

IA41

Intelligence artificielle : concepts
fondamentaux et langages dédiés

Projet

Sujet : Pogo

ALBRECHT Elise

JOVENIN Eileen

PERRIN Raphaël

SOMMAIRE

| | |
|---|----|
| ANALYSE DU PROBLÈME | 3 |
| STRUCTURE DE DONNÉES ET TRAITEMENTS ASSOCIÉS..... | 6 |
| Description de l'architecture du projet..... | 7 |
| Description de l'algorithme Min-Max..... | 8 |
| Description des variables d'états | 9 |
| Fin du jeu..... | 11 |
| DÉMONSTRATION | 11 |
| ANALYSE DES ACTIONS DE L'IA | 14 |
| AMÉLIORATIONS ENVISAGEABLES..... | 14 |
| ANNEXE | 15 |
| Méthode distance | 15 |
| Méthode move_to | 17 |

Analyse du problème

Pour permettre de faire une IA capable de jouer correctement au Pogo contre nous, il a d'abord fallu étudier les règles à respecter pour jouer :

- Le nombre de pièces déplacées peut varier, mais la distance de déplacement est corrélée au nombre de jetons joués.
- La couleur du joueur doit forcément être en haut de la pile avec laquelle il joue.
- Il n'y a pas de déplacement en diagonale, mais on peut faire un coude si on déplace 2 jetons.
- Il est possible de passer à travers les autres tours.
- Il n'y a pas de limite de hauteur.

Si les 2 derniers points sont simples à suivre car ils ne demandent pas de contrainte à rajouter, les 3 premières règles sont celles qu'il faudra coder, et sur lesquelles il faudra se pencher pour implémenter l'IA.

Après réflexion, comme le but ici est de tester plusieurs déplacements possibles et de choisir le meilleur, un algorithme Min-Max avec une fonction d'évaluation nous a paru être la méthode de résolution la plus adaptée, puisqu'une bonne fonction d'évaluation nous permettrait d'estimer le déplacement optimal, quelle que soit la situation de jeu.

La première étape de résolution a donc été de concevoir la fonction d'évaluation.

Pour ce faire, nous avons d'abord joué entre nous, et relevé des critères :

- **La prise ou la perte d'une pile**

Le but même du jeu est de « prendre » toutes les piles de l'adversaire. Il faut donc voir si notre action va nous faire gagner (dans le meilleur cas) ou perdre (le cas à éviter) une tour.

- **La taille de la pile prise et les déplacements possibles depuis cette dernière**

Gagner une pile est une bonne chose, mais dans le cas où on pourrait en prendre 2, mieux vaudrait avoir une pile depuis laquelle de nombreux déplacements sont possibles. En effet, si la composition de la tour nous permet de déplacer de nombreux jetons sans perdre cette dernière, c'est mieux.

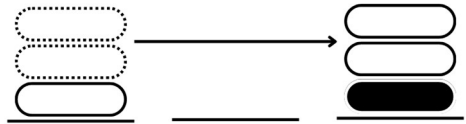
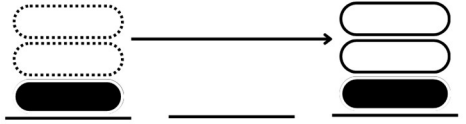
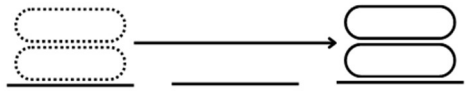
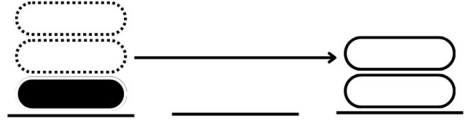
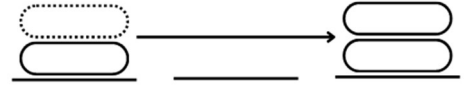
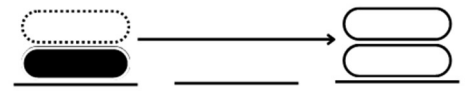
- **La sécurité de la pile prise/du jeton déplacé**

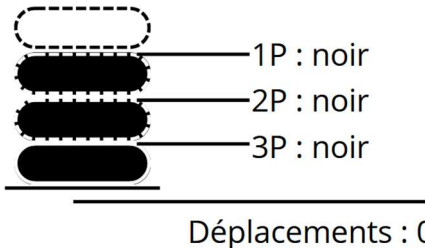
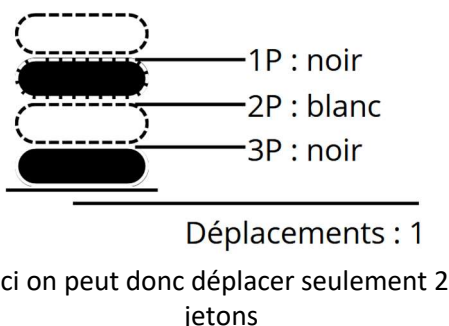
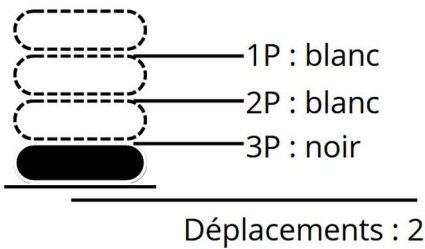
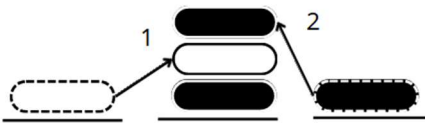

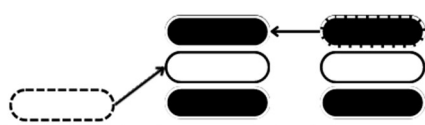
Il faut éviter de perdre des tours, ou que l'adversaire puisse trop facilement reprendre les jetons gagnés.

- **Fin du jeu**

Si une action nous permet de mettre fin au jeu, il faut à tout prix la réaliser (si on gagne) ou l'éviter (si on perd).

Par suite de ces observations, il nous restait encore à mettre des valeurs à toutes les actions/situations possibles :

| Critères | Description | Cas possibles | Valeur | Explications | Illustration (joueur blanc) |
|--------------------|---|--|--------|------------------------------------|---|
| 1 : Prise ou perte | Lors d'un déplacement, on regarde si on gagne une nouvelle pile et si on conserve celle depuis laquelle on joue | Prise sans perte | 10 | Meilleure action possible |  |
| | | Prise avec perte | -1 | Pas très intéressant |  |
| | | Pas de prise mais pas de perte | 0 | Ni intéressant ni contre-productif |  |
| | | Pas de prise et perte | -4 | Absurde |  |
| | | Déplacement sur la même couleur sans perte | -4 | Pas intéressant |  |
| | | Déplacement sur la même couleur avec perte | -10 | Absolument absurde |  |

| Critères | Description | Cas possibles | Valeur | Explications | Illustration (joueur blanc) |
|--------------------------------------|---|-----------------------------|--------|--|---|
| 2 : Nombre de déplacements possibles | Les déplacements possibles depuis la nouvelle pile sont intéressants, et ils dépendent des jetons de notre couleur dans la pile | 0 déplacement | 0 | Impossible de jouer avec la pile sans la perdre |  |
| | | 1 déplacement | 1 | On peut déplacer uniquement 1, 2 ou 3 jetons |  |
| | | 2 déplacements | 2 | Plusieurs tailles sont possibles |  |
| 3 : Reprise immédiate | Est-ce que l'adversaire est en mesure de récupérer immédiatement la pile | Oui | -1 | Peu intéressant |  |
| | | Non | 2 | Très intéressant |  |
| | | Oui mais en perdant sa pile | 1 | Intéressant : il y a de grandes chances que l'adversaire ne le fasse pas |  |

| Critères | Description | Cas possibles | Valeur | Explications | Illustration (joueur blanc) |
|----------------|--------------------------|--------------------|--------|-----------------------------------|---|
| 4 : Fin du jeu | Est-ce un coup décisif ? | On gagne | 100 | Objectif | Toutes les piles ont un jeton blanc au-dessus |
| | | On perd | -100 | Ne surtout pas faire cette action | Aucune pile n'a de jeton blanc au-dessus |
| | | Ni l'un ni l'autre | 0 | On continue | Ni l'un ni l'autre |

Voici un exemple avec un cas de jeu concret. Le(s) jeton(s) déplacé(s) sont en rouge, et on joue donc le joueur blanc.

| | | | | | | | |
|---------------------------|---|----|----|----|----|------|------|
| 1) Prise / perte pile | 0 | 10 | -4 | -1 | -1 | 0 | 0 |
| 2) Déplacements possibles | 1 | 2 | 2 | 1 | 2 | 2 | 2 |
| 3) Reprise immédiate | 1 | 1 | 1 | 2 | 1 | -1 | -1 |
| 4) Fin du jeu | 0 | 0 | 0 | 0 | 0 | -100 | -100 |
| TOTAL | 2 | 13 | -1 | 2 | 2 | -99 | -99 |

Illustration de l'heuristique du jeu

Structure de données et traitements associés

Description de l'architecture du projet

- Les jetons

Afin de pouvoir manipuler facilement les jetons et de pouvoir réaliser des opérations simples entre eux, nous avons décidé de décomposer notre jeu ainsi :

- Un jeu possède des tours, nous avons donc créé une liste de tours nommée `towers`.
- Chaque tour possède des jetons, nous avons donc créé une liste de jetons nommée `tower`.
- Chaque jeton (= pawn) est une instanciation de la classe `Pawn`, possédant une position `[x, y]` et une couleur.
- La liste `towers` est donc une liste de liste (il s'agit d'une liste de tours).

- La grille

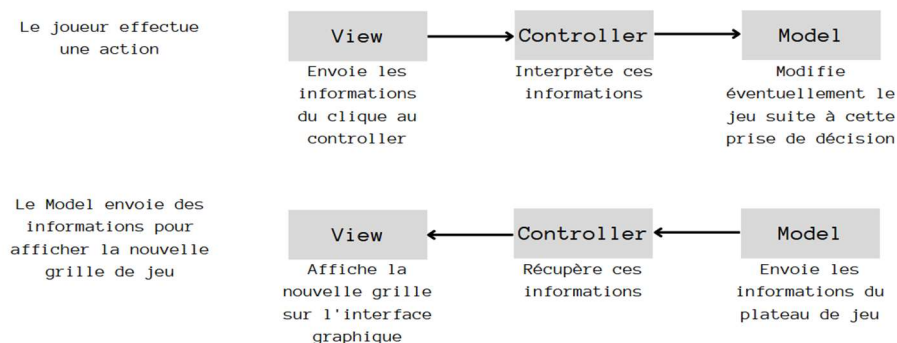
Nous avons décidé de représenter la grille 3*3 sous la forme d'une liste de tours afin de réaliser les manipulations plus aisément, en ajoutant et supprimant des éléments de cette dernière dynamiquement. Les opérations à l'intérieur de cette liste sont assurées grâce aux coordonnées des jetons. Ces coordonnées sont importantes, nous les utilisons par exemple pour calculer la distance de déplacements des jetons, grâce au théorème de Pythagore, ou lorsque nous souhaitons comparer deux emplacements de tours entre-elles.

Il est important de noter qu'une tour ne possède pas de coordonnées à proprement parler dans notre programme. Cependant, il est clair que le jeton du dessus possèdera toujours nécessairement les mêmes coordonnées que la tour elle-même. C'est par cette réflexion que nous avons décidé de déterminer la position d'une tour grâce aux coordonnées du premier jeton de la tour : `tower[0]`.

- Architecture VMC (View Model Controller)

Nous avons souhaité implémenter une base d'architecture VMC. Cette dernière permet ainsi de ne pas mélanger l'interface graphique, qui se contente seulement d'être une vue pour l'utilisateur, avec le model, qui correspond au code pur du programme, dont l'intermédiaire entre les deux est assuré par la classe `Controller`.

Ainsi :



Comme énoncé précédemment, la partie IA ne s'intéresse pas réellement à ce système de structure, étant donné qu'il n'est pas nécessaire pour cette dernière d'interagir avec l'interface graphique pour cliquer d'éventuelles positions.

Description de l'algorithme Min-Max

Nous avons implémenté un algorithme Min-Max comme nous avons pu le voir lors de nos séances de CM.

Cet algorithme se situe dans une classe nommée MinMax.py.

Quelques informations concernant les méthodes `min_value` et `max_value` :

```
def max_value(self, node, depth):
    self.model.force_turn(self.ia, True)

    if depth != 0:
        self.ia.determine_states(node)

    if node.children == [] or depth == 0:
        return [node.evaluation(False), node]

    return_state = [-10000000, None]

    for child in node.children:
        temp_tab = self.min_value(child, depth - 1)
        if temp_tab[0] >= return_state[0]:
            return_state[0] = temp_tab[0]
            return_state[1] = temp_tab[1]

    return return_state
```

Nous avons fait le choix de ne pas seulement manipuler directement des entiers MAX et MIN pour notre algorithme. Car nous avons un problème associé : Lorsque l'IA déterminait le meilleur coup, il fallait qu'elle puisse retracer le chemin complet de l'arbre associé à ce coup précis.

Afin de remédier à ce problème, nous avons créé un couple [valeur, État], qui permet ainsi de continuer à manipuler des entiers pour effectuer les opérations Min/Max, tout en gardant en mémoire l'origine de l'état intéressé.

Finalement, notre algorithme ne renvoie donc pas seulement un entier, mais un couple [entier, État]

Intéressons-nous ainsi à la méthode `determine_states(node)`, qui se situe dans la classe IA :

```
def determine_states(self, decided_state):
    self.determine_all_towers(decided_state.towers)

    for tower in self.towers_to_examine:
        for dx in range(3):
            for dy in range(3):
                distance = self.model.distance(tower[0].x, tower[0].y, dx, dy)

                if len(tower) >= distance != - 1:
                    child = state.State(self.model, self, dx, dy, distance,
                                         False)
                    decided_state.add_child(child)
                    child.set_prev_tower(tower)
                    child.set_father(decided_state)
                    child.set_hierarchy(child.father.depth + 1)

                    self.model.ref = tower
                    child.towers = self.model.decide_type_of_moving(dx, dy,
                                                                    distance, decided_state.towers, False)

                    child.determine_new_tower()

    self.towers_to_examine.clear()
```

À chaque fois que nous arrivons sur une nouvelle profondeur de l'arbre MinMax, il est nécessaire de déterminer les fils possibles de chaque nœud. C'est pourquoi cette méthode étudie tous les mouvements possibles que les tours déterminées peuvent réaliser.

- Ces tours, dont le tableau est nommé `towers_to_examine`, sont déterminées grâce à la méthode `determine_all_towers()`, qui ajoute dans ce tableau toutes les tours dont la couleur du jeton le plus haut (donc celle dont le joueur peut jouer) correspond au tour du joueur actuel.

- La méthode `determine_states()` parcourt tout le tableau du jeu (de taille 3 x 3) et vérifie si la distance et la taille de la tour concordent afin de pouvoir se déplacer à la case désirée du tableau.
- Si la tour a la capacité de se déplacer sur une nouvelle case, nous ajoutons un nouvel état fils à l'état initial.
- Nous effectuons ce processus jusqu'à ce qu'il n'y ait plus de mouvements possibles à partir de ce nœud d'état.

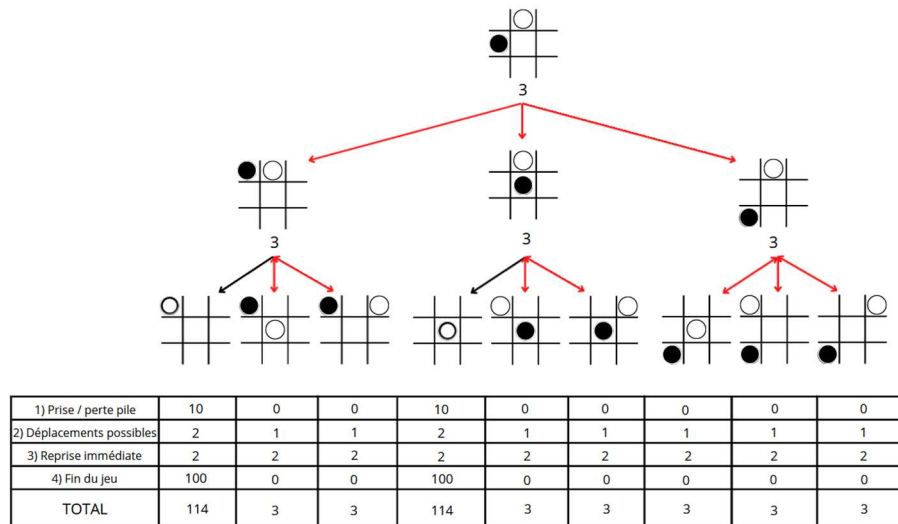


Illustration de Min-Max avec évaluation

Description des variables d'états

Nous avons vu précédemment que chaque état père possède éventuellement des états fils. L'état initial de l'arbre, c'est-à-dire sa racine, ne possède aucune information, à part le tableau actuel du jeu, sa hiérarchie (c'est-à-dire sa hauteur dans l'arbre) et l'attribut `root = True`.

```
decided_state = state.State(self.model, self, None, None, None, True)
decided_state.towers = self.model.towers
decided_state.set_hierarchy(0)
```

Cela s'explique tout simplement, car comme mentionné précédemment, la racine de l'arbre est la racine qui doit être déterminée à la fin de l'algorithme Min Max.

Pour le cas des fils, nous ajoutons différentes informations :

```
child = state.State(self.model, self, dx, dy, distance, False)
decided_state.add_child(child)
child.set_father(decided_state)
child.set_hierarchy(child.father.depth + 1)
child.towers = self.model.decide_type_of_moving(dx, dy, distance,
decided_state.towers, False)
```

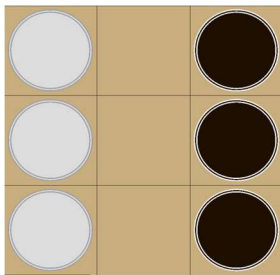
Les premiers paramètres `self.model` et `self` servent seulement à garder une référence vers les instanciations du model et d'une IA.

Ainsi, un état possède :

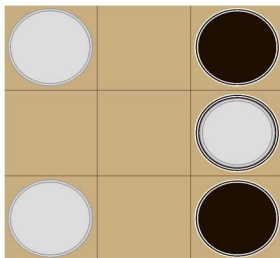
- Une position `[x, y]` qui correspond à la nouvelle position de la tour grâce à `dx` et `dy` de la méthode `determine_states()` de la classe IA.
- Une distance qui correspond à la distance qui a été parcourue pour passer d'un état à l'autre
- D'un booléen `True` ou `False` qui indique si oui ou non ce nouvel état est une racine. Dans le cas de fils, il est nécessairement mis à `False`.
- Un père : cet état père est ajouté grâce à la méthode `set_father()`, permettant de connaître le père de l'état et ainsi effectuer des opérations avec ce dernier.
- Comme tout nœud, un état possède une hiérarchie.
- Et enfin le plus important, un état possède un nouveau tableau de jeu ayant évolué à la suite d'un coup.
- Une tour qui sera déterminée à nouveau grâce à ce nouveau tableau de jeu.

Prenons pour exemple :

État initial avec comme état père la tour blanche positionnée en `x = 0` et `y = 1`:



L'un des états fils peut être celui-ci (bien-sûr, il en existe d'autres) :



Nous pouvons ainsi déterminer les informations suivantes du nouvel état fils :

- `x = 2` et `y = 1`.
- Distance = 2.
- `root = False`, car il s'agit d'un état fils.
- `self.father` = tour précédente, c'est-à-dire la tour [jeton blanc, jeton blanc] de coordonnées `[0, 1]`.
- Hiérarchie = 1, en considérant que nous sommes à la première étape de détermination des nœuds.
- `self.towers` = nouveau tableau de tours du jeu, nous n'allons pas ici tout écrire, mais il s'agit d'un tableau comprenant toutes les tours de l'image juste ci-dessus.
- Ainsi, `self.tower` = [jeton blanc, jeton blanc, jeton noir, jeton noir] qui est déterminée grâce au coup donné et le nouveau tableau de jeu.

Fin du jeu

Afin de déterminer une fin de jeu possible, il suffit de parcourir toutes les tours du jeu. Si aucune tour ne possède de jeton de couleur noire ou blanche sur le dessus, c'est que telle ou telle couleur a gagné.

```
def check_win(self):
    white = False
    black = False
    for t in self.towers:
        if t[0].color == "white":
            white = True
        if t[0].color == "black":
            black = True

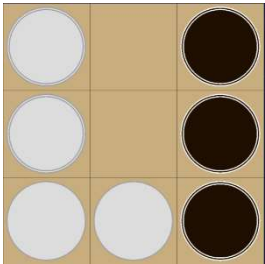
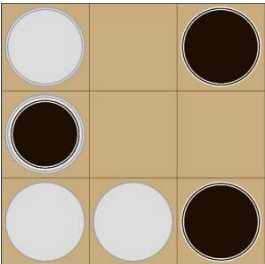
    if not white:
        self.winner = "Black"
    elif not black:
        self.winner = "White"
```

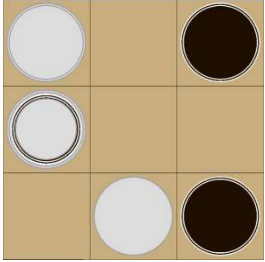
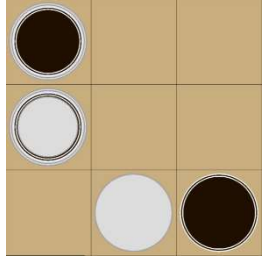
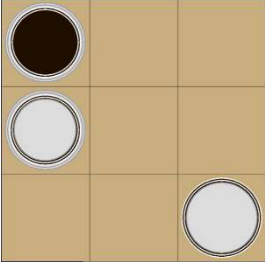
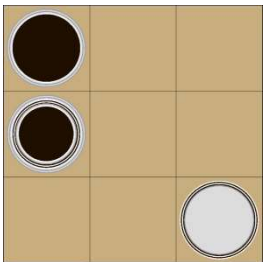
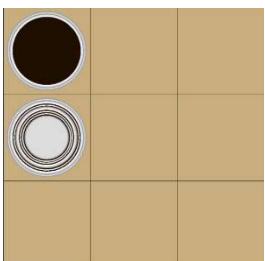
Cela peut ainsi se coder aisément en initialisant deux variables de couleur en False. Et à partir du moment où un joueur peut déplacer une tour, nous la renseignons en True.

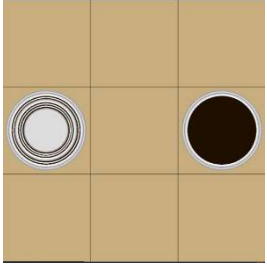
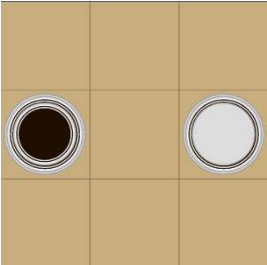
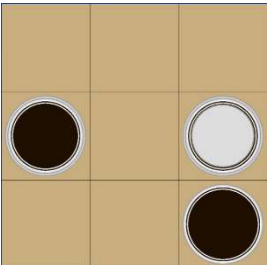
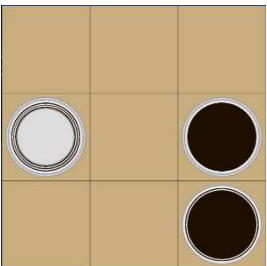

A la fin, si une variable est toujours False, cela signifie qu'un joueur a gagné.

Démonstration

Maintenant, analysons une partie humain/IA, cette dernière étant les pions noirs.

| Jeu | Commentaires |
|---|--|
|  | On joue le blanc |
|  | Il choisit de prendre la tour adverse. Il en prend une sans perdre, c'est une bonne action |

| | |
|---|---|
|  | <p>On décide de la récupérer</p> |
|  | <p>Il prend la tour du haut</p> |
|  | <p>On prend celle d'en bas</p> |
|  | <p>Il prend celle du milieu gauche</p> |
|  | <p>On peut la récupérer, et on sait qu'il ne peut plus la récupérer tout de suite</p> |

| | |
|---|--|
|  | <p>Ici, le coup est intéressant : il s'est déplacé de 3 pour ne pas perdre sa tour (noir-blanc-blanc) Et il a choisi de se mettre ici car si nous on en déplace 2 on perd notre tour</p> |
|  | <p>C'est ce qu'on a fait : déplacer 2 jetons pour prendre sa tour nous fait perdre la première</p> |
|  | <p>Il en déplace 3 pour ne pas la perdre</p> |
|  | <p>On récupère la tour de gauche</p> |
|  | <p>Il gagne</p> |

Analyse des actions de l'IA

On peut constater que la priorité de l'IA est de prendre des tours sans en perdre, ce qui est le but du jeu. Cela est prévisible, car c'est la plus grande valeur donnée dans la fonction d'évaluation.

Au début elle priorise donc les déplacements de 2 jetons pour prendre les tours en face. Ensuite, On voit qu'elle essaie toujours de prendre des tours, et que lorsqu'elle ne peut pas elle se met au moins dans une situation où nous ne pouvons pas non plus lui reprendre des tours (du moins sans en perdre une à nous).

La fonction d'évaluation et le MinMax fonctionnent donc bien ; l'IA évite soigneusement les cas de fin de jeu où elle perdrait, et ses mouvements sont corrects pour faire avancer le jeu et qu'elle gagne.

Améliorations envisageables

Au tout début du projet, le parcours de l'arbre se faisait en recopiant toutes les listes du jeu, autant de fois qu'il le fallait, pour tout parcourir. Ce qui était très long en temps d'exécution (d'autant plus si les possibilités d'action étaient nombreuses). Une première amélioration a donc été de travailler depuis les listes directement, sans les copier. Le temps de réaction de l'IA a chuté par 4, fluidifiant beaucoup le jeu.

D'autres améliorations sont néanmoins possibles, notamment sur l'algorithme MinMax, qui est ici est un "simple" algorithme MinMax; il n'a pas d'élagage Alpha-Bêta.

Nous avons également songé à modifier la structure de donnée initiale, en prenant un tableau à 2 dimensions pour la grille, et en donnant ainsi les coordonnées immédiatement (pas dans chaque pion), voire pas du tout, en ne travaillant qu'avec les indices du tableau.

La fonction d'évaluation peut sans doute être également améliorée, car elle obtient souvent des résultats similaires, mais nous pensons avoir couvert les points majeurs du jeu.

ANNEXE

Méthode distance

```
def distance(self, x, y, dx, dy):
    dist = math.sqrt(math.pow((dx - x), 2) + math.pow((dy - y), 2))
    if x == dx and y == dy:
        return -1
    if dist == 1:
        return 1
    elif dist == math.sqrt(math.pow(1, 2) + math.pow(2, 2)):
        return 3
    elif dist < math.sqrt(math.pow(2, 2) + math.pow(2, 2)):
        return 2
    return -1
```

La méthode distance de la classe Model est très importante dans notre système de résolution. En effet, c'est cette dernière qui permet de calculer la distance entre deux tours grâce à deux points de coordonnées différentes.

Il est nécessaire au début de cette méthode de vérifier que les coordonnées $[x, y]$ soient différents de $[dx, dy]$. En effet, nous utilisons la relation suivante afin de calculer la distance entre deux points :

La **distance AB** est donnée par la formule $\sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$.

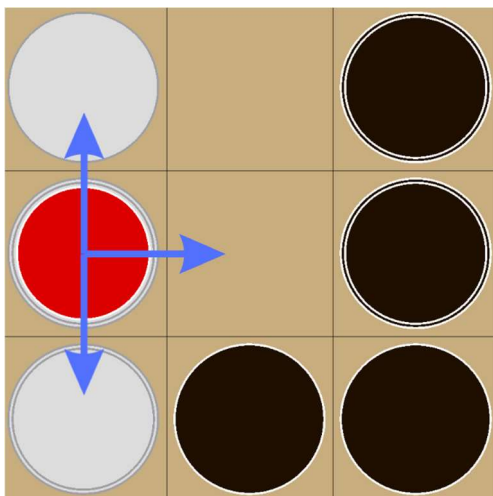
Or, si $x = dx \Rightarrow (dx - x)^2 = 0^2 = 0$

et si $y = dy \Rightarrow (dy - y)^2 = 0^2 = 0$

$\Rightarrow \sqrt{0 + 0} = 0 \neq 1$

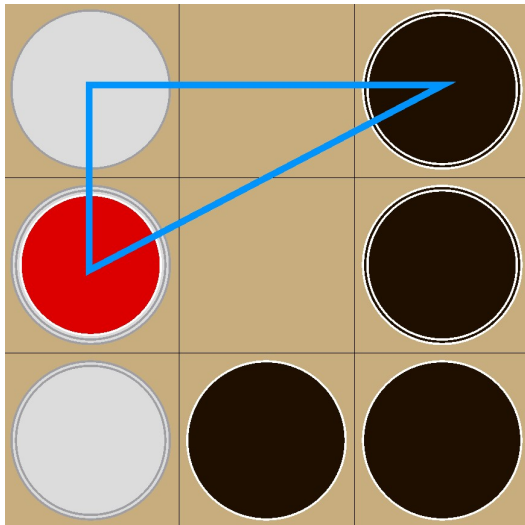
Ce qui est absurde, car lorsque les coordonnées sont identiques, nous voulons considérer qu'il s'agit d'une distance parcourue de 0, ce qui n'est pas le cas ici.

Également, dès lors que nous avons calculé cette distance, il est nécessaire de réfléchir à plusieurs situations possibles :

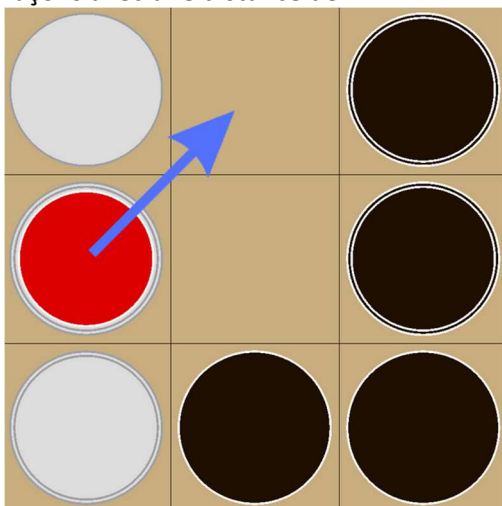


Dès lors où la distance calculée est de 1, c'est que le jeton se déplace d'une distance de 1 nécessairement, car ce dernier n'a pas l'autorisation de se déplacer en diagonale.

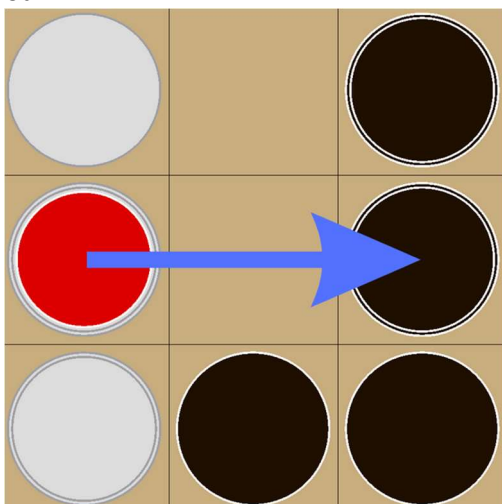
De plus, le seul moyen de se déplacer de 3 cases est sous forme d'hypoténuse d'un triangle rectangle de cotés 1 et 2 :



Une fois que nous connaissons les déplacements relatifs à 1 et 3 de distance, il est plus facile de déterminer celui caractéristique d'un déplacement de 2, car nous pouvons nous déplacer de deux façons avec une distance de 2 :



ou



Ce déplacement est déterminé grâce aux anciennes vérifications, s'il ne s'agit pas d'une distance de 1 ou de 3, alors on vérifie simplement que la distance n'est pas supérieur à la diagonale complète du plateau. En effet, la diagonale complète du plateau est la distance maximale possible sur le jeu, qui représente une distance de 4 (hypoténuse d'un triangle rectangle de taille 2 par 2). Or, la distance de déplacement maximale autorisée est de 3. Donc si cette distance calculée est inférieure à la diagonale maximale, nous considérons qu'il s'agit d'une distance de 2.

Autrement, la méthode renvoie -1. Ainsi, dès lors où le joueur effectue un déplacement impossible (distance de 4) ou un déplacement absurde (0 de distance, donc le joueur clique sur la même position que la position initiale du jeton), -1 est renvoyé.

Méthode move_to

```
def move_to(self, amount, x, y, tower, towers, is_free, player_):
    if not player_:
        tower = deepcopy(tower)
        towers = deepcopy(towers)
        for t in towers:
            if self.ref[0].x == t[0].x and self.ref[0].y == t[0].y:
                self.ref = t

    for i in range(amount):
        self.ref[0].x = x
        self.ref[0].y = y

        self.pawns.append(self.ref[0])
        self.ref.pop(0)

    if not is_free:
        self.pawns += tower
        for t in towers:
            for p in t:
                if p.x == tower[0].x and p.y == tower[0].y:
                    if t in towers:
                        towers.remove(t)

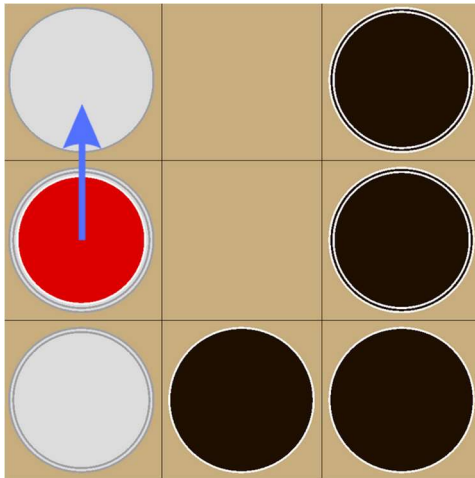
        for p in self.pawns:
            tower.append(p)
        towers.append(self.pawns)

    if not self.ref:
        towers.remove(self.ref)

    if is_free:
        towers.append(self.pawns)
    self.pawns = []
    self.ref = None

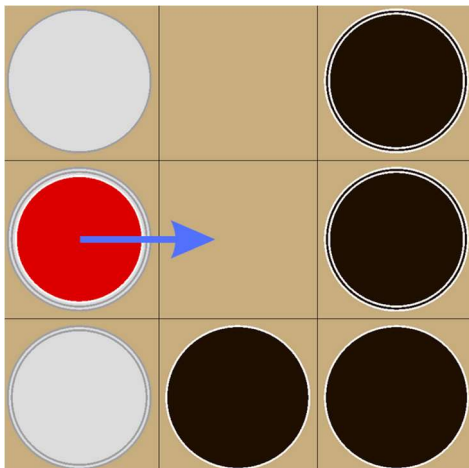
    return towers
```

Cette méthode est aussi importante à évoquer, elle découle d'une méthode précédente nommée `decide_type_of_moving()`, dont cette dernière décide si le déplacement de la tour décidée par le joueur s'effectuera sur une tour déjà présente, ou sur une case vide. Prenons l'exemple suivant :



Dans cette situation, le joueur décide de se déplacer sur une tour déjà existante, la méthode `decide_type_of_moving()` appellera ainsi la méthode `move_to` en ajoutant en paramètres `is_free = False`

Dans un autre cas comme celui-ci :



Le joueur fait le choix de se déplacer sur une case du plateau dont il n'y a aucune tour. La méthode `move_to` prendra donc en paramètres `is_free = True`

Informations sur certains paramètres :

- `amount` : Distance parcourue entre l'ancienne position de la tour et sa nouvelle position.
- `x, y` : la position de la nouvelle tour créée.
- `tower` : La tour déjà présente sur laquelle le(s) jeton(s) souhaite(nt) se déplacer. Dans le cas où la tour se déplace sur une zone vide, `tower` est vide également.
- `towers` : Ensemble des tours présentes sur le jeu.
- `player_` : Cet attribut permet de spécifier qu'il s'agit d'un mouvement fait par un joueur, et non une IA. Ce dernier est important, car le joueur interagit directement avec les tours actuelles du jeu, tandis que l'IA interagit avec les tours qui sont disposées telles qu'elles le sont à un niveau de parcours de l'arbre. Cela signifie qu'il ne faut pas modifier directement le plateau de jeu actuel, mais effectuer une copie à chaque nœud de l'arbre.

Ainsi, afin de manipuler le plateau de jeu, il est nécessaire de garder une référence sur la tour qui souhaite être déplacée : elle est stockée dans la variable `self.ref`.

```
for i in range(amount):
    self.ref[0].x = x
    self.ref[0].y = y

    self.pawns.append(self.ref[0])
    self.ref.pop(0)
```

Etant donné que l'on enlève seulement le nombre de jetons en rapport avec la distance parcourue, nous modifions tout d'abord la position des jetons en rapport avec leur nouvelle destination à chaque itération. À chaque itération, il est aussi nécessaire de supprimer le jeton de l'ancienne tour, mais de tout de même les stocker dans un tableau nommé `self.pawns` pour ne pas perdre leur trace.

```
if not is_free:
    self.pawns += tower
    for t in towers:
        for p in t:
            if p.x == tower[0].x and p.y == tower[0].y:
                if t in towers:
                    towers.remove(t)

    for p in self.pawns:
        tower.append(p)
    towers.append(self.pawns)
```

Dans le cas où notre destination n'est pas libre, il est seulement nécessaire de concaténer la nôtre nouvelle liste créée précédemment avec la tour de la destination. Cela crée ainsi une nouvelle liste possédant les jetons déplacés de l'ancienne tour + les jetons de la tour destinatrice.

Il ne reste plus qu'à supprimer cette ancienne tour qui possède les mêmes coordonnées que la nouvelle tour créée de la liste des tours du jeu, et d'ajouter cette nouvelle tour en échange.

```
if not self.ref:
    towers.remove(self.ref)
```

Cette vérification permet de s'assurer qu'il n'y ait aucune liste de tours vides dans le jeu.

```
if is_free:
    towers.append(self.pawns)
```

À l'inverse, si la destination ne possède aucune tour, nous ajoutons simplement cette nouvelle tour dans la liste de tours.