École Centrale Lyon

UE APPRO

Stratégies de résolution de Problèmes

Rapport

BE2 - Le jeu Morpion

Élèves :
Matías Duhalde
matias.duhalde@ecl22.ec-lyon.fr

Enseignant :
Alexandre Saidi

12 novembre 2022

Table des matières

1 Introduction

Le **morpion**, aussi appelé ***tic-tac-toe***, est un jeu à deux joueurs, qui jouent sur un **plateau** carré (**grille** carrée) de taille $D > 2$, dans lequel les joueurs placent des **pions** (traditionnellement X et O). Chaque joueur possède D pions, et le but est de, à partir d'un plateau vierge, les placer tous sur une même **ligne**, **colonne** ou **diagonale** pour gagner. Avant de commencer, un premier joueur est choisi et après avoir placé une pièce, le joueur suivant jouera son tour. Si un joueur à déjà placé tous ses pions, il devra en choisir un dans le plateau et le **déplacer** à une nouvelle case. Le jeu peut être déclaré **nul** si les joueurs atteignent un état déjà atteint précédemment (donc, le jeu a toujours une fin, puisque le nombre total d'états est fini).

Plusieurs modalités de jeu sont proposées dans ce rapport : **humain contre humain**, **humain contre machine (IA)** et **machine contre machine**.

Ce *BE* nous permet d'aborder une stratégie de gain pour la machine (**Best First** : choix du meilleur coup local ou global) via différentes méthodes (**fonction cout**, **Min-Max**, **Alpha-Beta**, etc.).

2 Mise en œuvre

Dans ce rapport, le principal point de différence entre les algorithmes proposés est lié à la façon dont la machine calcule le coup suivant. Ainsi, toutes les fonctionnalités "de base" du jeu ont un code similaire en ***Python***. Par conséquent, une classe de base (`MorpionBase`) a été créée avec des fonctions communes à toutes les implémentations. Ce code est disponible dans le fichier `morpion_base.py`. Dans cette section, le code et les algorithmes du fichier seront expliqués et analysés.

2.1 Modélisation

L'une des parties les plus importantes du modélisation est la façon dont nous gardons l'état du plateau. Pour le faire, nous pourrions recourir à une matrice carrée de taille D, chaque élément correspondant à une case et indiquant si un jouer la tient (et quel joueur) ou si elle est vide. Cependant, lors de ce rapport, une autre représentation a été choisie : utiliser une série de **listes** (taille N avec $0 \leq N \leq D$) contenant les cases qui chaque joueur possède et une liste qui contient les cases vides (taille N_v avec $0 \leq N_v \leq D^2$). Chaque fonction de évaluation et chaque algorithme utilisera ce représentation pour faire les calculs et déterminer le prochaine coup.

2.2 Cycle de jeu

Comme indiqué précédemment, le jeu est au **tour par tour**, il est donc nécessaire d'avoir un moyen d'**alterner** entre les joueurs. Ci-dessous (extrait 1) le code qui implémente le **cycle principale du jeu** (fonction `jouer`).

```
1 def jouer(self):
2     self.joueur_actuel = choice(self.JOUEURS)
3     fini = False
4     while not fini and not self.jeu_nul:
5         self.log(f'Joueur actuel: {self.joueur_actuel}')
6         self.print_matrice()
```

```
            case_placee = self.placer_un_pion()
            if case_placee:
                fini = self.gagnant(case_placee)
                if not fini:
                    self.basculer_joueur()
        self.print_matrice()
        if self.jeu_nul:
            self.log('Jeu nul (répétition)')
        else:
            self.vainqueur = self.joueur_actuel
            self.log(f'Joueur {self.vainqueur} a gagné !')
```

Listing 1 – Code de la fonction du cycle de jeu

Avant de commencer, un **joueur est choisi au hasard** pour jouer en premier. Ensuite, une boucle ***while*** est lancée, qui montre l'état actuel du plateau (fonction `print_matrice`) et demande au joueur (humain ou machine) de **jouer une pièce** (fonction `placer_un_pion`). Si un pion est placé avec succès sur le plateau, nous vérifions si l'état est **gagnant**. Si ce n'est pas le cas, le joueur actuel change (son tour se termine) et la boucle recommence. Le cycle continue jusqu'à ce que l'un des joueurs atteigne une **position gagnante**, ou qu'un état précédent soit à nouveau atteint (**jeu nul**). La condition de jeu nul est vérifiée en plaçant le pion.

2.3 Placer un pion

Le code de la fonction `placer_un_pion` mentionnée ci-dessus se trouve dans l'extrait 2.

```
def placer_un_pion(self):
    joueur_est_humain = self.joueurs[self.joueur_actuel] == 'humain'
    if not self.joueur_peut_ajouter_nouvelle_forme(self.joueur_actuel):
        # retirer pion
        if joueur_est_humain:
            case_choisi = self.get_input()
            if not self.essai_marquer_case(case_choisi):
                return None
        else:
            if not self.essai_marquer_case(self.jouer_ia()):
                return None
        self.print_matrice()
    # ajouter pion
    if joueur_est_humain:
        case_choisi = self.get_input()
        if self.essai_marquer_case(case_choisi):
            return case_choisi
    else:
        case_choisi = self.jouer_ia()
        if self.essai_marquer_case(case_choisi):
            return case_choisi
    return None
```

Listing 2 – Code de la fonction de placer un pion

Avant de placer un pion, il faut vérifier si le joueur peut effectivement en rajouter. Nous le faisons en utilisant la fonction `joueur_peut_ajouter_nouvelle_forme`, qui vérifie simplement la longueur de la liste de pièces placées du joueur et si elle est égal à D

($O(1)$). **Si le joueur peut placer un nouveau pion, la fonction ne lui demande qu'une seule case, sinon il devra fournir deux cases** : la première à enlever et la deuxième (nécessairement différente) a placer.

Lorsque le joueur est humain, nous obtenons son *input* à partir de la fonction `get_input`. Sinon, quand le joueur est la machine (IA), nous obtenons la case à jouer de la fonction `jouer_ia`, qui dépend de chaque algorithme utilisé.

Après d'avoir reçu la choix du joueur, la fonction `_essai_marquer_case` [1] est chargée de **vérifier si la choix est valide**, et au même temps de **faire le changement dans le plateau**. L'extrait 3 montre le code de cette fonction.

```
def _essai_marquer_case(self, case: Case) -> Union[bool, Case]:
    if (self.case_est_libre(case) and
            self.joueur_peut_ajouter_nouvelle_forme(self.joueur_actuel)
    and
            self.case_videe != case):
        self.marquer_case(case, self.joueur_actuel)
        self.case_videe = None
        self.enregistrer_etat()
        return case
    if (self.case_videe is None and
        not self.joueur_peut_ajouter_nouvelle_forme(self.joueur_actuel)
    and
            self.case_est_de_joueur(case, self.joueur_actuel)):
        self.liberer_case(case, self.joueur_actuel)
        self.case_videe = case
        return case
    return False
```

Listing 3 – Code de la fonction de marquer la case

La première condition vérifie si la case choisie peut être ajouté par le joueur. Pour cela, il faut vérifier que **la case soit libre** (`case_est_libre`, $O(D^2)$), que **le joueur puisse ajouter un nouveau pion** et que **la case soit différente à la case vidée par le joueur dans le même tour** ($O(1)$), s'il y en a eu une. Si la vérification réussit, nous ajoutons la case au plateau (`marquer_case`, $O(D)$) et nous enregistrons son état (`enregistrer_etat`).

La deuxième condition vérifie si la case choisie peut être libérée par le joueur. Il faut confirmer que **le joueur n'ait pas déjà vidée une autre case** dans le même tour ($O(1)$), **la case appartienne au joueur** (`case_est_de_joueur`, $O(D)$) et que **le joueur ne puisse pas placer un nouveau pion** (les cases ne peuvent pas être retirées si le joueur à des pions à placer). Si la vérification réussit, nous libérons la case (`liberer_case`, $O(D)$).

2.4 Enregistrer l'état et vérifier le jeu nul

L'extrait de code 4 montre le processus de **sauvegarde des états** et de **vérification**. Le code utilise la structure de données *dict* de Python pour vérifier si **l'état à été déjà atteint**, dont la complexité en temps est $O(1)$ (cas moyen). Si on trouve que l'état se répète, on marque `jeu_nul` comme True, et **le jeu s'arrêtera avant le prochain tour**.

1. Il existe une autre fonction dans le code portant le même nom mais sans tiret bas initial, mais son seul but est d'afficher des messages et non de faire la vérification.

```python
def enregistrer_etat(self):
    nouvel_etat = tuple(frozenset(self.coords_joueur[k]) for k in self.coords_joueur)
    if len(self.cases_vides) <= self.dimension**2 - 2 * self.dimension:
        if nouvel_etat in self.etats_precedents:
            self.jeu_nul = True
        self.etats_precedents[nouvel_etat] = True
```

Listing 4 – Code de la fonction d'enregistrer et vérifier l'état

2.5 Vérifier plateau gagnant

Après chaque tour, l'algorithme vérifie si le plateau est gagnant ou non pour déterminer si le jeu doit continuer. La vérification est simple, et on utilise les coordonnées de la dernière case placée pour réduire la quantité de lignes / colonnes / diagonales à vérifier, en observant seulement la ligne, la colonne, et les diagonales qui contiennent la dernière case placée. La fonction `gagnant` dans le code fait la vérification, et sa complexité en temps (pire cas) est $O(4 \cdot D) = O(D)$

2.6 Interface

Pour visualisation plus facile du problème, un exemple de code *Python* utilisant ***tkinter*** nous a été fourni. Ce code a été modifié pour permettre son utilisation avec toutes les versions de l'IA, de sorte que seule une petite modification doit être effectuée à chaque fois. Il se trouve dans le fichier `interface_base.py`.

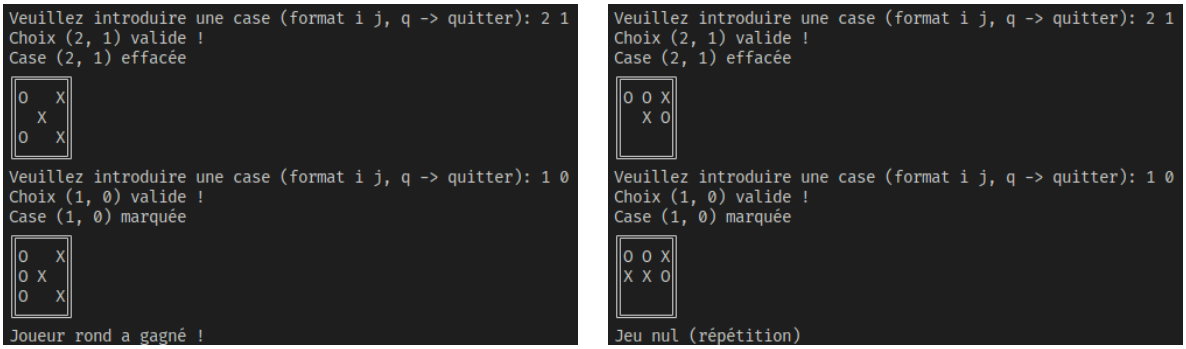
2.7 Sommaire fonctions de base

En résumé, dans chaque boucle, les actions les plus importantes sont :

1. **Placer un pion** : la complexité dépend de la fonction que la machine utilise pour choisir une case, qui sera discutée dans la section suivante.
2. **Vérifier condition jeu nul** : l'algorithme cherche l'état dans une *hash table*, où l'ajout d'un élément coûte $O(1)$ et la vérification de la présence d'un élément coûte aussi $O(1)$.
3. **Vérifier condition gagnante** : la complexité est $O(D)$, nous vérifions la ligne (1), la colonne (1) et les diagonales (2) de la dernière case placée, dont la longueur est D.
4. **Basculer le joueur** : simplement changer le joueur dont c'est le tour, $O(1)$.

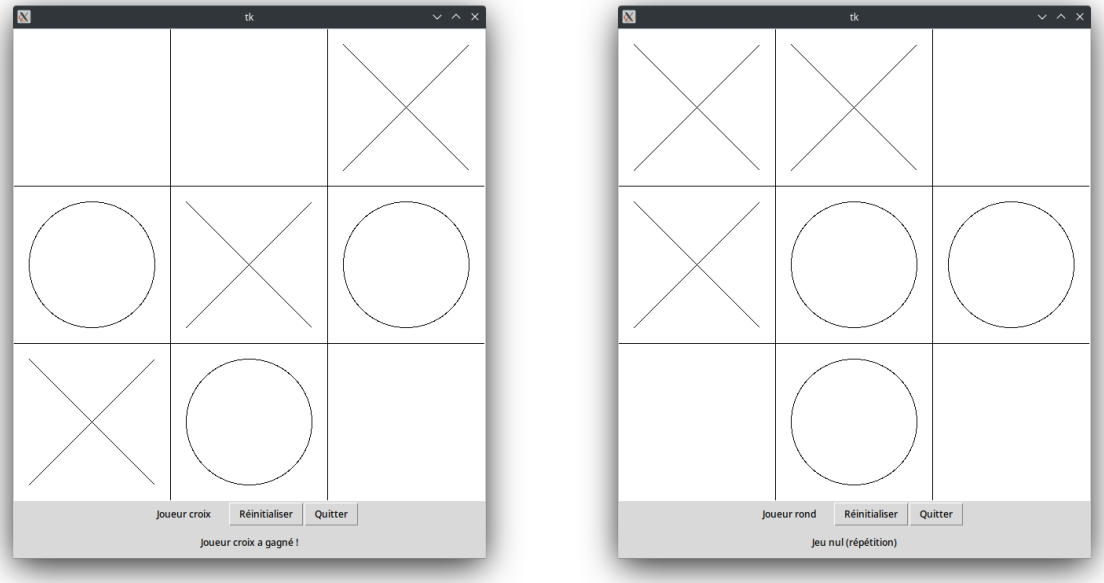
2.8 Exemple d'exécution

L'exécution des fichiers de code donnés dans les sections suivantes donnera une *output* dans la console comme montré dans la figure 1 et une interface comme montré dans la figure 2.



(a) Exemple d'exécution gagnante (b) Exemple d'exécution nulle

FIGURE 1 – Exemple d'exécution - Console



(a) Exemple d'exécution gagnante (b) Exemple d'exécution nulle

FIGURE 2 – Exemple d'exécution - Interface

3 IA

Dans cette section, le code lié à l'intelligence artificielle et la différence entre chaque implémentation seront décrits et analysés.

3.1 Choix aléatoire

3.1.1 Implémentation

Le code ci-dessous (extrait 5) montre la mise en œuvre de l'IA avec choix aléatoire. L'idée est assez simple : Si le joueur doit enlever un pion, choisir une de ses cases par **hasard**, et si le joueur doit en placer un, choisir aléatoirement une des cases vides (sauf la case que vient d'être libérée, appelée `restriction` dans le code).

```
def jouer_ia(self) -> Case:
    if self.joueur_peut_ajouter_nouvelle_forme(self.joueur_actuel):
        rest = []
        if self.case_videe:
            rest.append(self.case_videe)
        return self.choisir_case_vide(rest)
    return self.choisir_case_propre()

def choisir_case_vide(self, restriction: Union[None, list[Case]] = None)
    -> Case:
    cases = self.cases_vides
    if restriction:
        cases = [case for case in cases if case not in restriction]
    return choice(cases)

def choisir_case_propre(self, restriction: Union[None, list[Case]] =
   None) -> Case:
    cases = self.coords_joueur[self.joueur_actuel]
    if restriction:
        cases = [case for case in cases if case not in restriction]
    return choice(cases)
```

Listing 5 – Code de l'implémentation de l'IA avec choix aléatoire

Le code se trouve dans le fichier `morpion_v1.py`, correspondant au **travail à rendre 1**, et peut être exécute en utilisant le commande `python morpion_v1.py`. Par défaut, l'exécution lancera un jeu *machine contre machine* avec un plateau de taille 3. Cependant le code dans `if __name__ == "__main__"` peut être modifié pour changer la taille ou jouer contre la machine ou un autre joueur humain.

Le fichier `morpion_v2.py` correspond au **travail à rendre 2**, et utilise la même heuristique aléatoire décrite ci-dessus. Le fichier peut être modifié de la même façon décrite précédemment.

3.1.2 Analyse

La fonction `jouer_ia` est appelée deux fois dans un tour où le joueur à déjà placée tous ses pions, une fois pour choisir un pion à retirer, et une autre pour choisir un case à remplir. La complexité en temps est $O(1)$ pour choisir une case propre, et $O(D^2)$ pour choisir une case vide si le joueur vient de retirer un pion, étant donné qu'il faut enlever la dernière case retirée des options possibles.

Au total, la complexité en temps de chaque tour s'élève à $O(D^2)$.
Après plusieurs jeux machine contre machine (taille 3) :

1. 48.3% : joueur qui commence gagne
2. 43.3% : joueur qui suive gagne
3. 8.4% : jeu nul

Avec tailles plus grandes, l'avantage du premier joueur diminue et les jeux nuls augmentent. Pour $D = 5$:

1. 25.2% : joueur qui commence gagne
2. 24.8% : joueur qui suive gagne
3. 50% : jeu nul

3.2 Fonction de coût (Best-First)

3.2.1 Implémentation

Pour améliorer notre IA, au lieu de la laisser choisir son prochain coup au hasard, nous utilisons une heuristique et une fonction qui permet à l'ordinateur d'analyser l'état de chaque plateau. Pour atteindre cet objectif, nous utilisons une stratégie *best-first*, c'est-à-dire, analyser chaque mouvement qui peut être dans le tour actuel et choisir le meilleur selon notre fonction.

Pour attribuer un score à chaque configuration possible nous utilisons une approche *équationnelle*, avec une fonction qui considère la quantité de pions du joueur et de l'adversaire dans chaque ligne, colonne et diagonale, en privilégiant le placement d'un nouveau pion dans une position qui bloque les pièces bien placées de l'adversaire.

```
def jouer_ia(self) -> Case:
    if len(self.res_stack) != 0:
        return self.res_stack.pop()
    self.res_stack.extend(self.best_first())
    return self.res_stack.pop()

def best_first(self) -> tuple[Case, ...]:
    valeurs = defaultdict(list)
    if self.joueur_peut_ajouter_nouvelle_forme(self.joueur_actuel):
        for case in self.cases_vides:
            valeurs[self.obtenir_valeur_case(case)].append((case,))
    else:
        for case_a_enlever in [*self.coords_joueur[self.joueur_actuel]]:
            cases_vides_avant = [*self.cases_vides]
            self.liberer_case(case_a_enlever, self.joueur_actuel)
            for case_a_placer in cases_vides_avant:
                valeurs[self.obtenir_valeur_case(case_a_placer)].append(
                    (case_a_placer, case_a_enlever))
            self.marquer_case(case_a_enlever, self.joueur_actuel)
    best = valeurs[max(valeurs)]
    # privilégier diag
    best_diags = list(filter(lambda x: x[0][0] == x[0][1]
                    or x[0][0] + x[0][1] == self.dimension - 1, best))
    if best_diags:
        return choice(best_diags)
    return choice(best)
```

```
27
28 def obtenir_valeur_case(self, case: Case) -> int:
29     autre = 'croix' if self.joueur_actuel == 'rond' else 'rond'
30     nl = self.nombre_pions_ligne(case)
31     nc = self.nombre_pions_colonne(case)
32     nl_1 = nl[self.joueur_actuel]
33     nc_1 = nc[self.joueur_actuel]
34     nl_2 = nl[autre]
35     nc_2 = nc[autre]
36     facteur_nl = -1 if nl_2 > nl_1 else 1
37     facteur_nc = -1 if nc_2 > nc_1 else 1
38     g_partial = facteur_nl * (nl_1 - nl_2)**2 + facteur_nc * (nc_1 -
    nc_2)**2
39
40     facteur_nd1 = 1
41     facteur_nd2 = 1
42     if case[0] == case[1]:
43         nd1 = self.nombre_pions_diag_1(case)
44         nd1_1 = nd1[self.joueur_actuel]
45         nd1_2 = nd1[autre]
46         facteur_nd1 = -1 if nd1_2 > nd1_1 else 1
47         g_partial = g_partial + facteur_nd1 * (nd1_1 - nd1_2)**2
48         facteur_nd1 = 2 if nd1_2 > 1 else 1
49
50     if case[0] + case[1] == self.dimension - 1:
51         nd2 = self.nombre_pions_diag_2(case)
52         nd2_1 = nd2[self.joueur_actuel]
53         nd2_2 = nd2[autre]
54         facteur_nd2 = -1 if nd2_2 > nd2_1 else 1
55         g_partial = g_partial + facteur_nd2 * (nd2_1 - nd2_2)**2
56         facteur_nd2 = 2 if nd2_2 > 1 else 1
57
58     facteur_nl = 2 if nl_2 > 1 else 1
59     facteur_nc = 2 if nc_2 > 1 else 1
60     g = g_partial * facteur_nc * facteur_nl * facteur_nd1 * facteur_nd2
61     return abs(g)
```

Listing 6 – Code de l'implémentation de l'IA avec fonction de coût (best first)

Le code (**travail à rendre 3**) se trouve dans le fichier `morpion_v3.py` pour la version console et le fichier `morpion_v3_interface.py` pour la version interface. Leur exécution fonctionne de la même manière à celle décrite dans la section précédente. Par défaut, le code lance un jeu *humain contre machine*, mais cela peut être modifié.

3.2.2 Analyse

Dans l'extrait ci-dessus (extrait 6), la fonction `best_first` est appelée chaque fois que l'IA doit décider une nouvelle case à jouer : si le joueur peut ajouter un nouveau pion, elle itère sur toutes les coordonnées vides (D^2 **possibilités** au pire) et calcule le score de chaque possibilité. Si le joueur doit enlever un pion avant de jouer une case, la fonction calcule les scores de toutes les possibilités en enlevant un pion puis en le déplaçant sur chaque case vide ($D \cdot D(D-2)$ **possibilités** dans le pire des cas).

Dans chaque possibilité, nous utilisons la fonction `obtenir_valeur_case` pour **obtenir la valeur tentative du nouveau plateau si le joueur place un pion dans cette case**. La stratégie est plutôt *équationnelle* et elle est basée sur le nombre de pions

du joueur et de l'adversaire dans chaque ligne, colonne et diagonale, valeurs obtenus à l'aide des fonctions `nombre_pions_ligne`, `nombre_pions_colonne`, etc. La complexité de ces fonctions est idéalement $O(D)$. Le reste des calculs dans la fonction sont de complexité en temps constante.

Finalement, la fonction trouve la valeur maximale calculée dans l'étape précédente et ses cases correspondantes ($O(D)$). S'il y en a plusieurs, nous en choisissons une par hasard, en privilégiant celles des diagonales.

Étant donné qu'au pire cas la fonction `best_first` analyse $D \cdot D(D-2)$ coups, et chaque fois elle calcule la valeur de la case $O(D)$, la complexité temporelle de cette approche est $O(D^4)$.

En mettant la machine contre elle-même, dans un plateau de taille 3, les résultats sont différents à ceux de l'approche par hasard :

1. 47.5% : jeu nul
2. 23.5% : joueur qui commence gagne
3. 29% : joueur qui suive gagne

Si nous augmentons la taille du plateau, la quantité de jeux nuls grandit de manière significative. Avec taille 4, nous obtenons :

1. 95% : jeu nul
2. 5% : joueur qui commence gagne
3. 0% : joueur qui suive gagne

Et avec tailles supérieures (5, 6 et 7), les jeux se terminent presque toujours par répétition :

1. 99.6% : jeu nul
2. 0.2% : joueur qui commence gagne
3. 0.2% : joueur qui suive gagne

3.3 Min-Max

3.3.1 Implémentation

Pour améliorer encore notre IA, Nous adoptons une nouvelle stratégie **Min-Max**. L'idée principale de cette approche est de analyser tous les coups possibles dans le tour actuel, et un nombre k de tours suivantes, en évaluant chaque plateau final (où l'algorithme atteint la profondeur k ou un état gagnant) par une fonction qui tient compte du nombre de lignes, colonnes, et diagonales qui peuvent être potentiellement gagnants. Dans les tours où le joueur joue, **Min-Max** cherche a maximiser la valeur de la fonction d'évaluation (pour prendre le meilleur coup). Par contre, dans les tours où l'adversaire joue, **Min-Max** minimise la valeur, en assumant que l'adversaire jouera aussi le meilleur coup qu'il trouve.

```
def __init__(self, dimension: int, player1='humain', player2='humain'):
    self.min_max_profondeur = 2
    super().__init__(dimension, player1, player2)

def jouer_ia(self) -> Case:
    if len(self.res_stack) != 0:
        return self.res_stack.pop()
```

```
8       self.res_stack.extend(self.min_max())
9       return self.res_stack.pop()
10
11  def min_max(self) -> tuple[Case, ...]:
12      coords_joueur_copy = deepcopy(self.coords_joueur)
13      cases_vides_copy = [*self.cases_vides]
14      plateau, _ = self.min_max_rec(
15          (self.coords_joueur, self.cases_vides, (0, 0), None),
16          self.min_max_profondeur, self.joueur_actuel, 'max'
17      )
18      self.coords_joueur = coords_joueur_copy
19      self.cases_vides = cases_vides_copy
20      if isinstance(plateau[3], tuple):
21          return (plateau[2], plateau[3])
22      return (plateau[2],)
23
24  def min_max_rec(self, plateau: Plateau, k: int, joueur_a_jouer: str,
        mode: str) -> tuple[Plateau, float]:
25      autre = self.autre_joueur(joueur_a_jouer)
26      # set vars pour pouvoir utiliser les méthodes de la classe
27      self.coords_joueur = plateau[0]
28      self.cases_vides = plateau[1]
29      derniere_case_placee = plateau[2]
30
31      # si on est à la profondeur max ou l'autre joueur gagne, on renvoie
        le plateau + son score
32      if k != self.min_max_profondeur:
33          est_gagnant = self.gagnant(derniere_case_placee, autre)
34          if est_gagnant:
35              score = math.inf if autre == self.joueur_actuel else -math.
        inf
36              return plateau, score
37          if k == 0:
38              score = self.valeur_min_max(self.joueur_actuel)
39              return plateau, score
40
41      min_score = +math.inf
42      max_score = -math.inf
43
44      # trouver successeurs
45      liste_plateaux_successeurs: list[Plateau] = []
46      cases_vides_avant = [*self.cases_vides]
47      if self.joueur_peut_ajouter_nouvelle_forme(joueur_a_jouer):
48          for case_vide in cases_vides_avant:
49              self.marquer_case(case_vide, joueur_a_jouer)
50              new_coords_joueur = deepcopy(self.coords_joueur)
51              new_cases_vides = [*self.cases_vides]
52              liste_plateaux_successeurs.append(
53                  (new_coords_joueur, new_cases_vides, case_vide, None))
54              self.liberer_case(case_vide, joueur_a_jouer)
55      else:
56          for case_occupee in [*self.coords_joueur[joueur_a_jouer]]:
57              self.liberer_case(case_occupee, joueur_a_jouer)
58              for case_vide in cases_vides_avant:
59                  self.marquer_case(case_vide, joueur_a_jouer)
60                  new_coords_joueur = deepcopy(self.coords_joueur)
61                  new_cases_vides = [*self.cases_vides]
```

```
                liste_plateaux_successeurs.append(
                    (new_coords_joueur, new_cases_vides, case_vide,
    case_occupee))
                self.liberer_case(case_vide, joueur_a_jouer)
            self.marquer_case(case_occupee, joueur_a_jouer)

    best_plateaux = []
    best_score = 0
    new_mode = 'min' if mode == 'max' else 'max'
    # tester tous les plateaux successeurs
    for plateau_successeur in liste_plateaux_successeurs:
        plateau_successeur_2, score = self.min_max_rec(
            plateau_successeur, k - 1, autre, new_mode)
        if mode == 'max':
            if score == max_score:
                best_plateaux.append(plateau_successeur_2)
            elif score > max_score:
                best_score = score
                best_plateaux = [plateau_successeur_2]
                max_score = best_score
        else:
            if score == max_score:
                best_plateaux.append(plateau_successeur_2)
            if score < min_score:
                best_score = score
                best_plateaux = [plateau_successeur_2]
                min_score = best_score
    if k == self.min_max_profondeur:
        cx = choice(best_plateaux)
        return cx, best_score
    return plateau, best_score

def valeur_min_max(self, joueur) -> int:
    autre = self.autre_joueur(joueur)
    return self.gagnants_possibles(joueur) - self.gagnants_possibles(
    autre)

def gagnants_possibles(self, joueur: str) -> int:
    compteur = 0
    autre = self.autre_joueur(joueur)
    for m in range(self.dimension):
        if self.nombre_pions_ligne((m, 0))[autre] == 0:
            compteur += 1
        if self.nombre_pions_colonne((0, m))[autre] == 0:
            compteur += 1
    if self.nombre_pions_diag_1((0, 0))[autre] == 0:
        compteur += 1
    if self.nombre_pions_diag_2((0, self.dimension - 1))[autre] == 0:
        compteur += 1
    return compteur
```

Listing 7 – Code de l'implémentation de l'IA avec min-max

Le code (**travail à rendre 4**) est dans le fichier `morpion_v4.py`. Il y a aussi une version interface dans `morpion_v4_interface.py`. Par défaut, le code lance un jeu *humain contre machine*, mais cela peut être modifié.

3.3.2 Analyse

Dans l'extrait 7, on modèle l'algorithme **Min-Max** récursivement. La fonction `min_max` démarre la recursion avec l'état initial, en appelant la fonction `min_max_rec`. Le `plateau` dans ces fonctions est défini comme une *tuple*, dont le premier élément sont las cases prises de chaque joueur, le deuxième sont las cases vides et le troisième est la dernière case placée dans la récursion précédente.

Dans chaque récursion, la fonction vérifie si la profondeur maximale à été atteint ($k = 0$). Dans ce cas, la **valeur du plateau** est calculée (`valeur_min_max`, $O(D^2)$) puis renviée. Similairement, si la fonction trouve un **état gagnant** (`gagnant`, $O(D)$), la récursion se termine et la fonction renvoie le tableau avec une **valeur infinie** (positive si le joueur gagne, négative si l'adversaire gagne). Ceux-ci correspondent aux **cas de base**.

Si la fonction n'a pas atteint la profondeur maximale ($k > 0$) et l'état n'est pas gagnant, elle cherche a faire une **récursion** avec **tous les coups possibles**. Pour les trouver, l'algorithme utilisée est similaire à celui de la stratégie **best-first** : si le joueur peut placer un nouveau pion, essayer avec toutes les cases libres (D^2), sinon, enlever un des pions, et le déplacer vers une case libre ($D \cdot D(D-2)$).

Ensuite, la fonction itère sur la liste contenant tous les **successeurs immédiats trouvés**, et fait un appel **récursive** à la fonction `min_max_rec` avec chaque plateau successeur. Cette nouvelle appel renvoie la **meilleure valeur trouvée par la récursion**. Dans le niveau (ou profondeur) actuel, nous devons trouver le **maximum** si le tour correspond à la machine, ou le **minimum** si le tour correspond à l'adversaire. À la fin du cycle, la fonction **renvoie la valeur trouvée**, sauf si on se trouve dans la première appel (profondeur $= k$), où la fonction doit **choisir un coup** définitif. Si plusieurs plateaux ont la même valeur maximale, la fonction en choisit un au hasard.

Comme on peut vérifier, **l'arbre de récursion s'étend de façon exponentielle** en fonction de k : chaque fois que l'algorithme progresse dans la profondeur, nous visitons environ D^3 nouveaux plateaux (donc D^{k3} branches quand la fonction est au $k = 0$). Encore, dans chaque niveau de profondeur la fonction doit faire des **calculs lourds**, comme vérifier la condition gagnante et trouver la valeur maximale ou minimale, et calculer la valeur du plateau au niveau final.

Les jeux machine contre machine donnent les résultats suivants (taille 3 et $k = 2$:

1. 97% : jeu nul
2. 3% : joueur qui commence gagne
3. 0% : joueur qui suive gagne

4 Conclusion

Dans les jeux comme celui analysé dans ce rapport, plus nous aurons d'informations avant d'agir, meilleur sera le résultat. Cependant, un problème se pose lorsque nous sommes confrontés au coût d'acquisition de ces informations. Quand l'espace des états atteignables est trop grand, faire l'analyse de tous les mouvements possibles devient déraisonnable, et nous devons recourir à limiter la quantité d'états que nous vérifions, ce qui au même temps affecte la qualité de la solution choisie. C'est pour cette raison qu'il est important de définir des heuristiques et des algorithmes intelligents (comme **Best-First** ou **Min-Max**) qui permettent de trouver une solution suffisamment décente et dans un délai raisonnable, selon les problèmes auxquels nous sommes confrontés.