IA & RÉSOLUTION DE CONTRAINTES

# L'escampe

Maxime Fromont - Kevin Leborgne - Marin Sebirot

# Détails du projet

| Informations général | |
|---|---|
| Ecole | Polytech Paris Saclay |
| Module | IA & Résolution de contraintes |
| Professeur | MA Yue |
| Groupe | Maxime F. Kevin L. Martin S. |

| Code source |
|---|
| https://github.com/maximefromont/IA-constraints |

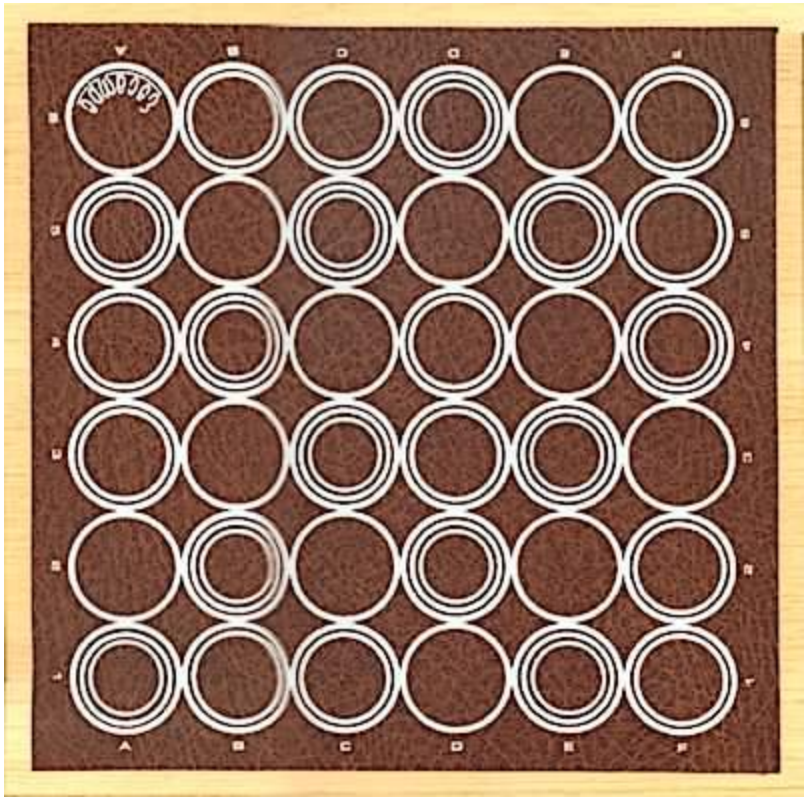
# Introduction

L'objectif au travers de ce projet était de réaliser une IA dont l'objectif est de jouer au jeu de l'escampe. Le jeu d'escampe est un jeu de plateau dont le jeu est directement contraint par le coup précédent de l'adversaire. Les règles de ce jeu sont simples :

1. Le joueur Noir choisit un bord du plateau (haut ou bas), et place comme il le souhaite ses pièces sur les deux premières lignes de ce bord.
2. Le joueur Blanc pose ensuite ses propres pièces sur les deux premières lignes du bord opposé. C'est lui qui joue en premier, en choisissant librement la pièce qu'il désire déplacer.
3. Pendant le reste de la partie, chaque joueur peut à son tour bouger sa licorne ou l'un de ses paladins, en devant toutefois respecter la consigne suivante : la pièce jouée doit partir d'une case ayant le même liseré que celle sur laquelle l'autre joueur a posé sa propre pièce au tour précédent.
4. En outre, le liseré du cercle de départ détermine le nombre de cases que la pièce doit parcourir : une case si le liseré est simple, deux cases s'il est double, et trois

cases s'il est triple. Lors d'un déplacement, il est interdit de faire passer la pièce par une case occupée, ou de la faire passer deux fois par la même case. Les mouvements en diagonal ne sont pas autorisés.

5. Si un joueur ne peut bouger aucune pièce, il saute son tour. L'autre joueur peut alors jouer la pièce de son choix.
6. La partie se termine quand l'un des joueurs prend la licorne de son adversaire avec l'un des ses paladin. Il s'agit du seul type de prise possible, les paladins étant imprenables.



*Exemple de plateau d'escampe*

Afin de réaliser cette IA, différentes contraintes et éléments nous ont été fournis :

- Le programme doit être réalisé est en Java
- Un JAR correspondant à un serveur de jeu ainsi qu'une classe correspondant à un client de jeu ont été fournées afin de réaliser l'IA

# Modélisation du jeux

## Modélisation du plateau

Pour modéliser le jeu Escampe, nous avons utilisé trois classes principales : `Pion`, `Case`, et `EscampeBoard`. Ces classes permettent de représenter de manière structurée les éléments essentiels du jeu, à savoir les pièces, les cases du plateau et l'état global du jeu.

### Classe Pion

La classe `Pion` représente une pièce sur le plateau. Elle contient des informations sur le type de la pièce (licorne ou paladin) et l'équipe à laquelle elle appartient (noir ou blanc). Ces informations sont stockées à l'aide d'énumérations pour garantir une gestion efficace et éviter les erreurs.

**Avantages :**

- Utilisation des énumérations pour une gestion claire et sécurisée des types et équipes.
- Encapsulation des informations sur les pièces.

### Classe Case

La classe `Case` représente une case du plateau. Chaque case a une valeur de liseré (simple, double, ou triple) qui influence les mouvements des pièces, et peut contenir un pion s'il est positionné dessus.

**Avantages :**

- Représentation claire des propriétés de chaque case.
- Facilite le calcul des mouvements en fonction du liseré.

### Classe EscampeBoard

La classe `EscampeBoard` contient un tableau de cases représentant le plateau de jeu et des variables d'état pour gérer la progression de la partie. Elle inclut les méthodes nécessaires pour manipuler le plateau, vérifier la validité des coups, et gérer l'état du jeu.

Nous avons décidé de représenter notre plateau sous forme d'un tableau de dimensions double, composé d'objets nommés `Case.` Ces `Cases` contiennent une valeur de liseret fixé au moment de leur instanciation, ainsi qu'une information sur la présence ou non de pion.

La taille du tableau est donc de 6 par 6 `Cases`. Chaque `Case` est instancié de manière formelle, afin de transmettre chaque liseret, en se référant aux règles du jeu de l'Escampe.

```
//Initializing the board with straight values from the subject
boardArray = new Case[][]{
        {new Case( value: 1), new Case( value: 2), new Case( value: 2), new Case( value: 3), new Case( value: 1), new Case( value: 2)},
        {new Case( value: 3), new Case( value: 1), new Case( value: 3), new Case( value: 1), new Case( value: 3), new Case( value: 2)},
        {new Case( value: 2), new Case( value: 3), new Case( value: 1), new Case( value: 2), new Case( value: 1), new Case( value: 3)},
        {new Case( value: 2), new Case( value: 1), new Case( value: 3), new Case( value: 2), new Case( value: 3), new Case( value: 1)},
        {new Case( value: 1), new Case( value: 3), new Case( value: 1), new Case( value: 3), new Case( value: 1), new Case( value: 2)},
        {new Case( value: 3), new Case( value: 2), new Case( value: 2), new Case( value: 1), new Case( value: 3), new Case( value: 2)}
};
```

*Note : Vous trouverez un diagramme de classe complet du projet à la fin de ce rapport.*

**Avantages :**

- Intégration de toutes les fonctionnalités nécessaires pour gérer le jeu.
- Structuration claire permettant de manipuler facilement le plateau et les états du jeu.

Cette modélisation nous permet de représenter efficacement le jeu Escampe en respectant les règles et les contraintes spécifiques du jeu. La séparation en classes distinctes permet une gestion modulaire et facilite les extensions futures pour l'IA et d'autres fonctionnalités.

## Modélisation de l’état de la partie

Pour déterminer l'état complet du jeu Escampe, nous avons étendu la classe `EscampeBoard` avec les attributs suivants :

1. **lastLisere : int**
    - Cet attribut enregistre le liseré de la dernière case sur laquelle un joueur a placé sa pièce. Il est initialisé à -1 pour indiquer qu'aucun coup n'a encore été joué.
    - **Utilité :** Permet de contrôler le respect de la règle selon laquelle chaque joueur doit placer sa pièce sur une case ayant le même liseré que celui de la case précédemment jouée par l'adversaire.
2. **BlackTeamWin : boolean**
    - Cette variable indique si l'équipe noire a remporté la partie.
    - **Utilité :** Permet de déterminer si la condition de fin de partie a été atteinte avec la victoire de l'équipe noire.
3. **WhiteTeamWin : boolean**
    - Cette variable indique si l'équipe blanche a remporté la partie.
    - **Utilité :** Permet de déterminer si la condition de fin de partie a été atteinte avec la victoire de l'équipe blanche.
4. **move : ArrayList<Move>**
    - Cette liste enregistre les mouvements effectués tout au long de la partie, sous forme d'objets `Move`.
    - **Utilité :** Permet de garder une trace de l'historique des mouvements pour une éventuelle relecture de la partie ou pour l'analyse stratégique postérieure.

Ces attributs étendus garantissent que la classe `EscampeBoard` est en mesure de conserver l'état complet du jeu et de surveiller l'évolution de la partie. Ils fournissent également une base solide pour implémenter des fonctionnalités supplémentaires telles que la détection de la fin de partie et la gestion des victoires des équipes noire et blanche.

## Découverte des coups possible

### Détermination des hit-Map

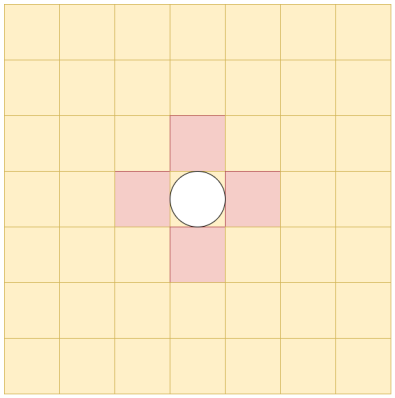
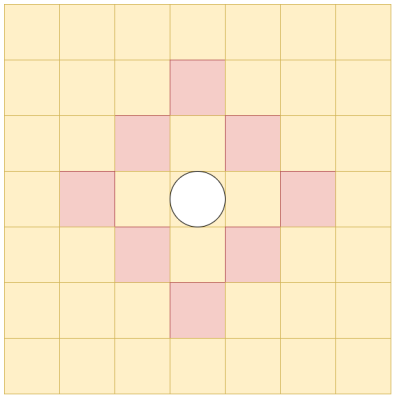
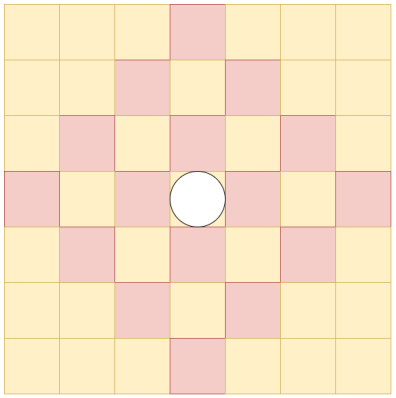
Avant d'implémenter la fonction pour déterminer les mouvements possibles des pièces dans le jeu Escampe, nous avons pris le temps d'étudier les différents mouvements possibles et avons établi des hitmaps pour chaque type de pièce en fonction du liseré sur lequel elle se trouve. Voici une explication plus détaillée de cette approche :

- Étude des mouvements possibles :

Nous avons analysé en détail les différentes possibilités de mouvement pour chaque type de pièce (licorne et paladin) en fonction du liseré sur lequel elle est positionnée. Cette étude a impliqué l'exploration de toutes les cases accessibles à partir de chaque position possible d'une pièce, en tenant compte des contraintes telles que le type de liseré et les règles de déplacement spécifiques à Escampe.

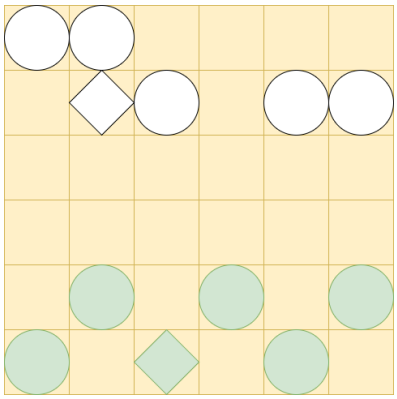
- Établissement des hitmaps :

Sur la base de cette analyse, nous avons construit des hitmaps pour chaque type de pièce (licorne et paladin) en fonction du liseré sur lequel elle se trouve. Une hitmap est une représentation graphique qui indique, pour chaque case du plateau, si elle est accessible à partir de la position actuelle d'une pièce en fonction du liseré de la case. Dans notre cas, nous avons établi des hitmaps pour un plateau de 7x7, ce qui permet de couvrir la gamme complète de mouvements possibles pour les pièces sur des liserés de valeur 3.

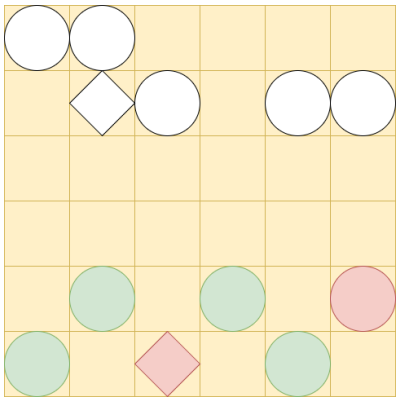
| | | |
|---|---|---|
|  |  |  |
| hit-map 1 liseré | hit-map 2 liseré | hit-map 3 liseré |

## Algorithme pour déterminer les coups possible

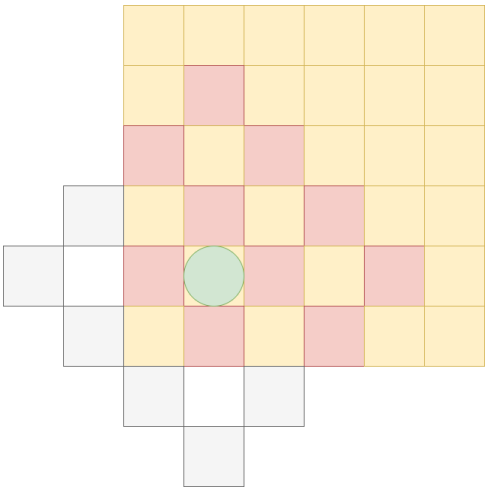
On entame la collecte des pions du joueur actuel, identifiant ainsi toutes les pièces lui appartenant, qu'il s'agisse des licornes ou des paladins.



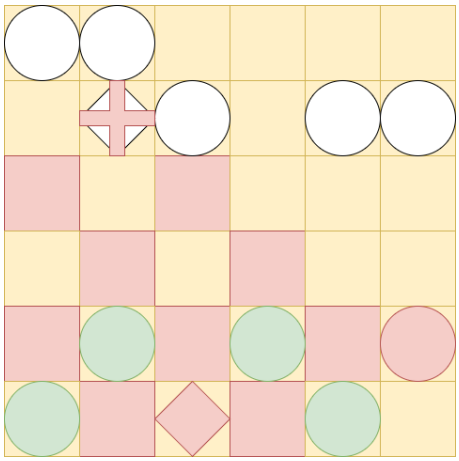
Ensuite, on filtre les pièces qui sont en mesure de bouger, en tenant compte des règles de jeu, telles que la nécessité de déplacer une pièce du même liseré que le coup précédent de l'adversaire.



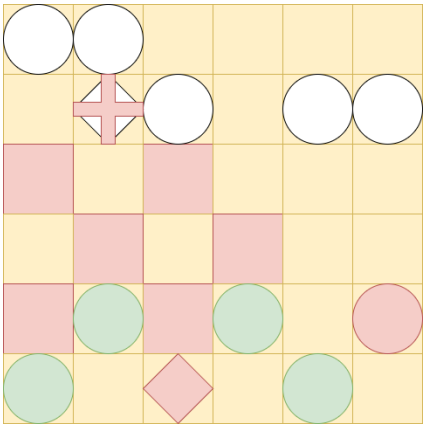
En appliquant la hit-map préalablement établie aux positions de chaque pièce, on superpose ainsi les possibilités de mouvement en fonction du type de liseré sur lequel se trouve la pièce.



Ensuite les cases où un paladin adverse est positionné sont alors éliminées, puisque les mouvements ne peuvent pas passer par une case occupée.



Pour conclure cette phase, on procède à une vérification approfondie pour s'assurer que les chemins de mouvement sont dégagés. Cela implique de vérifier si les cases intermédiaires entre la position actuelle de la pièce et sa destination sont vides.



## Gestion de fin de partie

Pour gérer la fin de partie, à chaque fois qu'un coup est joué, nous testons si ce coup est une prise. Si c'est le cas, nous changeons la variable d'état associée pour le joueur gagnant.

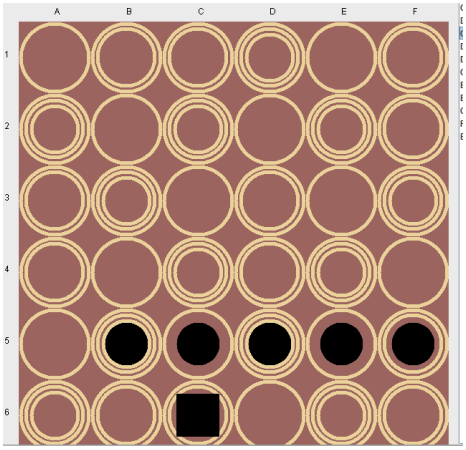
De cette manière, à chaque prise effectuée par un joueur, nous mettons à jour la variable d'état appropriée pour indiquer que ce joueur a remporté la partie. Cela permet de suivre en temps réel l'évolution de la partie et de déterminer le vainqueur dès qu'une prise décisive est réalisée.

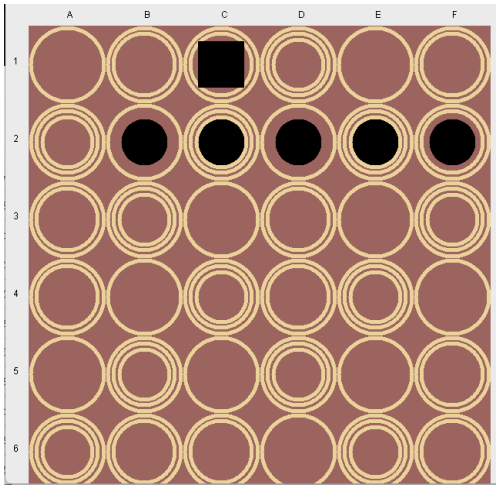
## Heuristique du jeux

Tout d'abord, avant de pouvoir débuter une partie, il faut choisir une configuration de départ pour notre IA. En effet, réaliser l'arbre de toutes les possibilités de première configuration peut prendre beaucoup de temps et peut se révéler relativement inutile en fin de partie. Pour cela différents points sont à prendre en compte lors du début de partie :

- Il est important de répartir suffisamment de pions sur chaque liserés afin d'éviter de se retrouver dans l'impossibilité de jouer
- Pouvoir protéger la licorne
- Pouvoir rester offensif sur la licorne adverse

A l'aide de ces critères, il nous ai donc apparu que les configurations suivantes étaient les plus optimisées pour répondre à ces objectifs :



Afin de pouvoir réaliser un choix sur le plateau, notre IA à besoin de prendre différents critères servant à évaluer le meilleur coup selon elle. Pour ce faire l'IA évalue notamment :

- La liberté de mouvement que notre coup nous offre par rapport à la liberté qu'il apporte à l'adversaire
- Si notre coup nous permet directement de gagner ou de laisser gagner l'adversaire
- Si ce coup nous permet de bloquer l'adversaire
- Si ce coup amène à une victoire future ou une défaite possible

Afin de réaliser cette recherche de meilleur coup, on se base donc sur l'algorithme MinMax dans l'objectif de maximiser nos résultats tout en réduisant ceux de l'adversaire.

## Implémentation des joueurs

Afin de mettre en place des joueurs qui puissent interagir avec le serveur de jeu via le client, nous nous sommes donc basés sur l'interface de joueur tout en y ajoutant un plateau d'escape afin de suivre localement le jeu. Trois types de joueurs ont été créés :

- **JoueurAleatoire** : il prends un mouvement possible parmis tous les mouvements plausible et le joue
- **JoueurHumain** : permets de laisser jouer un humain en lui laissant entrer directement les coups qu'il souhaite effectuer
- **JoueurIntelligent** : Il se base sur nos heuristiques afin de jouer

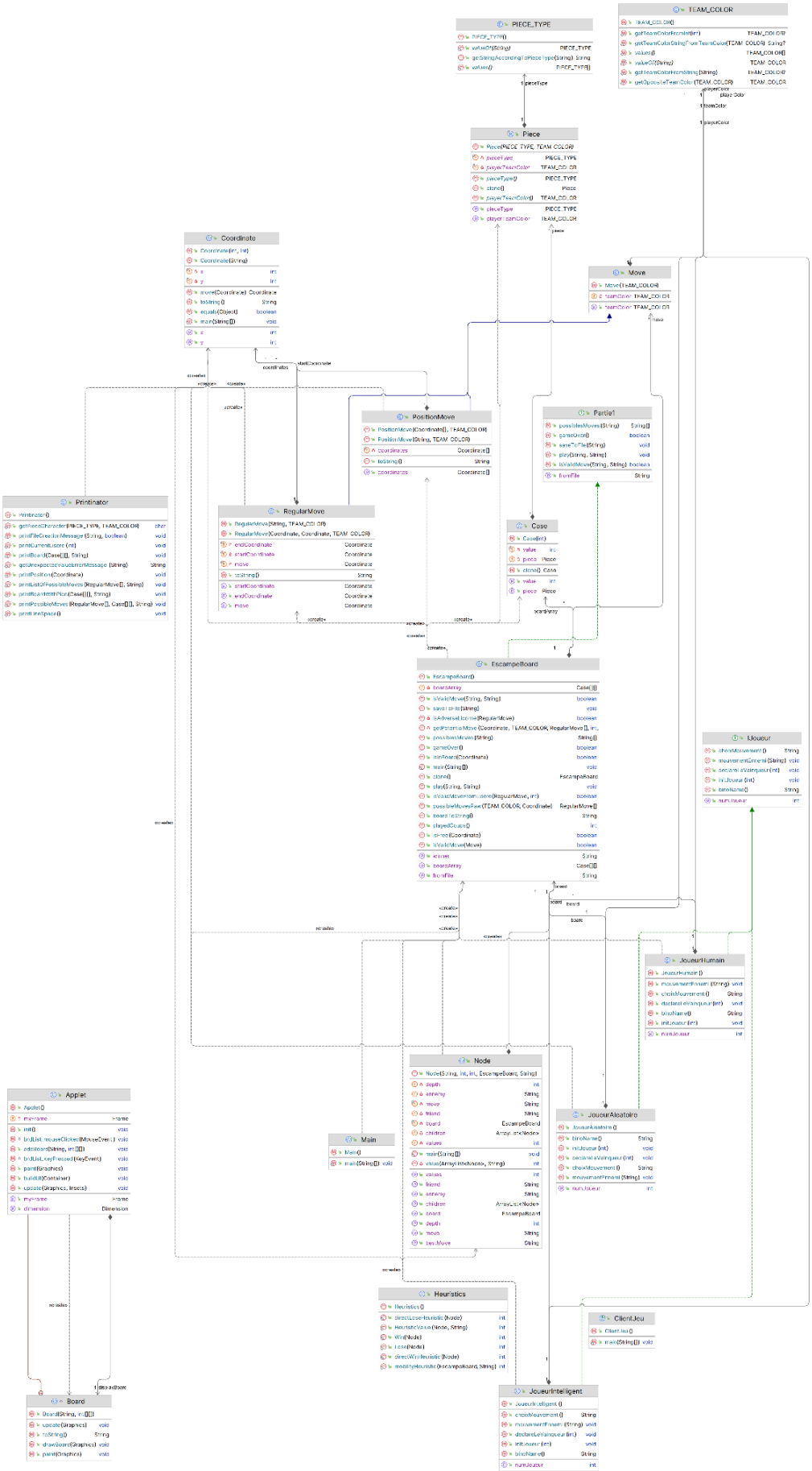
## Analyse des performances

Afin de confirmer les résultats positifs du joueur utilisant notre heuristique, nous avons effectué plusieurs tests chronométrés. Ces tests se sont déroulés de la manière suivante, nous faisons jouer le joueur intelligent, une fois en couleur noir, puis une fois en couleur blanche, contre le joueur aléatoire. Vous trouverez les résultats ci-dessous.

*Performances face à JoueurAléatoire*

| | Jouant les noirs | Jouant les blancs |
|---|---|---|
| Victoires | 1/5 | 1/5 |

On remarque donc ici que l'IA ne fait actuellement pas mieux que l'aléatoire. Cependant nous observons qu'elle se montre menaçante, étant régulièrement à 1 mouvement de conclure la partie mais ne l'effectue pas. Cela vient d'un problème que nous avons identifié : actuellement les déplacements possibles sont vers une case qui ne dispose d'aucun pions sur celle-ci. Or la licorne est un pion. Il ne reste donc plus qu'à rajouter les cases disposant de la licorne comme étant un déplacement possible et cela devrait régler le tout. cette correction devrait être effectuée dans les prochains jours


PIECE_TYPE
TEAM_COLOR
Piece
Coordinate
Move
PositionMove
Partie1
Printinator
RegularMove
Case
EscampeBoard
Joueur
JoueurHumain
Node
Applet
Main
JoueurAleatoire
Heuristics
ClientJeu
JoueurIntelligent
Board