

Table des matières

Table des figures	3
Introduction	4
I. Conception du jeu	5
1. Introduction	5
2. Pygame	5
3. Sélection du mode de jeu	5
4. Sélection de la couleur des pions	6
5. Interface de jeu	7
II. Règles de jeu	8
1. Introduction	8
2. Réinitialisation du plateau	8
3. Règles de mouvement et gestion des tours	9
4. Validation du mouvement	9
III. Etude comparative des algorithmes d'intelligence artificielle	11
1. Introduction	11
2. L'apprentissage par renforcement	11
3. MiniMax	12
4. Choix final	12
IV. Développement de l'IA	13
1. Introduction	13
2. Mouvements valides	13
3. Génération de l'état du plateau	13
4. Fonction Minimax	14
V. Simulation	16
Conclusion	19

Table des figures

Figure 1: Choix du mode de jeu.....	5
Figure 2: Fonction de choix du mode de jeu	6
Figure 3 Choix de couleur	6
Figure 4 Fonction choix couleur	7
Figure 5 Interface de jeu.....	7
Figure 6 Reset	8
Figure 7 Step.....	9
Figure 8 Fonction is_valid_move	10
Figure 9 Reinforcement learning	11
Figure 10: Fonction get_valid_moves.....	13
Figure 11: generate_board_state	14
Figure 12 Minimax 1	14
Figure 13 Minimax 2	15
Figure 14 Phase 1	16
Figure 15 Phase 2	16
Figure 16 Phase 3	17
Figure 17 Phase 4	17
Figure 18 Phase 5	18
Figure 19 Phase 6	18

Introduction

Dans ce rapport, nous allons décrire le processus que nous avons suivi afin de terminer la conception et la réalisation du jeu Force3, et ensuite l'algorithme utilisé pour représenter l'IA, son développement et finalement l'intégrer dans notre jeu. Notre objectif principal dans ce projet est de créer une interface de jeu interactive permettant aux utilisateurs d'affronter une intelligence artificielle basée sur l'algorithme Minimax que nous avons choisi, ou simplement de regarder l'IA jouer contre elle-même dans une partie.

Le jeu de société Force3 se joue sur un plateau carré de neuf cases, avec deux joueurs en compétition pour aligner trois pièces de leur couleur horizontalement, verticalement ou en diagonale. L'algorithme Minimax a été introduit pour donner à l'IA la capacité de prendre des décisions stratégiques et compétitives pour rivaliser avec les joueurs humains.

Ce rapport détaillera la conception du jeu, l'intégration de l'algorithme Minimax et les fonctionnalités permettant aux utilisateurs de jouer contre l'IA ou de suivre des parties entre deux versions de l'IA. L'accent sera mis sur la phase de développement ensuite les ajustements spécifiques à l'IA pour le jeu Force3 et l'impact de l'algorithme Minimax sur l'expérience de jeu globale.

En résumé, cette étude détaille les différentes étapes de réalisation de ce projet, offrant un aperçu approfondi de la création du jeu Force3 en plus de l'algorithme Minimax, pour une expérience de jeu plus stimulante et stratégique.

I. Conception du jeu

1. Introduction

La phase de conception du jeu nous a nécessité une approche réfléchie pour offrir une expérience interactive et immersive aux joueurs. Dans cette section, nous allons décrire certains aspects importants de cette conception.

2. Pygame

Pygame est une bibliothèque libre multiplate-forme qui facilite le développement de jeux vidéo temps réel avec le langage de programmation Python. Elle est distribuée selon les termes de la licence GNU LGPL. Elle permet la programmation de la partie multimédia (graphismes, son et entrées au clavier, à la souris ou au joystick), sans se heurter aux difficultés des langages de bas niveaux comme le C et ses dérivés. Cela se fonde sur la supposition que la partie multimédia, souvent la plus contraignante à programmer dans un tel jeu, est suffisamment indépendante de la logique même du jeu pour qu'on puisse utiliser un langage de haut niveau (en l'occurrence le Python) pour la structure du jeu.

3. Sélection du mode de jeu

Au premier lancement, l'utilisateur devra choisir le mode de jeu qu'il souhaite adopter, soit de jouer contre l'IA soit regarder l'IA jouer avec elle-même comme le montre la figure ci-dessous :

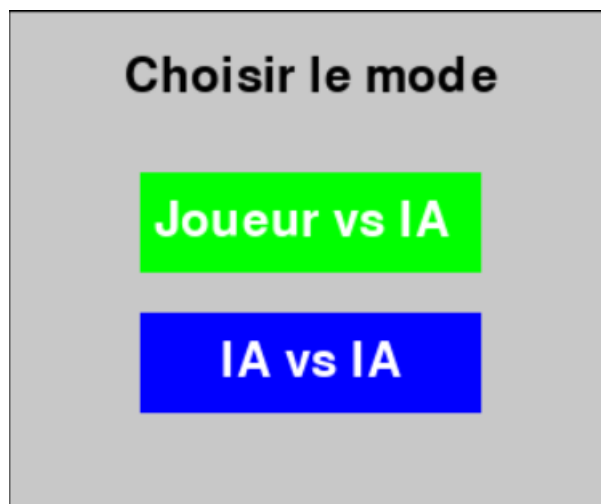


Figure 1: Choix du mode de jeu

Ceci est possible grâce à la fonction **choix_mode_de_jeu** :

```

def choix_mode_jeu():
    dialog_rect= pygame.Rect(LARGEUR // 8 , HAUTEUR /
pygame.draw.rect(ecran,(200,200,200),dialog_rect)
question_text = font.render("Choisir le mode", True,
text_x = dialog_rect.x + (dialog_rect.width - que
text_y = dialog_rect.y + 20
ecran.blit(question_text, (text_x, text_y))
bouton_largeur = dialog_rect.width // 2 +20
human_rect = pygame.Rect(dialog_rect.x + 65, dial
ia_rect = pygame.Rect(dialog_rect.x + 65, dialog
pygame.draw.rect(ecran, (0, 255, 0), human_rect)
pygame.draw.rect(ecran, (0, 0, 255), ia_rect)
human_text = font.render("Joueur vs IA", True, (2
ia_text = font.render("IA vs IA", True, (255, 255
ecran.blit(human_text, (human_rect.x + 7, human_r
ecran.blit(ia_text, (ia_rect.x + 40, ia_rect.y +

32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
pygame.display.update()
while True:
    for event in pygame.event.get():
        mode = None
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()
        elif event.type == pygame.MOUSEBUTTONDOWN:
            if human_rect.collidepoint(event.pos):
                mode= "Joueur"
            elif ia_rect.collidepoint(event.pos):
                mode= "IA"
        return mode
def choix_couleur():

```

Figure 2: Fonction de choix du mode de jeu

En effet, cette fonction permet de premièrement créer une boîte de dialogue, avec comme titre : Choisir le mode et deux rectangles placer un au dessous de l'autre, un pour faire le joueur contre l'IA et l'autre pour voir l'IA jouer contre elle-même.

Ensuite, deux boutons sont instanciés dans les rectangles des modes de jeu disponibles.

Finalement, dans la boucle pygame surveille les événements de clics de souris pour savoir si l'utilisateur choisi de jouer contre l'IA, le mode de jeu correspondant est stocké dans la variable **mode** qui est finalement renvoyé.

4. Sélection de la couleur des pions

Après avoir choisi le mode de jeu désiré. Si ce mode est joueur contre IA, le joueur sera renvoyé vers une nouvelle fenêtre, celle-ci est dédié pour qu'il puisse choisir la couleur des pions qu'il veut jouer.



Figure 3 Choix de couleur

Ceci est possible grâce à la fonction **choix_couleur** :

```

def choix_couleur():
    dialog_rect= pygame.Rect(LARGEUR // 8 , HAUTEUR /
pygame.draw.rect(ecran,(200,200,200),dialog_rect)
question_text = font.render("Choisir votre couleur
text_x = dialog_rect.x + (dialog_rect.width - que
text_y = dialog_rect.y + 20
ecran.blit(question_text, (text_x, text_y))
bouton_largeur = dialog_rect.width // 2 - 40
red_rect = pygame.Rect(dialog_rect.x + 20, dialog
blue_rect = pygame.Rect(dialog_rect.x + dialog_re
pygame.draw.rect(ecran, (255, 0, 0), red_rect)
pygame.draw.rect(ecran, (0, 0, 255), blue_rect)
red_text = font.render("Rouge", True, (255, 255,
blue_text = font.render("Bleu", True, (255, 255,
ecran.blit(red_text, (red_rect.x + 20, red_rect.y
ecran.blit(blue_text, (blue_rect.x + 25, blue_rec

63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78

pygame.display.update()
while True:
    for event in pygame.event.get():
        couleur = None
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()
        elif event.type == pygame.MOUSEBUTTONDOWN:
            if red_rect.collidepoint(event.pos):
                couleur= (255,0,0)
            elif blue_rect.collidepoint(event.pos):
                couleur= (0,0,255)

        return couleur

```

Figure 4 Fonction choix couleur

Pour cette, elle permet également créer une boîte de dialogue, mais pour titre : Choisir la couleur et deux rectangles placer un à côté de l'autre, rouge et bleu respectivement pour effectuer le choix des pions que l'utilisateur désire jouer.

Ensuite, deux boutons sont instanciés dans les rectangles de couleur disponibles.

Finalement, dans la boucle pygame surveille les événements de clics de souris pour savoir si l'utilisateur choisi la couleur rouge ou bleu, cette couleur sera stockée dans la variable **couleur** qui est finalement renvoyé.

5. Interface de jeu

L'interface de notre jeu se présente comme suit :

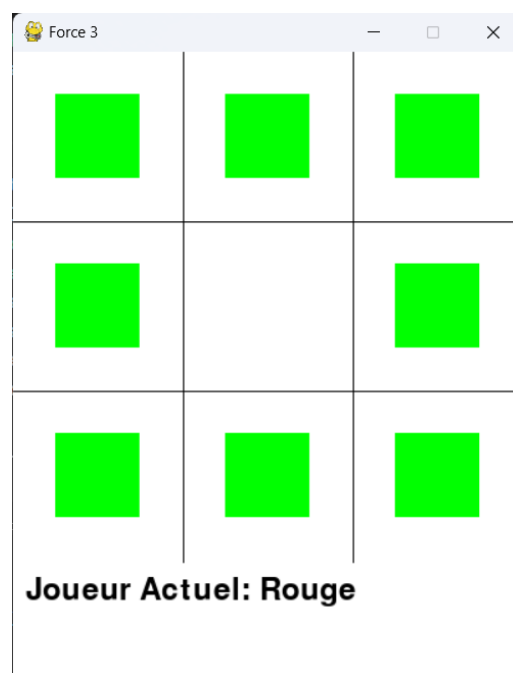


Figure 5 Interface de jeu

L'interface est sous forme d'un plateau de 9 cases (3 x 3), et dans les huit cases, on trouve un carré sauf dans la case centrale. Et en bas, on trouve la couleur du joueur actuelle, dans notre cas maintenant c'est le joueur rouge. Et à la fin du jeu la couleur du gagnant est aussi mentionnée au-dessous.

II. Règles de jeu

1. Introduction

Pour les règles de jeu, nous avons rassemblé l'intégralité de ces règles dans une classe Force3, nous allons maintenant parler des différentes méthodes de cette classe

2. Réinitialisation du plateau.

Pour gérer la réinitialisation du plateau, nous utilisons la méthode **reset** représentée ci-dessous :

```
def reset(self):
    # La carte est représentée
    # 0 représente un carré vid
    # 1 et -1 représentent les
    # 2 représente un jeton car
    self.board = [
        [2, 2, 2],
        [2, 0, 2],
        [2, 2, 2]
    ]
    # Le joueur 1 ou 2 commence
    self.current_player = 1
    # Garder une trace du nombre
    self.round_tokens_placed =
    # Etat du jeu - si le jeu e
    self.game_over = False
    # Gagnant du jeu, None si a
    self.winner = None
    # Mémoriser le dernier mouv
    self.last_move = None
    # Liste pour stocker l'hist
    self.move_history = []

    # Renvoie l'état initial de
    return self.board
```

Figure 6 Reset

Cette méthode réinitialise le plateau de jeu à l'état initial. Le terrain de jeu est représenté par une grille 3x3, avec des espaces vides, des marqueurs de joueurs et des marqueurs carrés définis avec la valeur 2. Tout d'abord, nous choisissons le joueur actuel sur 1. Cette méthode suit également le nombre de marqueurs de tour placés par chaque joueur, initialise l'état du jeu comme étant incomplet, définit le vainqueur qui n'a pas encore été décidé et réinitialise la liste

de l'historique des tours. Cette méthode vous permet de démarrer un nouveau jeu à partir d'un état de sauvegarde propre.

3. Règles de mouvement et gestion des tours

Pour gérer les mouvements possibles du jeu et les tours, nous utilisons la méthode **step** représentée ci-dessous :

```
def step(self, action):
    if self.game_over:
        return self.board, self.game_over, self.winner, False, "Game is over. Please res

    action_type, row, col, target_row, target_col, move_two_decision = action

    # Vérifier si l'action est valide
    if not self.is_valid_move(action_type, row, col, target_row, target_col):
        return self.board, self.game_over, self.winner, False, "Invalid move."

    if action_type == 'place_round':
        self.board[target_row][target_col] = self.current_player
        self.round_tokens_placed[self.current_player] += 1
        self.is_second_move_of_double = False # Réinitialiser pour le prochain tour
    elif action_type == 'move_square':
        self._move_square(row, col, target_row, target_col, move_two_decision)
        if move_two_decision and not self.is_second_move_of_double and (target_row in [0
            self.is_second_move_of_double = True # Marquer que le prochain mouvement es
        else:
            self.is_second_move_of_double = False # Réinitialiser pour le prochain tour
    elif action_type == 'move_round':
        self._move_round(row, col, target_row, target_col)
        self.is_second_move_of_double = False # Réinitialiser pour le prochain tour
```

Figure 7 Step

Dans cette méthode, nous vérifions d'abord si le jeu est terminé. Dans ce cas, il renvoie l'état actuel du jeu. Ensuite, l'action est décomposée en composants pour déterminer le type de mouvement et l'emplacement de la cible, la conformité de l'action aux règles est vérifiée et le jeu commence. Chaque type d'action, comme placer une tuile ronde ou déplacer une tuile carrée ou circulaire, est géré différemment. Cette méthode met à jour le plateau de jeu, vérifie s'il y a un gagnant et change le joueur actuel si nécessaire. Cette fonctionnalité est importante pour le déroulement du jeu, car elle gère à la fois les règles et la progression du jeu.

4. Validation du mouvement

Avant d'effectuer n'importe quel mouvement, ce dernier doit d'abord respecter certaines règles, tout ceci est garantie par la fonction **is_valid_move**


```

def is_valid_move(self, action_type, row, col, target_row, target_col):
    if self.is_second_move_of_double and action_type in ['place_round', 'move_round'] and
        return False
    # Vérification pour empêcher le mouvement inverse
    if len(self.move_history) > 0:
        case_depart, case_arrivee, player = self.move_history[-1]
        if (row == case_arrivee[0] and col == case_arrivee[1] and target_row == case_dep
            return False

    if action_type == 'place_round':
        # Vérifier si la cellule cible est carrée et si le joueur actuel n'a pas placé t
        return self.board[target_row][target_col] == 2 and self.round_tokens_placed[self
    elif action_type == 'move_square':
        # Vérifier si les indices sont dans les limites du tableau
        if not (0 <= row < 3 and 0 <= col < 3 and 0 <= target_row < 3 and 0 <= target_co
            return False
        else:
            if self.is_second_move_of_double:
                return (self.board[row][col] == 2 or self.board[row][col] == self.current
            # Vérifier si la cellule source a un jeton carré ou rond et que la cellule c
            return (self.board[row][col] == 2 or self.board[row][col] == self.current_pl
    elif action_type == 'move_round':
        # Vérifier si les indices sont dans les limites du tableau
        if not (0 <= row < 3 and 0 <= col < 3 and 0 <= target_row < 3 and 0 <= target_co
            return False
        else:
            # Vérifier si la cellule source contient le jeton de tour du joueur actuel e

```

Figure 8 Fonction *is_valid_move*

Elle envisage différents scénarios, puisqu'elle interdit les mouvements illégaux lors des doubles mouvements et bloque les mouvements inverses instantanés pour empêcher la répétition des stratégies de jeu. Elle vérifie également si le placement et le mouvement des pièces sont légaux, si le jeton tour est placé dans un espace valide ou si un mouvement est effectué dans les limites du plateau de jeu. Cette méthode est très importante pour jouer en douceur et éviter les erreurs de jeu.

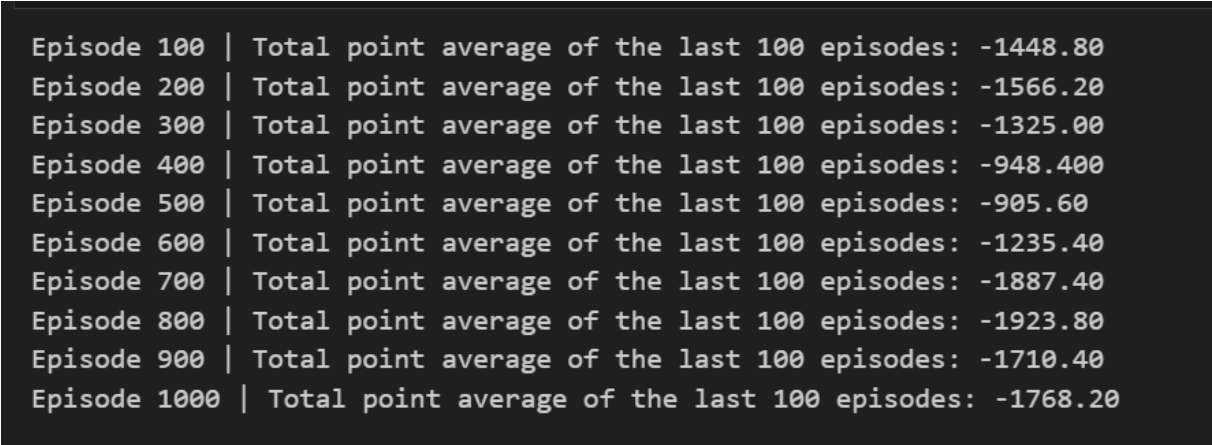
III. Etude comparative des algorithmes d'intelligence artificielle

1. Introduction

Dans cette partie nous allons voir les deux algorithmes possibles pour la réalisation de l'IA qui va pouvoir faire ce jeu qui sont, l'apprentissage par renforcement, et l'algorithme MiniMax.

2. L'apprentissage par renforcement

L'apprentissage par renforcement est une technique d'apprentissage machine où un agent apprend à prendre des décisions en interagissant avec un environnement pour maximiser une récompense cumulative. Ce type d'agent se caractérise par une adaptabilité aux environnements complexes, c.-à-d. qu'il est capable de traiter des environnements complexes où les actions prises peuvent entraîner des répercussions à long terme et des conséquences multiples. Mais aussi il peut apprendre tout seul en jouant avec lui-même et en s'affectant des récompenses ou des punitions pour mieux apprendre. Mais ce type d'apprentissage peut s'avérer un peu trop couteux, il nécessite de grande performance et de ressources informatiques GPU puisqu'il va aussi ajouter des couches de réseaux de neurones pour un apprentissage approfondi, il requiert également un grand nombre d'interactions avec l'environnement pour converger vers une politique optimale. Nous avons essayé d'adopter ce type d'algorithme mais la plupart des fois il n'arrive pas à converger.



Episode 100		Total point average of the last 100 episodes:	-1448.80
Episode 200		Total point average of the last 100 episodes:	-1566.20
Episode 300		Total point average of the last 100 episodes:	-1325.00
Episode 400		Total point average of the last 100 episodes:	-948.400
Episode 500		Total point average of the last 100 episodes:	-905.60
Episode 600		Total point average of the last 100 episodes:	-1235.40
Episode 700		Total point average of the last 100 episodes:	-1887.40
Episode 800		Total point average of the last 100 episodes:	-1923.80
Episode 900		Total point average of the last 100 episodes:	-1710.40
Episode 1000		Total point average of the last 100 episodes:	-1768.20

Figure 9 Reinforcement learning

Comme nous le remarquons dans cette figure l'algorithme n'arrive pas à converger, il se peut que le problème vienne de la configuration des réseaux de neurones, pour remédier à cela une des solutions était d'utiliser gridSearchCV qui permet de trouver les meilleurs hyperparamètres, mais nous retombons dans le même problème du manque de ressources.

3. MiniMax

L'algorithme minimax est un algorithme qui s'applique à la théorie des jeux pour les jeux à deux joueurs à somme nulle (et à information complète) consistant à minimiser la perte maximum. Pour une vaste famille de jeux, le théorème du minimax de von Neumann assure l'existence d'un tel algorithme, même si dans la pratique il n'est souvent guère aisé de le trouver. Le jeu de hex est un exemple où l'existence d'un tel algorithme est établie et montre que le premier joueur peut toujours gagner, sans pour autant que cette stratégie soit connue.

Il amène l'ordinateur à passer en revue toutes les possibilités pour un nombre limité de coups et à leur assigner une valeur qui prend en compte les bénéfices pour le joueur et pour son adversaire. Le meilleur choix est alors celui qui minimise les pertes du joueur tout en supposant que l'adversaire cherche au contraire à les maximiser (le jeu est à somme nulle).

4. Choix final

Pour notre choix final, nous avons décidé de continuer avec l'algorithme le Minimax pour notre projet car il est particulièrement adapté aux jeux à deux joueurs comme Force 3, où les joueurs alternent leurs coups. En plus de certaines fonctionnalités qui améliorent son utilisation, par exemple compte tenu de la portée limitée, l'algorithme minimax utilise un plateau de jeu de 9 cases et des règles strictes de placement et d'orientation, il peut effectivement explorer toutes les possibilités de déplacement sur un nombre limité de cases. Également, il fournit une évaluation précise des configurations possibles. De plus, l'algorithme évalue chaque configuration du tableau pour son potentiel à créer des placements, vous permettant ainsi de prendre des décisions stratégiques. Ceci est essentiel à la stratégie de ce jeu où il faut disposer les pièces d'une telle manière.

IV. Développement de l'IA

1. Introduction

Pour cette partie du développement, nous avons rassemblé toutes les méthodes nécessaires pour l'apprentissage dans une classe Bot.

2. Mouvements valides

Pour fonctionner, l'algorithme minimax a besoin des mouvement valides pour déterminer tout les états successeurs d'un état donné dans notre cas du plateau de jeu, ici la méthode **get_valid_moves** est adoptée.

```
def get_valid_moves(self,board):
    valid_moves= []
    action_types=['place_round','move_square','move_round']
    for action_type in action_types:
        for row in range (len(board)):
            for col in range(len(board)):
                for target_row in range(len(board)):
                    for target_col in range(len(board)):
                        if self.jeu.is_valid_move(action_type,row, col, target_row, target_col):
                            valid_moves.append((action_type,row, col, target_row, target_col,Fal
    return valid_moves
```

Figure 10: Fonction `get_valid_moves`

Cette méthode commence par initialiser une liste vide **valid_moves**, cette liste contiendra à la fin tous les mouvements possibles, et une autre liste qui contient tous les types d'actions possibles : **place_round**, **move_square**, **move_round**. Ensuite la méthode itère sur ces types d'actions, le nombre de lignes du plateau, le nombre de colonnes du tableau, et encore une fois le nombre de ligne pour déterminer la ligne cible et un dernière fois le nombre de colonne pour déterminer la colonne cible.

L'étape finale est la validation du mouvement, nous utilisons la méthode de la classe Force3 **is_valid_move** pour déterminer si le mouvement est valide, si oui le mouvement est stocké par la liste. Et finalement la liste est renvoyée.

3. Génération de l'état du plateau

L'algorithme minimax a aussi besoin des états successeur du plateau pour cela nous avons développé la méthode **generate_board_state** :

```

def generate_board_state(self, board, action, current_player):
    action_type, row, col, target_row, target_col, move_two_decision = action
    if action_type == 'place_round':
        board[target_row][target_col] = current_player
    elif action_type == 'move_square':
        if board[row][col] in [-1, 1, 2] and board[target_row][target_col] == 0:
            if board[row][col] == 2:
                # Déplacer un seul carré
                board[row][col], board[target_row][target_col] = 0, 2
            elif board[row][col] == 1:
                # Déplacer un seul carré
                board[row][col], board[target_row][target_col] = 0, 1
            else:
                # Déplacer un seul carré
                board[row][col], board[target_row][target_col] = 0, -1
    elif action_type == 'move_round':
        if board[row][col] == -1 and board[target_row][target_col] == 2 and ((row == target_row and col - target_col)
            board[row][col], board[target_row][target_col] = 2, current_player
    return board

```

Figure 11: generate_board_state

En effet, cette méthode utilise à peu près les mêmes règles de placement de pions ou déplacement de carré ou de pions que la méthode **step** dans la classe Force3, elle est simplement utilisée dans notre algorithme pour renvoyer le nouvel état du plateau.

4. Fonction Minimax

Maintenant nous passons à la méthode principale, Minimax :

```

def minimax(self, board, depth, current_player):
    best_move = ("place_round", 0, 0, 0, 0, False)
    if depth == 0 or self.jeu.game_over:
        best_score, best_move = self.evaluate(best_move)
        return best_score, best_move
    if current_player == -1:
        best_score = -inf
    else:
        best_score = +inf
    for valid_move in self.get_valid_moves(board):
        if self.last_move != 0:
            if valid_move[1] != self.last_move[3] and valid_move[2] != self.last_move[4]:
                copied = copy.deepcopy(board)
                board = self.generate_board_state(board, valid_move, current_player)
                score, _ = self.minimax(board, depth-1, -current_player)
                board[valid_move[1]][valid_move[2]], board[valid_move[3]][valid_move[4]] = copied[
            if current_player == -1:
                if score > best_score:
                    best_score = score
                    best_move = valid_move
            else:
                if score < best_score:
                    best_score = score
                    best_move = valid_move

```

Figure 12 Minimax 1

```

elif self.last_move==0:
    copied = copy.deepcopy(board)
    board = self.generate_board_state(board,valid_move,current_player)
    score,_ = self.minimax(board,depth-1,-current_player)
    board[valid_move[1]][valid_move[2]],board[valid_move[3]][valid_move[4]]=co
    if current_player == -1:
        if score>best_score:
            best_score=score
            best_move=valid_move
    else:
        if score<best_score:
            best_score=score
            best_move=valid_move

return best_score,best_move

```

Figure 13 Minimax 2

La fonction prend en paramètres le plateau de jeu dans la variable **board**, la profondeur actuelle dans l'arbre de recherche dans la variable **depth**, et le joueur actuel qui est **current_player**.

La fonction vérifie tout d'abord la profondeur, si elle est nulle ou si la partie est terminée, la fonction évalue la meilleure action actuelle en appelant la fonction **évalue** et renvoie le score obtenu ainsi que le mouvement associé.

Si le joueur actuel est l'IA, **best_score** est initialisé à moins l'infini donc son objectif est de maximiser cette valeur et il essaie de la minimiser pour le deuxième joueur.

La fonction parcourt tous les mouvements valides disponibles en utilisant la méthode **self.get_valid_moves(board)**. Ensuite, elle vérifie si le mouvement est répété pour empêcher le mouvement inverse qui est stockée dans la variable **self.last_move**.

Pour chaque mouvement valide, nous faisons une copie du plateau de jeu dans la variable **copied** conserver l'ancien état du plateau. Ensuite, nous mettons à jour le plateau de jeu avec ce mouvement valide en utilisant la méthode **get_board_state**. Après ceci, vient l'appel récursive de la méthode minimax avec le nouveau plateau, une profondeur réduite de 1 et en changeant le joueur. Nous restaurons l'état du plateau à son état précédent pour annuler mouvement. En fonction du joueur actuel, c'est-à-dire s'il veut maximiser ou minimiser le score, la fonction met à jour le **best_score** et le **best_move** avec le score obtenu pour le mouvement actuel. Une fois que tous les mouvements valides ont été évalués, la fonction retourne le meilleur score et le meilleur mouvement trouvés après exploration de tous les mouvements possibles à cette profondeur.

V. Simulation

Après lancement de jeu, comme nous l'avons précisé, nous allons choisir le mode de jeu et après la couleur du pion si le mode est joueur contre IA. Nous allons choisir ce mode ainsi que rouge comme couleur de notre pion.

Phase 1 : Je place le pion à la première position et l'IA à la deuxième de la première ligne

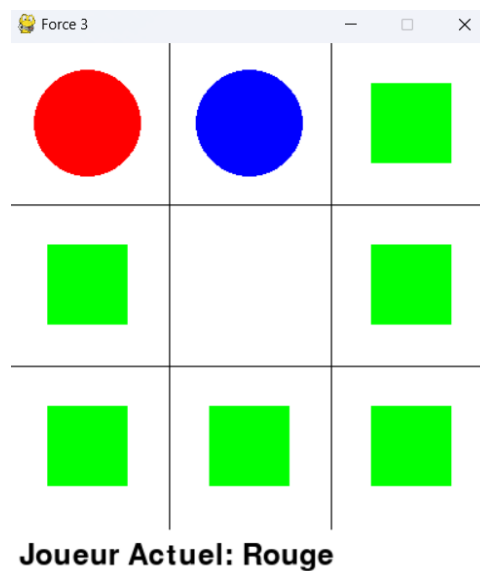


Figure 14 Phase 1

Phase 2 : Je place le pion à la deuxième ligne et l'IA à côté du premier pion

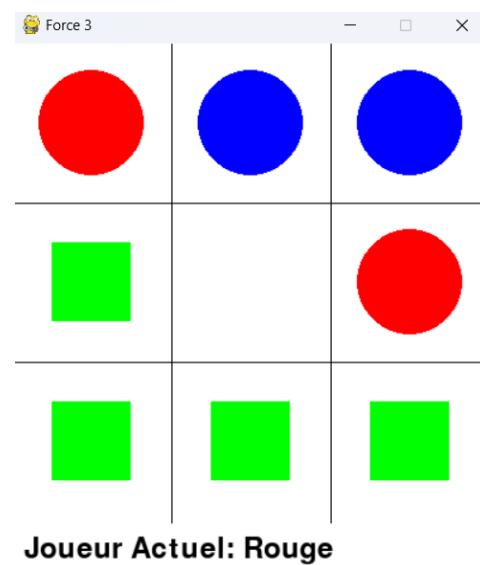


Figure 15 Phase 2

Phase3 : Je place le pion à la deuxième ligne et l'IA juste en dessous.

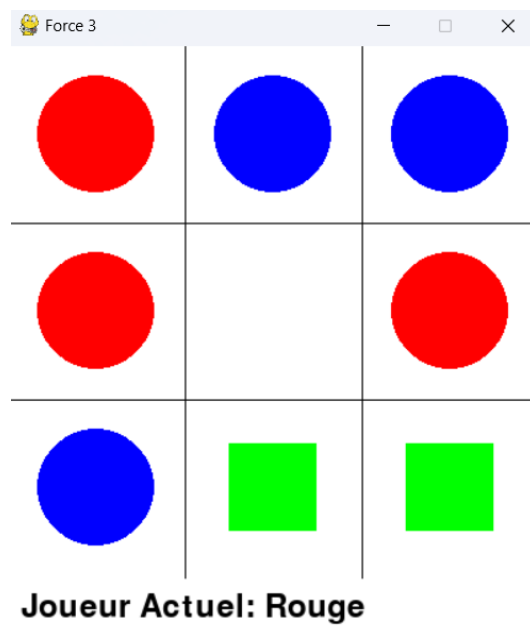


Figure 16 Phase 3

Phase 4 : Je mets le pion au milieu, l'IA bloque la position afin que je ne puisse pas gagner.

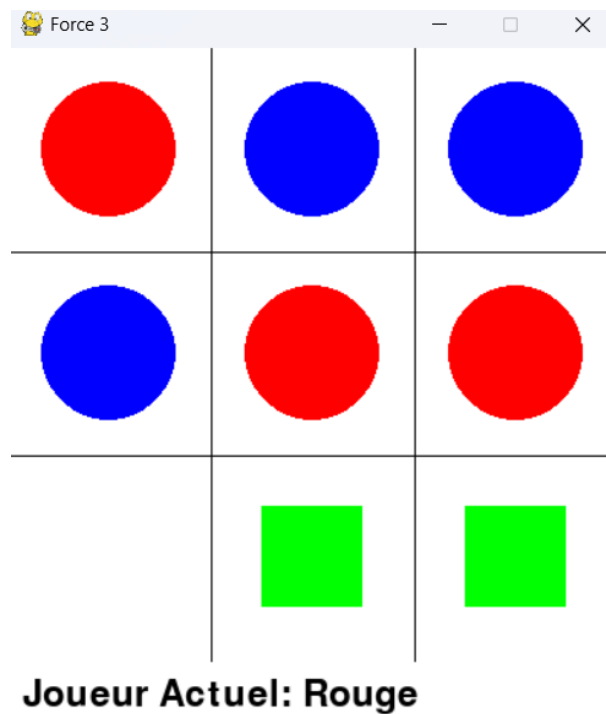


Figure 17 Phase 4

Phase 5 : je remets le pion en bas et l'IA également

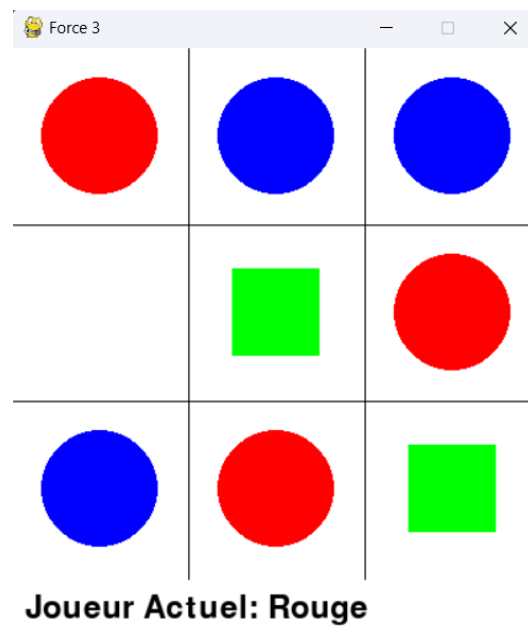


Figure 18 Phase 5

Phase 6 : je remets l'autre pions en base, et l'IA forme une ligne en diagonale et elle gagne

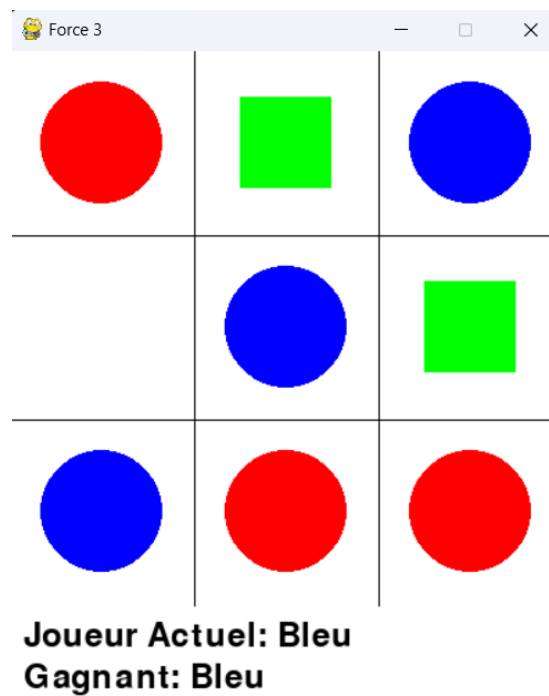


Figure 19 Phase 6