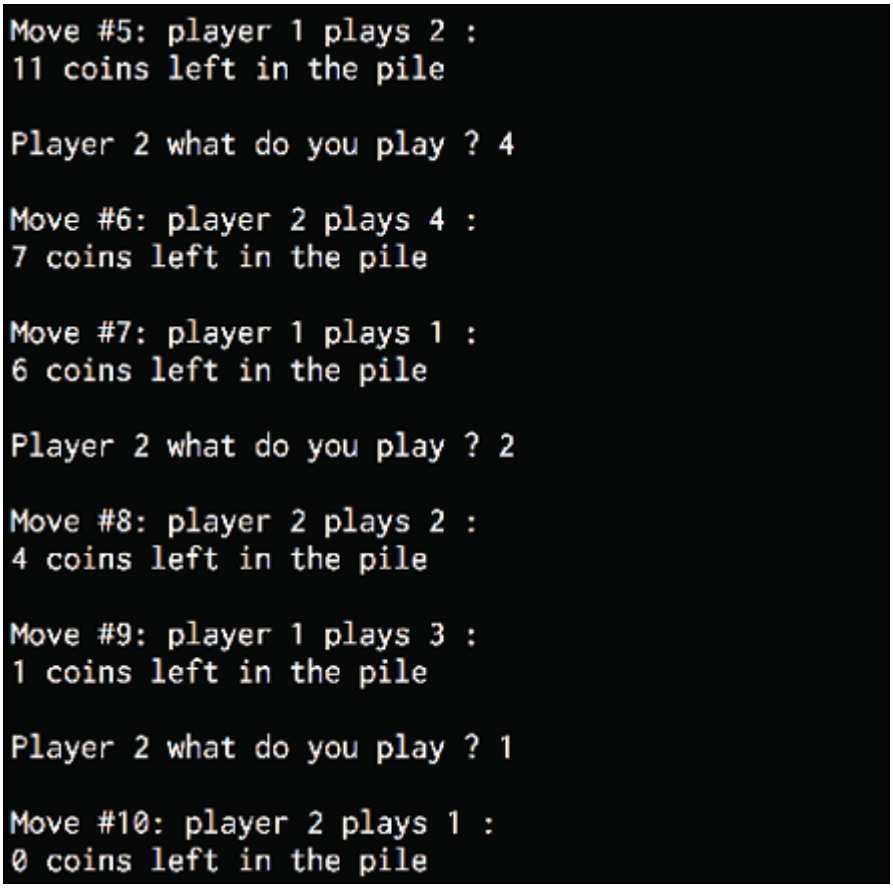

Figure 2 : Sortie finale du jeu Last Coin Standing

Comme nous pouvons le voir, l'ordinateur gagne le jeu parce que l'utilisateur a pris la dernière pièce.

Passons maintenant à la construction d'un bot pour un autre jeu : le Tic-Tac-Toe.

Construction d'un bot pour jouer au Tic-Tac-Toe

Le Tic-Tac-Toe (Morpion) est peut-être l'un des jeux les plus célèbres au monde. Voyons comment construire un jeu où l'ordinateur peut jouer contre l'utilisateur. Il s'agit d'une légère variante de la recette du Tic-Tac-Toe donnée dans la bibliothèque `easyAI`.

Créez un nouveau fichier Python et importez les packages suivants :

```
from easyAI import TwoPlayersGame, AI_Player, Negamax
from easyAI.Player import Human_Player
```

Définissez une classe qui contient toutes les méthodes nécessaires pour jouer au jeu. Commencez par définir les joueurs et qui commence le jeu :

```
class GameController(TwoPlayersGame):
    def __init__(self, players):
        # Définir les joueurs
        self.players = players

        # Définir qui commence le jeu
        self.nplayer = 1

        # Définir le plateau de jeu (3x3)
        self.board = [0] * 9
```

Définissez une méthode pour calculer tous les mouvements possibles :

```
    # Définir les mouvements possibles
    def possible_moves(self):
        return [a + 1 for a, b in enumerate(self.board) if b == 0]
```

Définissez une méthode pour mettre à jour le plateau après avoir effectué un mouvement :

```
    # Effectuer un mouvement
    def make_move(self, move):
        self.board[int(move) - 1] = self.nplayer
```

Définissez une méthode pour vérifier si quelqu'un a perdu le jeu en vérifiant si quelqu'un a trois en ligne :

```
    # L'adversaire a-t-il trois en ligne ?
    def loss_condition(self):
        possible_combinations = [
            [1,2,3], [4,5,6], [7,8,9],  # Lignes
            [1,4,7], [2,5,8], [3,6,9],  # Colonnes
            [1,5,9], [3,5,7]              # Diagonales
        ]
        return any([
            all([(self.board[i-1] == self.nopponent) for i in combination])
            for combination in possible_combinations
        ])
```

Vérifiez si le jeu est terminé en utilisant la méthode `loss_condition` :

```
    # Vérifier si le jeu est terminé
    def is_over(self):
        return (self.possible_moves() == []) or self.loss_condition()
```

Calculez le score en utilisant la méthode `loss_condition` :

```
    # Calculer le score
    def scoring(self):
        return -100 if self.loss_condition() else 0
```

Définissez une méthode pour afficher l'état actuel du plateau :

```
    # Afficher l'état actuel
    def show(self):
        print('\n'+'\n'.join([
            ' '.join([['.', 'O', 'X'][self.board[3*j + i]] for i in range(3)])
            for j in range(3)
        ]))
```

Définissez la fonction `main` et commencez par définir l'algorithme. Negamax sera utilisé comme algorithme IA pour ce jeu. Le nombre de mouvements que l'algorithme doit penser à l'avance peut être spécifié à l'avance. Dans ce cas, choisissons `7` :

```
if __name__ == "__main__":
    # Définir l'algorithme
    algorithm = Negamax(7)

    # Démarrer le jeu
    game = GameController([Human_Player(), AI_Player(algorithm)])
    game.play()
```

Le code complet est disponible dans le fichier `tic_tac_toe.py`. C'est un jeu interactif où vous jouez contre l'ordinateur. Si vous exécutez le code, vous verrez initialement la sortie suivante :

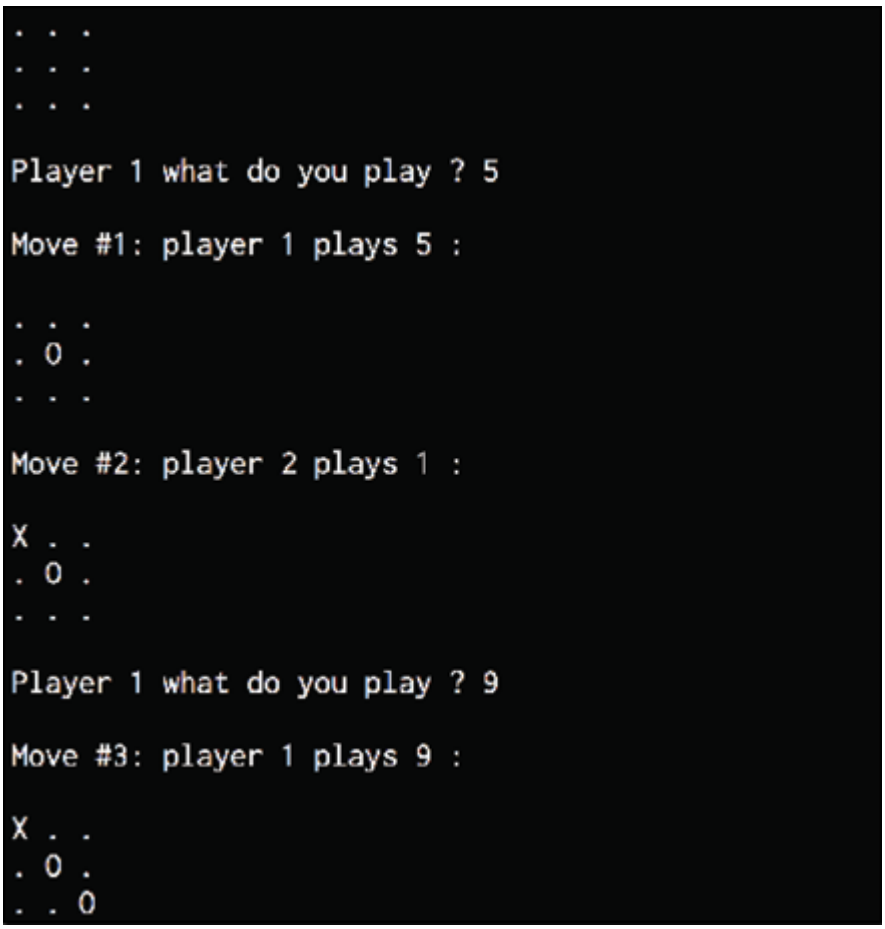

Figure 3 : Sortie initiale du jeu Tic-Tac-Toe

Si vous faites défiler, vous verrez la sortie suivante imprimée :

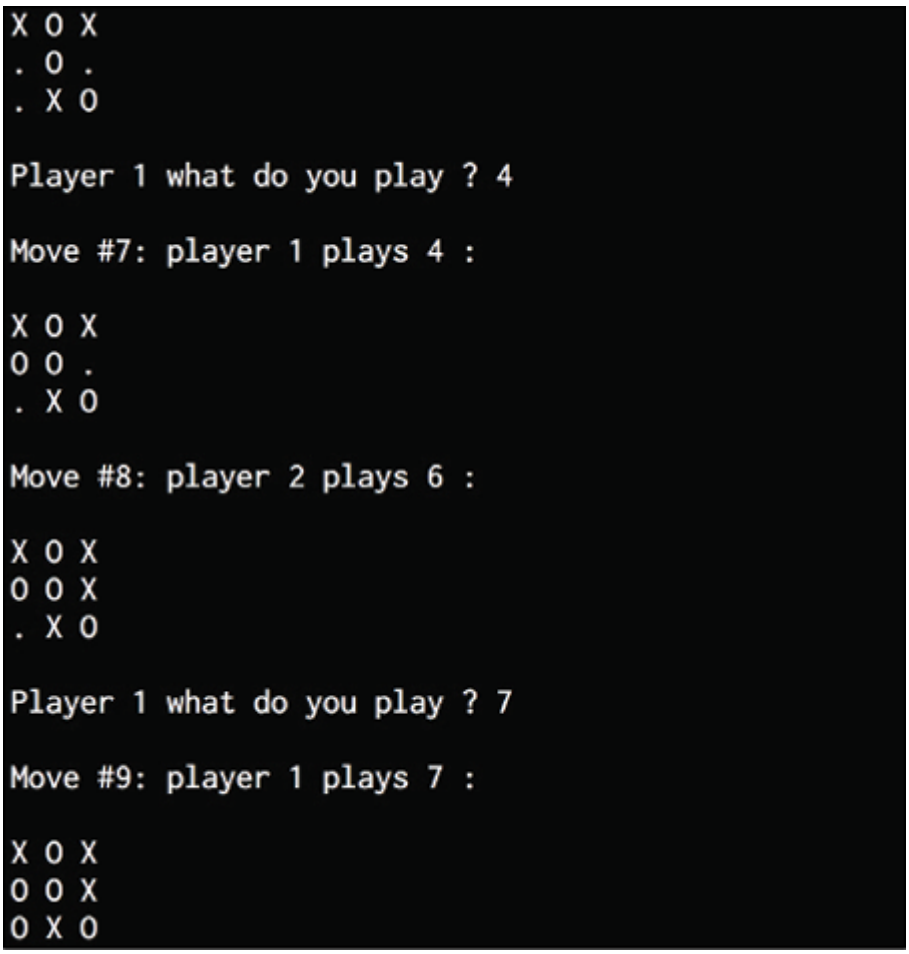

Figure 4 : Sortie finale du jeu Tic-Tac-Toe

Comme nous pouvons le voir, le jeu se termine par une égalité.

Passons à présent à la construction de deux bots pour jouer l'un contre l'autre au Puissance 4.

Construction de deux bots pour jouer au Puissance 4 contre eux-mêmes

Le Puissance 4 est un jeu à deux joueurs populaire vendu sous la marque Milton Bradley. Il est également connu sous d'autres noms tels que Quatre en ligne ou Quatre en haut. Dans ce jeu, les joueurs prennent à tour de rôle des disques qu'ils laissent tomber dans une grille verticale composée de six rangées et sept colonnes. L'objectif est d'obtenir quatre disques alignés. Il s'agit d'une variante de la recette du Puissance 4 donnée dans la bibliothèque `easyAI`. Voyons comment le construire. Dans cette recette, au lieu de jouer contre l'ordinateur, nous créerons deux bots qui joueront l'un contre l'autre. Un algorithme différent sera utilisé pour chacun afin de voir lequel gagne.

Créez un nouveau fichier Python et importez les packages suivants :

```
import numpy as np
from easyAI import TwoPlayersGame, AI_Player, Negamax, SSS
from easyAI.Player import Human_Player
```

Définissez une classe qui contient toutes les méthodes nécessaires pour jouer au jeu :

```
class GameController(TwoPlayersGame):
    def __init__(self, players, board=None):
        # Définir les joueurs
        self.players = players

        # Définir la configuration du plateau
        self.board = board if (board != None) else (
            np.array([[0 for i in range(7)] for j in range(6)]))

        # Définir qui commence le jeu
        self.nplayer = 1

        # Définir les positions (directions pour vérifier les alignements)
        self.pos_dir = np.array([
            [[i, 0], [0, 1]] for i in range(6)
        ] + [
            [[0, i], [1, 0]] for i in range(7)
        ] + [
            [[i, 0], [1, 1]] for i in range(1, 3)
        ] + [
            [[0, i], [1, 1]] for i in range(4)
        ] + [
            [[i, 6], [1, -1]] for i in range(1, 3)
        ] + [
            [[0, i], [1, -1]] for i in range(3, 7)
        ])
```

Définissez une méthode pour obtenir tous les mouvements possibles :

```
    # Définir les mouvements possibles
    def possible_moves(self):
        return [i for i in range(7) if (self.board[:, i].min() == 0)]
```

Définissez une méthode pour contrôler comment le robot fait avancer :

```
    # Définir comment faire un mouvement
    def make_move(self, column):
        line = np.argmin(self.board[:, column] != 0)
        self.board[line, column] = self.nplayer
```

Définissez une méthode pour afficher l'état actuel :

```
    # Afficher l'état actuel
    def show(self):
        print('\n' + '\n'.join([
            '0 1 2 3 4 5 6', 13 * '-'
        ] + [
            ' '.join([
                ['.', 'O', 'X'][self.board[5 - j][i]] for i in range(7)
            ]) for j in range(6)
        ]))
```

Définissez une méthode pour calculer ce qu'est une condition de perte. Si un joueur obtient quatre en ligne, ce joueur gagne le jeu :

```
    # Définir ce qu'est une condition de perte
    def loss_condition(self):
        for pos, direction in self.pos_dir:
            streak = 0
            while (0 <= pos[0] <= 5) and (0 <= pos[1] <= 6):
                if self.board[pos[0], pos[1]] == self.nopponent:
                    streak += 1
                    if streak == 4:
                        return True
                else:
                    streak = 0
                pos += direction
        return False
```

Vérifiez si le jeu est terminé en utilisant la méthode `loss_condition` :

```
    # Vérifier si le jeu est terminé
    def is_over(self):
        return self.loss_condition()
```

Calculez le score en utilisant la méthode `loss_condition` :

```
    # Calculer le score
    def scoring(self):
        return -100 if self.loss_condition() else 0
```

Définissez la fonction `main` et commencez par définir l'algorithme. Ensuite, les deux algorithmes joueront l'un contre l'autre. Negamax sera utilisé pour le premier joueur IA et l'algorithme **SSS*** pour le second joueur IA. SSS* est un algorithme de recherche qui effectue une recherche dans l'espace d'état en parcourant l'arbre de manière best-first. Les deux méthodes prennent en argument le nombre de tours à penser à l'avance. Dans ce cas, utilisons `5` pour les deux :

```
if __name__ == '__main__':
    # Définir les algorithmes qui seront utilisés
    algo_neg = Negamax(5)
    algo_sss = SSS(5)

    # Démarrer le jeu
    game = GameController([AI_Player(algo_neg), AI_Player(algo_sss)])
    game.play()

    # Imprimer le résultat
    if game.loss_condition():
        print('\nLe joueur', game.nopponent, 'gagne.')
    else:
        print("\nC'est une égalité.")
```

Le code complet est disponible dans le fichier `connect_four.py`. Ce n'est pas un jeu interactif. Le code oppose un algorithme à un autre. L'algorithme Negamax est le joueur un et l'algorithme SSS* est le joueur deux.

Si vous exécutez le code, vous verrez initialement la sortie suivante :

```
0 1 2 3 4 5 6
-------------
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .

Move #1: player 1 plays 0 :

0 1 2 3 4 5 6
-------------
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
O . . . . . .

Move #2: player 2 plays 0 :

0 1 2 3 4 5 6
-------------
. . . . . . .
```

Figure 5 : Sortie initiale du jeu Puissance 4

Si vous faites défiler, vous verrez la sortie suivante vers la fin :

```
O O O X O O .

Move #35: player 1 plays 6 :

0 1 2 3 4 5 6
-------------
X X O O X . .
O O X X O . .
X X O O X X .
O O X X O O .
X X O X X X .
O O O X O O O

Move #36: player 2 plays 6 :

0 1 2 3 4 5 6
-------------
X X O O X . .
O O X X O . .
X X O O X X .
O O X X O O .
X X O X X X X
O O O X O O O

Player 2 wins.
```

Figure 6 : Sortie finale du jeu Puissance 4

Comme nous pouvons le voir, le joueur deux gagne le jeu.

Essayons maintenant un autre jeu : Hexapawn.

Construction de deux bots pour jouer à Hexapawn contre eux-mêmes

Hexapawn est un jeu à deux joueurs commençant avec un échiquier de taille *N*__×__*M*. Des pions existent de chaque côté de l'échiquier et l'objectif est d'avancer un pion jusqu'au bout de l'échiquier opposé. Les règles des pions en échecs s'appliquent. Il s'agit d'une variante de la recette de Hexapawn donnée dans la bibliothèque `easyAI`. Deux bots seront créés et opposés l'un à l'autre. Voyons comment créer le code.

Créez un nouveau fichier Python et importez les packages suivants :

```
from easyAI import TwoPlayersGame, AI_Player, Human_Player, Negamax
```

Définissez une classe qui contient toutes les méthodes nécessaires pour contrôler le jeu. Commencez par définir le nombre de pions de chaque côté et la longueur de l'échiquier. Créez une liste de tuples contenant les positions :

```
class GameController(TwoPlayersGame):
    def __init__(self, players, size=(4, 4)):
        self.size = size
        num_pawns, len_board = size
        p = [[(i, j) for j in range(len_board)] for i in [0, num_pawns - 1]]

        for i, d, goal, pawns in [
            (0, 1, num_pawns - 1, p[0]),
            (1, -1, 0, p[1])
        ]:
            players[i].direction = d
            players[i].goal_line = goal
            players[i].pawns = pawns

        # Définir les joueurs
        self.players = players

        # Définir qui commence le jeu
        self.nplayer = 1

        # Définir les alphabets pour identifier les positions
        self.alphabets = 'ABCDEFGHIJ'

        # Convertir les chaînes en tuples
        self.to_tuple = lambda s: (self.alphabets.index(s[0]), int(s[1:]) - 1)

        # Convertir les tuples en chaînes
        self.to_string = lambda move: ' '.join([
            self.alphabets[move[i][0]] + str(move[i][1] + 1) for i in (0, 1)
        ])
```

Définissez une méthode pour calculer les mouvements possibles :

```
    # Définir les mouvements possibles
    def possible_moves(self):
        moves = []
        opponent_pawns = self.opponent.pawns
        d = self.player.direction

        for i, j in self.player.pawns:
            if (i + d, j) not in opponent_pawns:
                moves.append(((i, j), (i + d, j)))
            if (i + d, j + 1) in opponent_pawns:
                moves.append(((i, j), (i + d, j + 1)))
            if (i + d, j - 1) in opponent_pawns:
                moves.append(((i, j), (i + d, j - 1)))

        return list(map(self.to_string, moves))
```

Définissez une méthode pour effectuer un mouvement et mettre à jour les pions en conséquence :

```
    # Définir comment effectuer un mouvement
    def make_move(self, move):
        move = list(map(self.to_tuple, move.split(' ')))
        ind = self.player.pawns.index(move[0])
        self.player.pawns[ind] = move[1]

        if move[1] in self.opponent.pawns:
            self.opponent.pawns.remove(move[1])
```

Définissez une méthode pour vérifier la condition de perte. Si un joueur atteint la ligne de but ou ne peut plus effectuer de mouvements, l'autre joueur gagne :

```
    # Définir ce qu'est une condition de perte
    def loss_condition(self):
        return (
            any([i == self.opponent.goal_line for i, j in self.opponent.pawns])
            or (self.possible_moves() == [])
        )
```

Vérifiez si le jeu est terminé en utilisant la méthode `loss_condition` :

```
    # Vérifier si le jeu est terminé
    def is_over(self):
        return self.loss_condition()
```

Définissez une méthode pour afficher l'état actuel du plateau :

```
    # Afficher l'état actuel
    def show(self):
        f = lambda x: '1' if x in self.players[0].pawns else (
            '2' if x in self.players[1].pawns else '.'
        )
        print("\n".join([
            " ".join([f((i, j)) for j in range(self.size[1])])
            for i in range(self.size[0])
        ]))
```

Définissez la fonction `main` et commencez par définir la fonction de scoring :

```
if __name__ == '__main__':
    # Calculer le score
    scoring = lambda game: -100 if game.loss_condition() else 0

    # Définir l'algorithme
    algorithm = Negamax(12, scoring)

    # Démarrer le jeu
    game = GameController([AI_Player(algorithm), AI_Player(algorithm)])
    game.play()

    # Imprimer le résultat
    if game.loss_condition():
        print('\nLe joueur', game.nopponent, 'gagne après', game.nmove, 'tours')
    else:
        print("\nC'est une égalité.")
```

Le code complet est disponible dans le fichier `hexapawn.py`. Ce n'est pas un jeu interactif. Deux bots sont créés et opposés l'un à l'autre. Si vous exécutez le code, vous verrez initialement la sortie suivante :

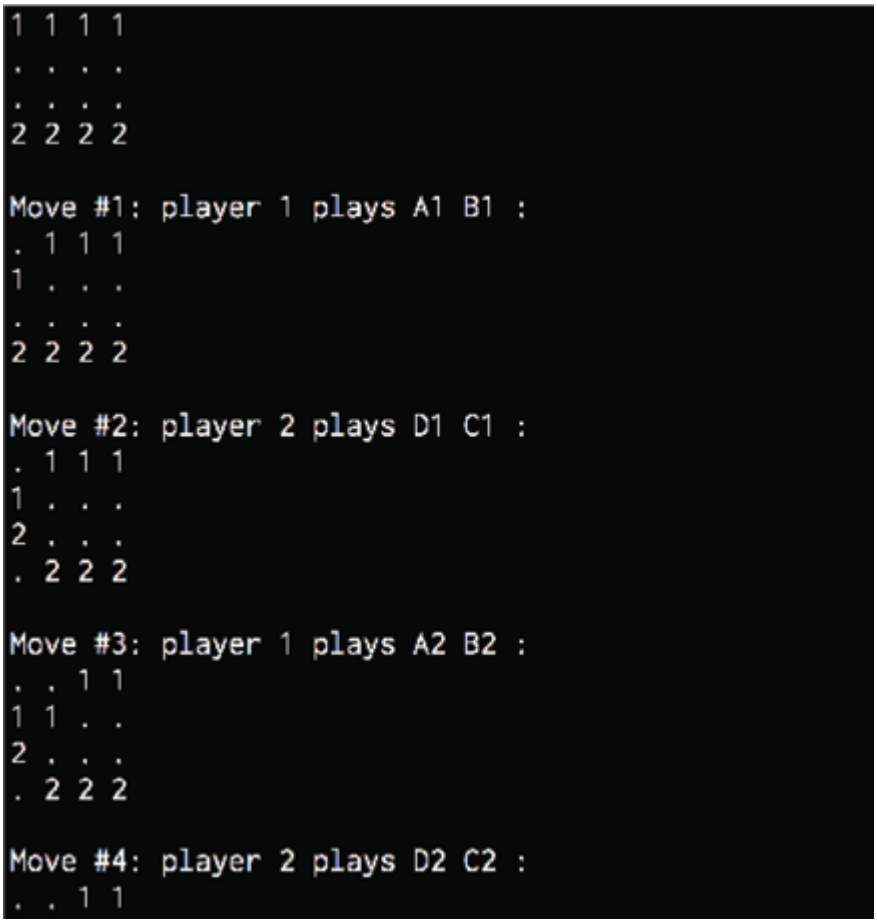

Figure 7 : Sortie initiale du jeu Hexapawn

Si vous faites défiler, vous verrez la sortie suivante vers la fin :

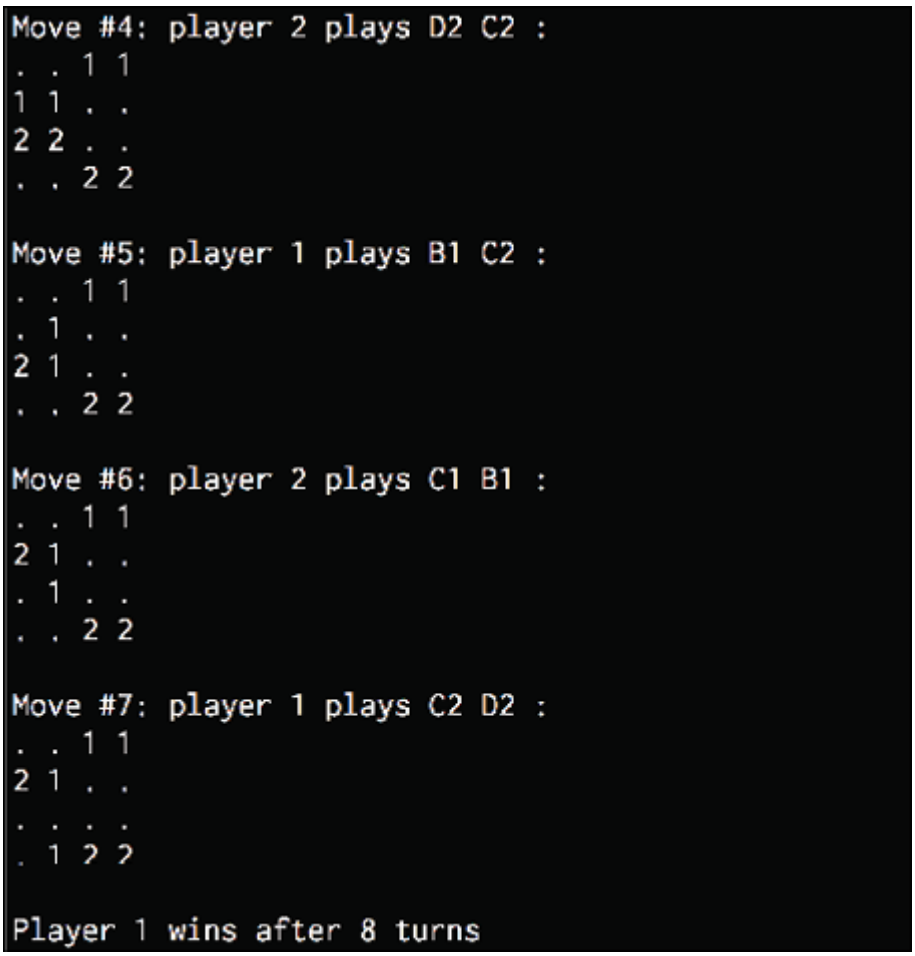

Figure 8 : Sortie finale du jeu Hexapawn

Comme nous pouvons le voir, le joueur un gagne le jeu.

Résumé

Dans ce chapitre, nous avons discuté de la manière de construire des jeux en utilisant un type spécial de technique d'intelligence artificielle appelée recherche combinatoire. Nous avons appris à utiliser ces types d'algorithmes de recherche pour élaborer efficacement des stratégies afin de gagner des jeux. Ces algorithmes peuvent être utilisés pour construire des machines de jeu pour des jeux plus compliqués ainsi que pour résoudre une large variété de problèmes. Nous avons parlé de la recherche combinatoire et de la manière dont elle peut être utilisée pour accélérer le processus de recherche. Nous avons appris les algorithmes Minimax et Alpha-Beta pruning. Nous avons appris comment l'algorithme Negamax est utilisé en pratique. Nous avons ensuite utilisé ces algorithmes pour construire des bots pour jouer à Last Coin Standing et au Tic-Tac-Toe.

Nous avons appris à construire deux bots pour jouer l'un contre l'autre à Puissance 4 et Hexapawn. Dans le prochain chapitre, nous apprendrons la reconnaissance vocale et construirons un système pour reconnaître automatiquement des mots parlés.

Références

1. A. Fukunaga et A. Stechert. Évolution de modèles prédictifs non linéaires pour la compression d'image sans perte avec la programmation génétique. Dans J. R. Koza, et al., éditeurs, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 95-102, University of Wisconsin, Madison, Wisconsin, USA, 22-25 juillet 1998. Morgan Kaufmann. ISBN 1-55860-548-7.