Rapport de Projet

Étude des performances des algorithmes Max^n et paranoid pour la prise de décision dans le jeu de Tron Multijoueur

Equipe de développement :

A. Catherine ATTY

Sékou DOUMBOUYA

Manne Emile KITSOUKOU

Amirath Fara OROU-GUIDOU

Parcours : Licence 3 Informatique

Groupe : 2B

Sous la supervison :

LEHEMBRE Etienne

7 avril 2023

Table des matières

Introduction

La **théorie des jeux** et l'**intelligence artificielle** sont deux(2) domaines interconnectés qui ont connu une évolution fulgurante ces dernières années. La théorie des jeux est une branche des mathématiques qui se concentre sur l'analyse de la prise de décision stratégique dans des situations où de multiples acteurs interagissent et ont des intérêts contradictoires. Elle fournit un cadre pour l'analyse des choix des agents rationnels [1] et la prédiction de leurs résultats dans différents scénarios. En revanche, l'intelligence artificielle vise à doter les machines de la capacité d'imiter le comportement humain. Elle repose sur diverses techniques, telles que les algorithmes de recherche, l'apprentissage automatique et le traitement du langage naturel, pour simuler un comportement intelligent et prendre des décisions sur la base de données et de règles.

Le mariage entre la théorie des jeux et l'intelligence artificielle a conduit à de nombreuses applications, telles que les moteurs d'échecs, les robots de poker et les systèmes de recommandation. De plus, cette alliance a permis l'émergence de nouvelles voies de recherche, notamment les systèmes multi-agents, l'apprentissage par renforcement et la théorie du choix social.

Dans ce projet, nous avons l'ambition d'utiliser la théorie des jeux et l'intelligence artificielle pour mettre en œuvre et étudier le **jeu de Tron** avec plusieurs joueurs et différentes stratégies. Le jeu de Tron est un jeu multijoueur captivant inspiré du célèbre jeu **Snake** [2]. Dans ce dernier, les joueurs contrôlent un point qui se déplace sur une grille de taille fixe et laisse derrière lui un mur infranchissable à chaque déplacement. Le but est d'être le dernier joueur debout en évitant les collisions avec les murs, les bords et les trajectoires des adversaires. Ce jeu présente plusieurs défis pour l'analyse, tels que l'observabilité partielle, les mouvements simultanés et l'incertitude quant aux intentions des adversaires. Pour relever ces défis, les chercheurs ont proposé différents algorithmes, tels que $negamax$ avec élagage $alpha - beta$, recherche arborescente de $Monte\ Carlo$ et Max^n. Toutefois, l'ajout de plusieurs joueurs et équipes crée de nouvelles dimensions de complexité et de stratégie.

L'objectif de ce projet est d'étudier l'efficacité de différents algorithmes de jeu en évaluant les performances d'abord d'un agent solitaire, puis d'un agent multi-joueur. Pour ce faire, une analyse statistique sera effectuée dans diverses configurations de jeu. Ainsi, nous pourrons déterminer les algorithmes les plus efficaces et les plus performants.

Dans ce document, nous introduirons d'abord le jeu de Tron. Puis, nous exposerons la structure du projet et les phases successives de son élaboration. Ultérieurement, nous approfondirons les algorithmes de jeu employés et l'architecture globale du projet. Finalement, nous présenterons les constatations dérivées de notre évaluation statistique et tirerons des conclusions sur la performance des algorithmes.

1. Un agent rationnel est une entité qui vise toujours à réaliser des actions optimales sur la base de prémisses et d'informations données.

2. Le snake, de l'anglais signifiant « serpent », est un genre de jeu vidéo où le joueur contrôle un serpent qui s'agrandit au fur et à mesure de la progression du jeu, créant ainsi des obstacles pour le joueur

1 Jeu de Tron : Un jeu multijoueur captivant

1.1 Desciption du jeu

Film iconique des années 1980, **Tron** est une science-fiction américain réalisé par Steven Lisberger au sein de l'entreprise Walt Disney Pictures. Ce dernier met en scène un jeu dans un monde virtuel où des motos futuristes se déplacent à une vitesse constante, n'effectuant que des virages à angle droit [3] et laissant derrière elles des trainées solides.

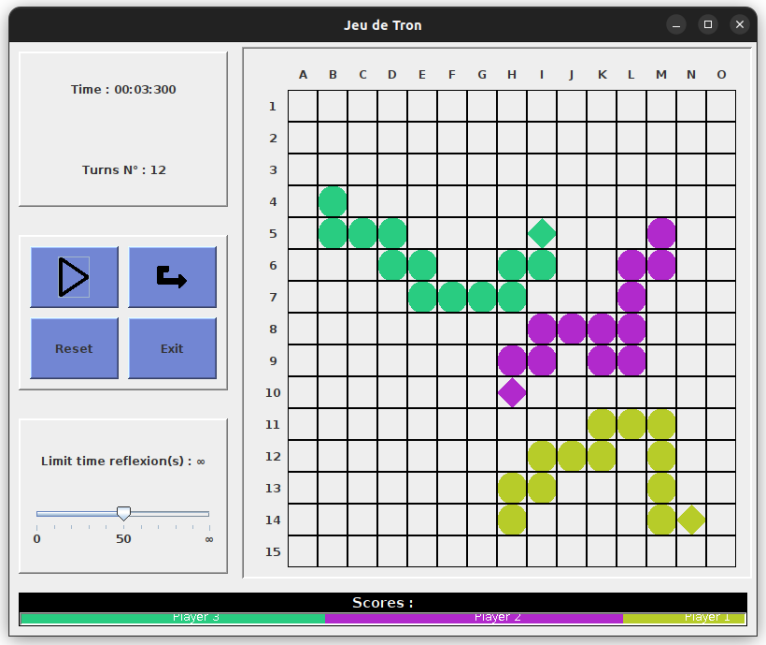

FIGURE 1 – Jeu de Tron

Ainsi, au fur et à mesure que le jeu avance, le plateau se remplit de murs et finalement un ou plusieurs joueurs se retrouvent coincés dans un cul-de-sac. Le but du jeu est de survivre le plus longtemps et d'être le dernier survivant. Ce jeu est devenu très populaire et a ensuite été implémenté sur diverses plateformes et diverses versions.

1.2 Analyse mathématiques du jeu

Tron est un jeu se jouant sur une grille de taille $N \times M$ avec N et M entiers positifs. Chaque cellule de la grille peut être `vide` ou `occupée` par un joueur. Généralement, la grille considérée est carrée, c'est à dire $N = M$. C'est un jeu multijoueur dans lequel les K joueurs, à chaque tour, font un choix :

- `continuer` dans la même direction
- `tourner` à 90° à gauche ou à droite

Un joueur ne peut pas s'arreter et doit toujours se déplacer. C'est un jeu assez rapide, car les joueurs se déplacent à une vitesse constante($\approx 100ms$ par tour). Tous les choix sont fait de manière simultanée ce qui rend le jeu très dynamique et complexifie l'anticipation des mouvements des adversaires. Le jeu se finit avec 2 états possibles :

- `victoire` d'un joueur, si il est le dernier survivant

3. Les virages à angle droit sont des virages où la nouvelle direction est perpendiculaire à la direction précédente.

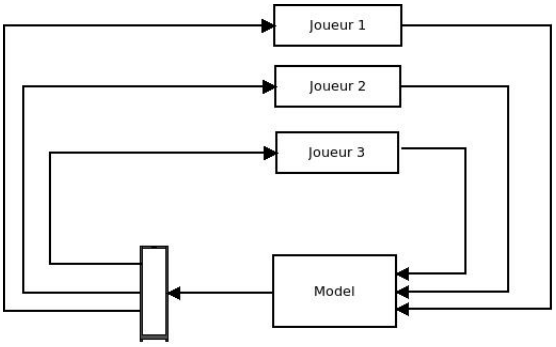

FIGURE 2 – Modèle d'interraction entre 3 joueurs

- `nulle` si tous les joueurs sont éliminés

Plus en détail, à chaque sequence discrète de temps $t = 0, 1, 2, ...$, les joueurs $i = 1, 2, ..., K$ reçoivent une même représentaion $s_t \in S$ de l'état du jeu, où S est l'espace des états possibles. En fonction de l'état s_t, chaque joueur i choisit une action $a_{i,t} \in A_i(s_t)$, où $A_i(s_t)$ est l'espace des actions possibles pour le joueur i dans l'état s_t. Ainsi, chaque joueur i. Comme conséquence, chaque joueur i reçoit une nouvelle représentation $s_{t+1} \in S$ de l'état courant du jeu.2

Ainsi le jeu de Tron est un jeu ayant une longueur fini. En effet, le nombre d'étape nécessaire pour terminer le jeu est majoré par :

$$\frac{N \times M}{K} \tag{1}$$

où N(resp. M) est la taille de la grille en largeur (resp. en hauteur) et K est le nombre de joueurs.
Toutefois en pratique, le jeu peut se terminer plus ou moins rapidement en fonction de la stratégie des joueurs.

2 Problématique et hypothèses de recherche

2.1 Problemématique de recherche

Le jeu de Tron est un jeu multijoueur complexes qui implique la prise de décision stratégique, la coordination et la compétition. L'un des principaux défis de ce jeu est d'anticiper les mouvements des adversaires et de concevoir des stratégies efficaces qui peuvent maximiser les chances de survie et de victoire d'un joueur. Cependant la stratégie optimale n'est pas toujours évidente, car elle dépend de divers facteurs. En effet, la taille de la grille, le nombre d'adversaires, la profondeur de recherche, la stratégie des adversaires et les ressources de calcul sont autant de facteurs qui peuvent influencer les performances d'un joueur. Ainsi, le projet vise à étudier l'efficacité de différents algorithmes de jeu à travers l'analyse de l'impact de ces divers facteurs sur les performances des joueurs.

2.2 Hypothèses de recherche

Pour répondre à cette problématique, diverses hypothèses de recherche ont été formulées :

- La performance des stratégies s'améliore avec la profondeur de recherche accordée à l'agent.

- L'efficacité des algorithmes se réduisent avec l'augmentation du nombre d'adversaires.
- Il existe une taille maximale de grille pour laquelle après dépassement, les performances des stratégies diminuent.

Ces hypothèses guideront les expérimentations, l'analyse des données et l'interpretation des résultats, et seront utilisées pour évaluer la pertinence des algorithmes de jeu.

3 Etat de l'art : théorie des jeux et Recherche adversarial

Le jeu Tron appartient à un genre prisé de jeux d'arcade basés sur la navigation d'un protagoniste à travers une grille labyrinthique. Depuis leur apparition, ces jeux ont évolué, intégrant désormais des fonctionnalités multijoueurs et des mécaniques de jeu plus élaborées. L'application de l'intelligence artificielle (IA) et de la théorie des jeux dans l'analyse de ces jeux a suscité un intérêt croissant, poussant les chercheurs à développer divers algorithmes et stratégies pour optimiser les performances des joueurs. Dans ce qui suit, nous présentons les algorithmes et stratégies prédominants dans le jeu de Tron, passons en revue la littérature existante concernant l'application de l'IA et de la théorie des jeux, mettons en évidence les avantages et les limites des différentes approches et identifiant les lacunes dans l'état actuel de la recherche.

3.1 Théorie des jeux

La théorie des jeux a été appliquée au jeu de Tron pour analyser la prise de décision stratégique de plusieurs joueurs et développer des stratégies de jeu optimales. Les recherches précédentes se sont concentrées sur l'utilisation de différents concepts de la théorie des jeux, tel que les équilibres de Nash et les stratégies mixtes, pour modéliser le comportement des joueurs et prédire leurs mouvements. Certaines études ont également exploré l'utilisation d'heuristique pour guider les joueurs vers des stratégies optimales dans divers scénarios.

En gandre partie, les études se sont concentrées sur l'examen de l'algorithme de recherche arborescente de *Monte Carlo* (MCTS). Des travaux tels que ceux de **Perick et al.** [5] ont comparé différentes méthodes de sélection pour optimiser les performances de l'algorithme. Il en découle que l'adoption d'une stratégie de sélection spécifique peut améliorer les résultats de l'algorithme. De plus, d'autres recherches, comme celle de **Den et al.** [3] ont exploré diverses heuristiques pour accroître l'efficacité de ces algorithmes. Ainsi, l'emploi d'une heuristique fondée sur l'estimation de l'espace libre constitue un bon indicateur de la qualité d'une action.

De plus, des travaux connexes ont été menées en utilisant des algorithmes de recherche adversarial(*minimax*, *alpha-beta*, *negamax*...). C'est le cas des travaux de **Boin**[2] dans lequel il présente divers versions de l'algorithme *minimax* et des heuristiques pour améliorer les performances de l'algorithme. Un autre article facinant est celui réalisé par le vainqueur de **Google IA Challenge** [6] qui a utilisé un algorithme de recherche adversarial pour sortir victorieux de ce Challenge. Il y montre et explique les différentes étapes de son travail, notamment celui sur l'implémentation des heuristiques qui apporte une amélioration significative des performances de l'algorithme.

3.2 Recherche adversarial multijoueur

La recherche adversarial est une technique fréquemment employée pour modéliser le comportement des participants dans les environnements multi-agents. Cela englobe l'analyse des algorithmes et des stratégies pour des scénarios impliquant plus de deux acteurs. Deux approches principales existent pour aborder ce défi, notamment l'application des algorithmes suivants :

- **Max**^{n}
- **Paranoid**

Ces algorithmes ont fait leur preuve dans le domaine des jeux multijoueurs et ont été largement utilisés pour analyser les stratégies de jeu optimales. En effet la recherche réalisé par [4] ont montré que l'algorithme paranoïaque est une option viable pour les jeux multijoueurs, puisqu'il surpasse systématiquement l'algorithme maxn au jeu de dames et obtient des résultats légèrement meilleurs au jeu de cœur. Cependant, l'algorithme maxn reste compétitif dans les jeux où il peut élaguer. Dans l'ensemble, nous concluons que l'algorithme paranoïaque est supérieur dans les jeux où l'algorithme maxn est contraint d'effectuer une recherche par force brute

4 Organisation et planification

Afin d'accomplir ce projet avec succès, nous avons opté pour une approche intégrant des **analyses théoriques**, la **mise en oeuvre d'algorithmes** et des **évaluations empiriques** [4]. Pour ce faire, nous devrions dans un premier temps fixer des objectifs clairs et définir les différentes étapes de notre travail. Ensuite, nous devrions nous pencher sur la relation de ces derniers en s'organisant et en planifiant notre travail.

4.1 Etapes du projet

Notre travail est divisé en trois grandes étapes principales :

- **Planification** : Définition des questions de recherche, formulation des hypothèses, Recherche bibliographique, choix de l'approche méthodologique et modèlisation du jeu.
- **Implémentation et Développement** : Développement du jeu, implémentation des algorithmes de recherche adversarial et des heuristiques, préparation des données et des expériences.
- **Evaluation et Analyse** : Analyse des résultats, interprétation des résultats et discussion des résultats.

Chacune des étapes comporte des objectifs et tâches spécifiques dont leur accomplissement est nécessaire pour le succès du projet.

4.2 Gestion du projet

Pour le bon déroulement du projet, nous avons décidé d'organiser globalement divers parties implémenter par chaque membre du groupe. Ainsi, nous avons décidé de nous répartir les tâches suivantes :

4. L'évaluation empirique désigne le processus consistant à utiliser des expériences, des mesures et des observations pour évaluer la performance ou l'efficacité d'un système ou d'une approche particulière

- **Manne** : Implémentation du modèle de jeu et des heuristiques
- **Amirath** : Implémentation des algorithmes de recherche adversarial (Max^n et $Paranoid$)
- **Catherine** : Implémentation des algorithmes de recherche adversarial et mise en place de la plateforme de test
- **Sekou** : Réalisation de l'interface graphique

En outre, nous avons décidé que chacun des membres du groupe devrait participer à l'analyse des résultats.

4.3 Planification du travail

Le diagramme de Gantt [5] ci-dessous représente la planification du travail. Nous avons

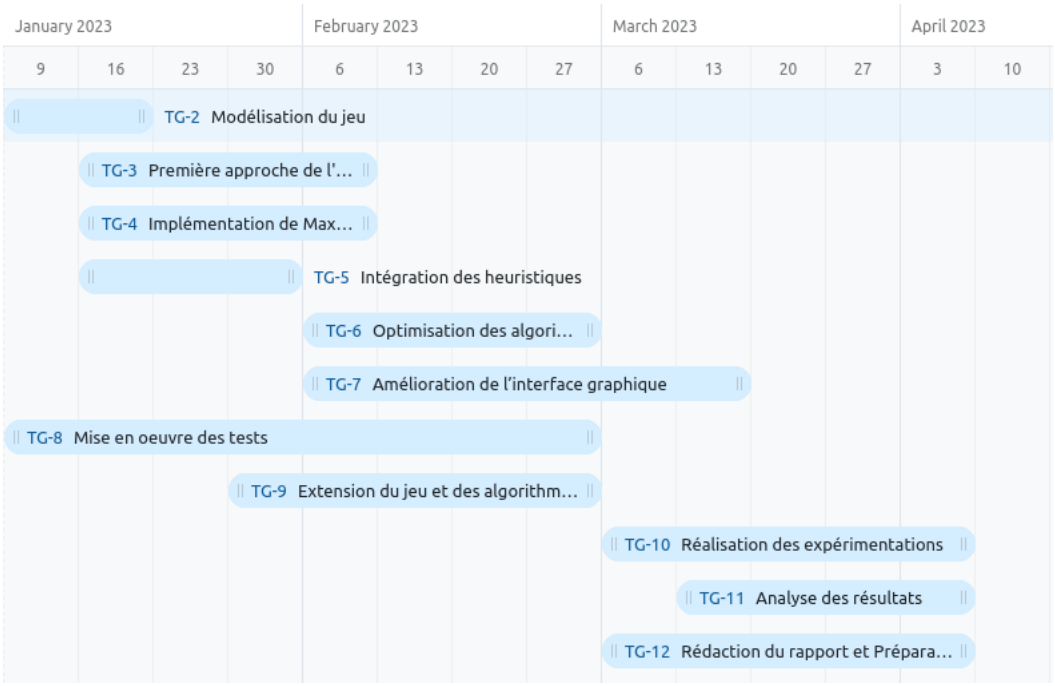

FIGURE 3 – Diagramme de Gantt

décidé de travailler sur le projet en trois phases, à savoir :

- **Phase 1** : Développement de l'environnement de jeu, implémentation des heuristiques, des algorithmes de recherche adversarial et mise en place de la plateforme de test
- **Phase 2** : Amélioration de l'environnement de jeu, algorithmes et extensions des algorithmes à la version multi-joueurs
- **Phase 3** : Réalisation des expériences et analyse des résultats

5. Diagramme de Gantt est un diagramme de type barre qui permet de représenter graphiquement l'avancement d'un projet.

5 Analyse et Démonstration des algorithmes : Les heuristiques

Dans le cadre de ce projet, différentes techniques heuristiques ont été utilisées pour améliorer l'algorithme de prise de décision. Nous avons pris la décision de nous concentrer exclusivement sur les heuristiques basées sur l'occupation de l'espace. Ces heuristiques sont particulièrement adaptées pour faciliter l'analyse des positions de jeu et l'évaluation de leur qualité.

5.1 OpenSpace

L'heuristique **OpenSpace** est une technique simple basée sur la comptabilisation du nombre de cases vides entourant un joueur donné. Elle se fonde uniquement sur la position de ce joueur et ne prend pas en considération la position des autres joueurs.

5.1.1 Implémentation

Son implémentation consiste à calculer le nombre de cases vides situées autour d'un joueur donné, à partir d'un état de jeu spécifique.

Algorithme 1 : OpenSpace

Entrées : *state* : état du jeu
Sortie : *scores* : score de l'état

```
1  scores ← {}
2  pour player ∈ state.players faire
3  |   score ← 0
4  |   pour neighbor ∈ state.neighbors(player) faire
5  |   |   si neighbor ∈ state.empty alors
6  |   |   |   score ← score + 1
7  |   |   fin
8  |   fin
9  |   scores[player] ← score
10 fin
11 retourner scores
```

5.1.2 Complexité

L'algorithme **OpenSpace** 1 a une complexité temporelle qui dépend uniquement du nombre de joueurs dans le jeu.

En effet la boucle extérieure de l'algorithme parcourt tous les joueurs présents dans l'état de jeu, ce qui prend un temps proportionnel à N, où N est le nombre de joueurs. De plus la boucle intérieure itère sur les quatres(4) voisins de chaque joueur, et la vérification de la liberté de chaque voisin(voir la ligne 5 1) se fait en temps constant($\mathcal{O}(1)$).

Par conséquent, la complexité de l'algorithme **Openspace** est de l'ordre de $\mathcal{O}(N)$, ce qui peut être considéré comme linéaire en fonction du nombre de joueurs.

5.2 GASLAP

GASLAP abréviation de "**G**o **AS** **L**ong **A**s **P**ossible" est une approche qui consiste à calculer la distance maximale qu'un joueur peut parcourir à partir de sa position de

départ dans une direction donnée, sans rencontrer d'obstacles. Cette méthode permet de prioriser les mouvements qui permettent au joueur de se déplacer le plus loin possible dans la direction choisie, tout en évitant les déplacements inutiles et en maximisant l'efficacité des mouvements.

5.2.1 Implémentation

Cette approche s'appuie sur la détermination du nombre de cases vides situées dans la direction choisie par le joueur.

Algorithme 2 : GASLAP

Entrées : *state* : état du jeu
Sortie : *scores* : score de l'état

```
scores ← {}
pour player ∈ state.players faire
    score ← 0
    direction ← state.direction(player)
    position ← player.position
    tant que position ∈ state.empty faire
        position ← position + direction
        score ← score + 1
    fin
    scores[player] ← score
fin
retourner scores
```

5.2.2 Complexité

Étant donné que le jeu se déroule sur une grille carrée de taille fixe $n \times n$,un joueur ne peut se déplacer dans une même direction que $n-1$ fois. Ainsi, la complexité de la boucle intérieure(boucle **While**) est de l'ordre de $\mathcal{O}(n)$. De plus la boucle extérieure (boucle **For**) parcourt tous les joueurs présents dans l'état de jeu, ce qui prend un temps proportionnel à J, où J est le nombre de joueurs.

Par conséquent, la complexité de l'algorithme **GASLAP** est de l'ordre de $\mathcal{O}(J \times n)$, ce qui peut être considéré comme linéaire en fonction du nombre de joueurs car n est une constante.

5.3 Voronoï

Le diagramme de **Voronoï** est une technique mathématique qui permet de diviser l'espace en régions distinctes en fonction de la proximité de points spécifiques. Dans le cadre de ce projet, nous avons utilisé cette technique pour déterminer les zones d'influence [6] des joueurs.

5.3.1 Implémentation

Pour mettre en œuvre cette technique, nous commençons par calculer la distance entre le joueur et toutes les autres cases vides [7] de la grille. Pour ce faire, nous adaptons les

6. Zone d'influence est la zone pour laquelle un joueur peut atteindre en premier à partir de sa position de départ.
7. Case vide est une case qui n'est pas occupée par un joueur.

principes de l'algorithme de **Dijkstra** à notre cas spécifique(voir algorithme 3).

Algorithme 3 : Détermination de la distance entre le joueur et les cases vides

Entrées : *initial_position* : Position initial du joueur, *positions_vides* : Ensemble des positions des cases vides

Sortie : *distances* : Dictionnaire contenant les distances entre le joueur et les cases vides

```
1  distances ← {}
2  queue ← {}
3  pour chaque position_vide ∈ positions_vides faire
4  |  distances[position_vide] ← ∞
5  fin
6  distances[initial_position] ← 0
7  queue ← queue ∪ {initial_position}
8  tant que queue ≠ {} faire
9  |  position ← queue[0]
10 |  queue ← queue \ {position}
11 |  pour chaque adjacent ∈ get_adjacents(position) faire
12 |  |  si distances[adjacent] > distances[position] + 1 alors
13 |  |  |  distances[adjacent] ← distances[position] + 1
14 |  |  |  queue ← queue ∪ {adjacent}
15 |  |  fin
16 |  fin
17 fin
18 retourner distances
```

Après avoir calculé les distances entre chaque joueur et les cases vides à l'aide de l'algorithm 3, nous utilisons ces informations pour déterminer les zones d'influence de chaque joueur en ajoutant les cases vides les plus proches de lui, et nous obtenons la taille de ces zones d'influence en utilisant la méthode décrite dans l'algorithme 4.

Algorithme 4 : Détermination de la zone d'influence d'un joueur

Entrées : *state* : Etat courant du jeu
Sortie : *zones_influence* : Dictionnaire contenant la taille de la zone d'influence de chaque joueur

```
1  zones_influence ← {}
2  distances ← {}
3  pour chaque player ∈ state.players faire
4  |  zones_influence[player] ← 0
5  |  distances[player] ← distance(player.position, state.vide_positions)
6  fin
7  pour chaque position_vide ∈ state.vide_positions faire
8  |  min_distance ← ∞
9  |  min_player ← None
10 |  pour chaque player ∈ state.players faire
11 |  |  si distances[player][position_vide] < min_distance alors
12 |  |  |  min_distance ← distances[player][position_vide]
13 |  |  |  min_player ← player
14 |  |  fin
15 |  |  sinon si distances[player][position_vide] == min_distance alors
16 |  |  |  min_player ← None
17 |  |  fin
18 |  fin
19 |  si min_player ≠ None alors
20 |  |  zones_influence[min_player] ← zones_influence[min_player] + 1
21 |  fin
22 fin
23 retourner zones_influence
```

5.3.2 Complexité

Étant donné que l'algorithm **Voronoï** 4 est basé sur l'utilisation de l'algorithme de **distance** 3, il existe un lien direct entre la complexité de ces deux algorithmes.

En effet, l'algorithme de **distance** 3 est basé sur l'algorithme de **Dijkstra** qui est un algorithme ayant une complexité de l'ordre de $\mathcal{O}(|A| + |S|log|S|)$ où A est l'ensemble des arêtes [8] de la grille et S est l'ensemble des cases vide de la grille. Vu que l'on itère J fois sur l'algorithme de **distance** 3 où J est le nombre de joueurs, la complexité de la première partie de l'algorithme **Voronoï** 4 (ligne 3-6) est de l'ordre de $\mathcal{O}(J(|A| + |S|log|S|))$. La deuxième partie de l'algorithme **Voronoï** 4 (ligne 7-15) est basée sur une double itération entre les cases vides et les joueurs, la complexité de cette partie est donc de l'ordre de $\mathcal{O}(|S|J)$.

La complexité totale de l'algorithme **Voronoï** 4 est donc de l'ordre de $\mathcal{O}(J(|A| + |S|log|S|) + |S|J)$.

Toutefois, en fonction de l'état de la grille, on distingue deux cas où l'on peut majoré beaucoup plus efficacement la complexité de l'algorithme **Voronoï** 4 :

- $|S| \gg |A|$: Dans ce cas, on peut majorer la complexité de l'algorithme **Voronoï** 4 en $\mathcal{O}(J|S|log|S|)$.
- $|S| \ll |A|$: Dans ce cas, on peut majorer la complexité de l'algorithme **Voronoï** 4 en $\mathcal{O}(J|A|)$.

8. Arête est une liaison entre deux cases.

C'est deux(2) cas sont ceux qui sont le plus souvent rencontrés dans le jeu de **Tron**.

5.4 Checker

Compte tenu de la structure du jeu qui se joue sur une grille, nous pouvons considérer cette grille comme un échiquier, où les cases sont alternativement blanches et noires. En conséquence, un joueur ne peut se déplacer que d'une case blanche à une case noire ou vice versa. Ainsi, en utilisant l'algorithme **Voronoï** 4, nous pouvons éliminer les cases blanches et noires excédentaires qui se trouvent dans la zone d'influence d'un joueur. Cela nous permet d'obtenir une limite supérieure plus précise de la zone d'influence d'un joueur.

5.4.1 Implémentation

L'algorithme **Checker** 5 est découle du développement réalisé sur celui de voronoï, une différence notable est que dans le case du **Checker**, nous comptons le nombre de cases blanches et noires dans la zone d'influence d'un joueur, ce qui permet d'en réduire le surplus.

Algorithme 5 : Détermination de la zone d'influence d'un joueur

Entrées : *state* : Etat courant du jeu
Sortie : *zones_influence* : Dictionnaire contenant la taille de la zone d'influence de chaque joueur

```
1  zones_influence ← {}
2  distances ← {}
3  count_white ← 0
4  count_black ← 0
5  pour player ∈ state.players faire
6  |  distances[player] ← distance(state, player)
7  |  zones_influence[player] ← 0
8  |  count_white[player] ← 0
9  |  count_black[player] ← 0
10 fin
11 pour chaque position_vide ∈ state.vide_positions faire
12 |  min_player ← None
13 |  min_distance ← ∞
14 |  pour player ∈ state.players faire
15 |  |  si distances[player][position_vide] < min_distance alors
16 |  |  |  min_player ← player
17 |  |  |  min_distance ← distances[player][position_vide] /* On récupère le
   |  |  |     joueur le plus proche de la case vide                    */
18 |  |  fin
19 |  |  sinon si distances[player][position_vide] == min_distance alors
20 |  |  |  min_player ← None
21 |  |  |  min_distance ← ∞
22 |  |  fin
23 |  fin
24 |  si min_player! = None alors
25 |  |  zones_influence[min_player] ← zones_influence[min_player] + 1
26 |  |  si position_vide.color ==′ white′ alors
27 |  |  |  count_white[min_player] ← count_white[min_player] + 1
28 |  |  fin
29 |  |  sinon
30 |  |  |  count_black[min_player] ← count_black[min_player] + 1
31 |  |  fin
32 |  |  zones_influence[min_player] ← zones_influence[min_player] + 1
33 |  fin
34 fin
35 pour player ∈ state.players faire
36 |  surplus ← |count_white[player] − count_black[player]|
37 |  zones_influence[player] ← zones_influence[player] − surplus
38 fin
39 retourner zones_influence
```

5.4.2 Complexité

La complexité dans le pire des cas de l'algorithme **Checker** 5 est du même ordre que celle de l'algorithme **Voronoï** 4, c'est à dire $\mathcal{O}(J(|A| + |S|log|S|))$. En effet, l'algorithme **Checker** 5 est une extension de l'algorithme **Voronoï** 4. Cette extension permet de trouver une limite supérieure plus précise de la zone d'influence d'un joueur.

6 Architecture du projet

Notre projet est basé sur l'architecture logicielle MVC (Modèle-Vue-Contrôleur) qui permet de séparer la logique métier de la présentation. Nous avons ainsi identifié 4 modules principaux(voir figure 4) :

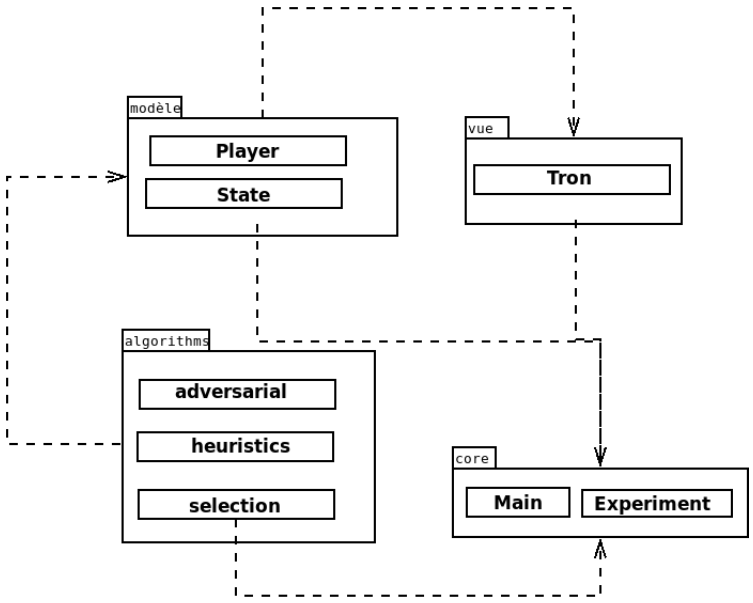

FIGURE 4 – Architecture du projet

- **Modèle** : Il contient les classes qui représentent les différents éléments du jeu comme les joueurs, la grille de jeu, etc. Il gère également les règles du jeu et les interactions entre les différents éléments.
- **Vue** : Il est responsable de l'affichage graphique du jeu et des interactions utilisateur. Il est constitué d'une interface utilisateur (UI) qui permet à l'utilisateur de jouer et de visualiser les différentes informations du jeu.
- **Core** : Contient l'ensemble des classes executables qui permettent de lancer le jeu.
- **algorithms** : Il contient l'ensemble des algorithmes de recherche adversarial, des techniques de selection et des heuristiques.

En somme, l'architecture MVC nous permet de séparer les différentes responsabilités de notre application, ce qui facilite la maintenance et l'évolution de notre code.

6.1 Modules et Classes

6.1.1 Modèle

Le modèle de notre application repose sur plusieurs classes interdépendantes(voir figure 5) pour gérer l'état du jeu, le plateau de jeu, les joueurs, leur position et leurs mouvements.

6.1.1.1 Move

La classe Move est une classe importante de notre modèle qui représente le mouvement effectué par un joueur. Elle contient les informations nécessaires pour décrire un mouvement, à savoir la position final du joueur(dans notre cas la tête du joueur), la direction du

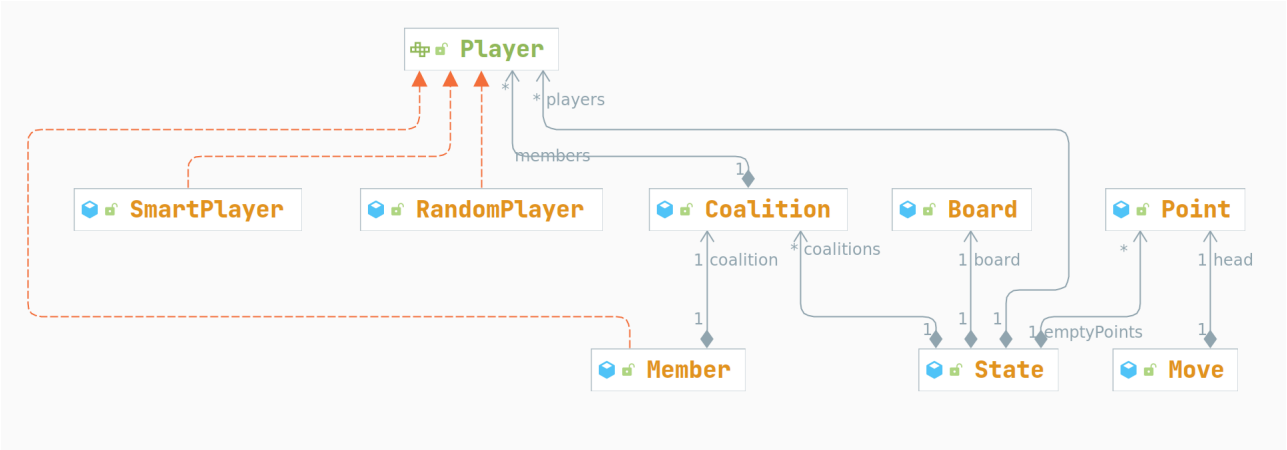

FIGURE 5 – Diagramme de classe du modèle

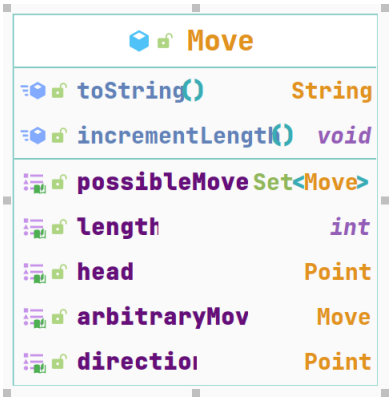

FIGURE 6 – Diagramme de classe de la classe Move

mouvement et la longueur du mouvement qui a été effectué dans la même direction. Grâce aux méthodes de la `Move`(voir figure 6), on peut effectuer la validation d'un mouvement, obtenir la position final d'un mouvement, obtenir la direction d'un mouvement, obtenir la longueur d'un mouvement, etc.

Le choix de stocker la longueur d'un mouvement dans la classe `Move` découle du besoin de stocker les mouvements des joueurs dans une liste. En effet en considérant les mouvements effectués par le joueur rouge dans la figure **??**, si on stockait chaque position à un temps donné, on aurait eu besoin de stocker 20 éléments pour reconstituer le chemin du joueur rouge. Cependant, en utilisant notre implémentation, nous n'avons besoin de stocker que 9 mouvements, ce qui permet d'optimiser l'utilisation de la mémoire de notre application.

6.1.1.2 State

La classe `State` représente l'état du jeu à un instant donné. Elle contient les informations nécessaires pour décrire l'état du jeu(voir figure 7).

À travers cette classe, nous fournissons un objet permettant non seulement de pouvoir l'utiliser dans les algorithmes de recherche adversarial, mais aussi de pouvoir l'utiliser dans l'interface graphique pour afficher l'état du jeu.

L'une des principales méthodes de cette classe est la méthode `initBeginPoints` qui permet d'initialiser les positions initiales des joueurs. Cette méthode est appelée dans le

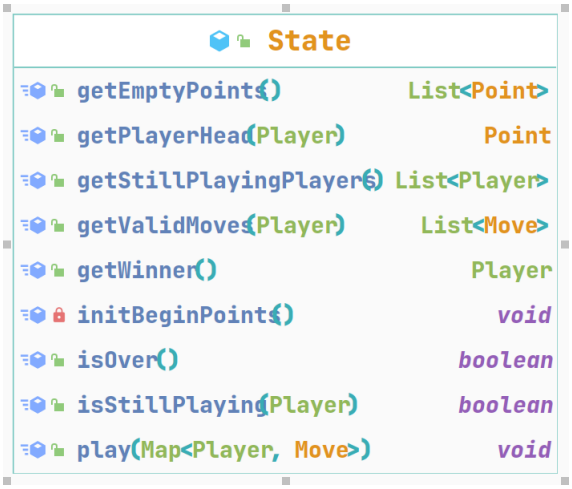

FIGURE 7 – Diagramme de classe de la classe State

constructeur de la classe `State` et permet de définir les positions initiales des joueurs. Pour un souci d'équitabilité, nous avons décidé de placer les joueurs de manière equidisante de centre du plateau de jeu. C'est à dire :

$\forall i \in$ [0, nbJoueurs[

$$pos_x(i) = \frac{width}{2} + \frac{width}{2} \times \frac{2\pi i}{nbJoueurs}$$
$$pos_y(i) = \frac{height}{2} + \frac{height}{2} \times \frac{2\pi i}{nbJoueurs} \quad (2)$$

où $width$ et $height$ sont les dimensions du plateau de jeu et $nbJoueurs$ est le nombre de joueurs.

6.1.2 Vue

vue de l'application comporte des classes spécialement conçues pour faciliter l'interaction avec l'interface graphique utilisateur et pour offrir une représentation visuelle détaillée du déroulement des algorithmes et des parties. Elle permet à l'utilisateur de suivre les différentes étapes de l'exécution des algorithmes, de visualiser les mouvements des joueurs et d'observer le plateau de jeu de manière plus détaillée. En conséquence, la vue améliore significativement la compréhension et l'expérience de l'utilisateur lors de l'utilisation de l'application.

6.1.2.1 Les popups

Les popups sont des éléments de l'interface graphique utilisateur qui sont utilisés pour fournir des informations supplémentaires à l'utilisateur ou pour les inviter à effectuer une action. Dans notre application, nous les avons utilisés pour afficher les informations relatives à la partie en cours tel que la fin de la partie, le gagnant, le nombre de coups joués(voir figure 8), etc.

FIGURE 8 – Popup de fin de partie

En outre, nous avons utilisé des fenêtres contextuelles pour permettre à l'utilisateur d'instancier les différentes IA qui s'affronteront sur le plateau et voir le récapitulatif global(voir figure 9).

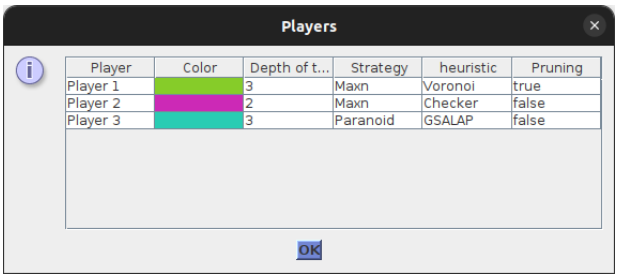

FIGURE 9 – Fenêtre contextuelle

Cette fonction est particulièrement utile car elle permet à l'utilisateur de sélectionner et de modifier les paramètres des différentes IA, tels que leur profondeur ou leur fonction heuristique. En fournissant ces fenêtres contextuelles, nous sommes en mesure d'améliorer l'expérience de l'utilisateur et de rendre l'application plus conviviale et intuitive.

6.1.2.2 La fenêtre de jeu

La fenêtre principale du jeu sert d'interface principale pour afficher l'état actuel du jeu. Elle se compose de trois parties principales(voir figure 10)

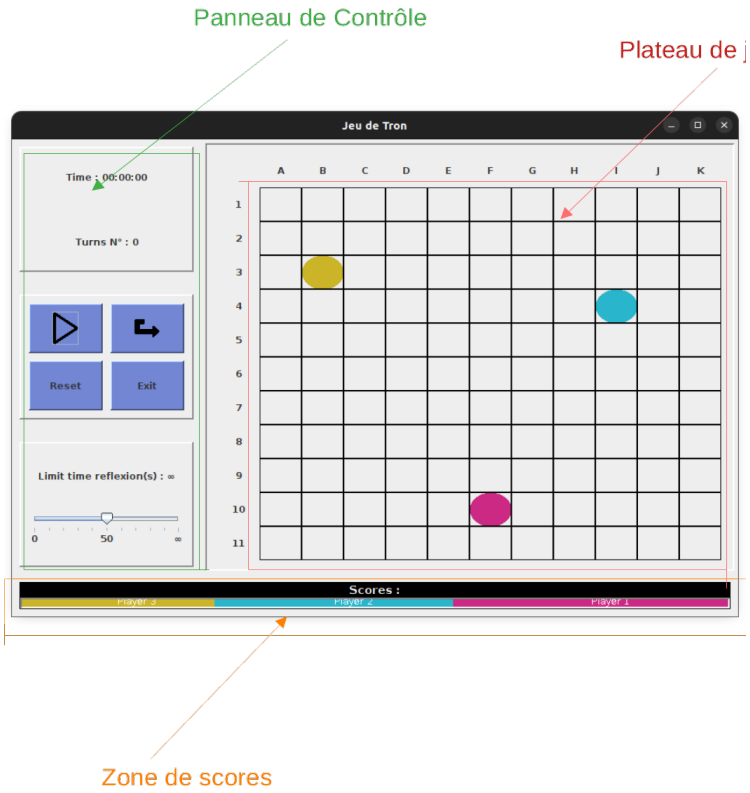

FIGURE 10 – Fenêtre principale du jeu

Le plateau de jeu est une grille qui représente le terrain de jeu et affiche la position actuelle des joueurs. La zone de score fournit une représentation graphique de la zone occupée par chaque joueur sous la forme d'un graphique à barres. Le panneau de contrôle contient divers boutons et commandes, notamment les boutons de démarrage, de réinitialisation et d'étape suivante, ainsi que des informations sur le nombre de tours, la durée

du jeu et un curseur permettant d'ajuster le temps de réflexion d'un algorithme.

6.1.3 Algorithmes

Le package `algorithms` contient tous les algorithmes spécifiques que nous avons mis en œuvre pour le jeu(voir figure 11).

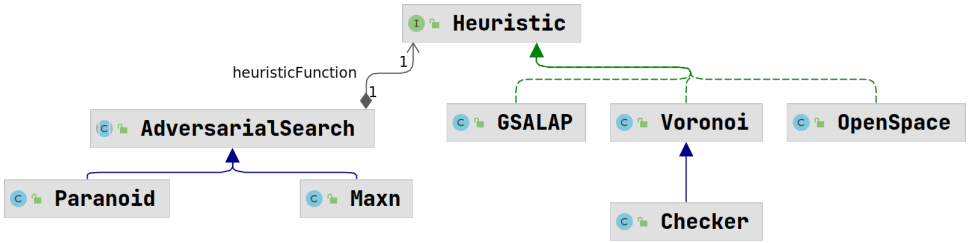

FIGURE 11 – Diagramme de classe du package algorithms

Ces algorithmes sont classés en deux catégories :

- les algorithmes de recherche adversarial
- les heuristiques(voir section 5) [9]

6.2 Les fonctionnalités

L'application offre diverses fonctionnalités qui permettent à l'utilisateur d'interagir avec le jeu, les algorithmes et l'interface. Les principales fonctionnalités sont les suivantes :

- **Configuration de la partie** : : l'utilisateur peut configurer le jeu en définissant la taille du plateau, le nombre de joueurs et les algorithmes à utiliser.
- **Coloration d'un joueur** : Une couleur est attribuée à chaque joueur(couleur aléatoire variant d'une partie à l'autre) et est utilisée pour représenter le joueur sur le plateau de jeu.
- **Contrôle du jeu** : l'utilisateur peut démarrer, mettre en pause, reprendre et réinitialiser le jeu. L'utilisateur peut également avancer dans le jeu un coup à la fois.
- **Configuration des joueurs** : On peut configurer le temps de réflexion, la profondeur maximale de la recherche et les poids des heuristiques utilisées dans les algorithmes.
- **Sélection des algorithmes** : l'utilisateur peut choisir les algorithmes(recherche adversarial et heuristique) à utiliser pour chaque joueur.
- **Visualisation** : l'utilisateur peut visualiser les mouvements des joueurs et les étapes de l'exécution des algorithmes.
- **Statistiques du jeu** : l'application affiche diverses statistiques pendant et après le jeu, telles que le nombre de coups, la durée du jeu et les scores de chaque joueur.
- **Personnalisation de l'interface utilisateur** : l'utilisateur peut personnaliser l'interface utilisateur notamment en redimensionnant la fenêtre principale du jeu.

9. Les heuristiques sont utilisées pour évaluer les états du jeu et les comparer entre eux. Elles sont utilisées par les algorithmes de recherche adversarial pour déterminer le meilleur mouvement à effectuer.

7 Expérimentations et Évaluation

Dans l'optique d'évaluer la performance des algorithmes Max^n et *Paranoid*, divers expérimentations ont été menées. Notre objectif était de confirmer ou d'infirmer plusieurs hypothèses formulées au cours du processus de développement (voir section 2). Pour ce faire, nous avons utilisé différentes mesures d'évaluation telles que le nombre de nœuds étendus par les algorithmes au cours de leur recherche et la qualité des stratégies générées (victoire, défaite ou match nul). Nous avons également évalué la robustesse des algorithmes par rapport à différents facteurs tels que la taille du plateau, le nombre de joueurs et le niveau de difficulté des algorithmes adversaires.

7.1 Cas d'utilisation testés

Dans le cadre du projet, nous avons réalisé plusieurs cas d'utilisation afin d'évaluer les performances des algorithmes mis en œuvre. Nous avons donné la priorité à l'utilisation des versions élaguées des algorithmes en raison de leur complexité et du nombre élevé de nœuds pendant l'exécution. L'élagage nous permet de réduire le nombre de nœuds visités sans compromettre la qualité des stratégies calculées, ce qui permet de réduire le temps d'exécution des algorithmes. En outre, nous nous sommes concentrés sur un jeu à trois joueurs, qui est considéré comme plus complexe que la version à deux joueurs. Ce choix a été fait pour faciliter l'évaluation de la robustesse des algorithmes, car plus nous augmentons le nombre de joueurs, plus la complexité du jeu augmente et nécessitera une plus grande puissance de calcul. Ces décisions ont été prises en tenant compte des limitations matérielles, car notre projet nécessite des machines dotées de grandes capacités de calcul en raison de la taille de notre problème.

En résumé, étant donné un jeu à 3 joueurs et les 2 algorithmes disponibles, il y a un total de $2^3 = 8$ combinaisons d'algorithmes possibles. Cependant, si nous excluons les cas où les 3 joueurs se voient attribuer le même algorithme, car ces configurations ne sont pas informatives, nous nous retrouvons avec un total de 6 affectations d'algorithmes possibles(voir tableau 1).

Joueur 1	Joueur2	Joueur3
Max^n	Max^n	*Paranoid*
Max^n	*Paranoid*	*Paranoid*
Max^n	*Paranoid*	Max^n
Paranoid	Max^n	*Paranoid*
Paranoid	Max^n	Max^n
Paranoid	*Paranoid*	Max^n

TABLE 1 – Cas d'utilisation testés

7.2 Méthodologie

Pour garantir la fiabilité et la précision de nos résultats, nous avons suivi une méthodologie rigoureuse pour les expériences. Nous nous sommes concentrés sur la performance des algorithmes lorsqu'ils sont utilisés par le premier joueur. Pour chaque configuration d'algorithme, nous avons répété les expériences au moins 10 fois, sauf dans les cas où le temps de calcul était excessivement long. Cela nous a permis d'obtenir une valeur moyenne précise pour la performance de chaque configuration d'algorithme. Les résultats présentés dans nos expériences sont donc fonction de la moyenne des répétitions.

L'utilisation de cette méthodologie présente plusieurs avantages. Tout d'abord, elle permet de réduire l'effet des fluctuations aléatoires et des anomalies qui peuvent survenir lors des différentes exécutions de l'algorithme. En prenant une moyenne sur plusieurs exécutions, nous pouvons obtenir une estimation plus représentative de la performance réelle de l'algorithme. Deuxièmement, cela nous permet de tester la robustesse des algorithmes dans différentes conditions initiales et différents états de jeu.

Cependant, le fait de répéter les expériences plusieurs fois présente également certains inconvénients. L'un des principaux inconvénients est l'augmentation du temps de calcul et des ressources nécessaires pour réaliser les expériences. En fonction de la complexité des algorithmes et de la taille du problème, l'exécution de plusieurs répétitions peut nécessiter un temps et une puissance de calcul considérable. Un autre inconvénient potentiel est la possibilité d'introduire un biais dans les résultats, en particulier s'il y a des erreurs systématiques dans la configuration expérimentale ou le processus de collecte des données.

Dans l'ensemble, en répétant les expériences plusieurs fois et en prenant la moyenne des résultats, nous avons pu obtenir une estimation plus précise de la performance des algorithmes dans différentes conditions. Bien que cette approche ait ses avantages et ses inconvénients, il s'agit d'une méthodologie couramment utilisée dans l'évaluation des algorithmes et elle fournit un cadre utile pour tirer des conclusions significatives de nos expériences.

7.3 Résultats des expérimentations et analyse

La première courbe12 de notre analyse présente une tendance intéressante. Lorsque nous augmentons la profondeur de l'algorithme de recherche, nous observons une amélioration progressive des performances jusqu'à un certain point, après quoi les performances chutent brusquement. Ce type de courbe est souvent appelé "courbe en cloche" ou "courbe en U inversé".

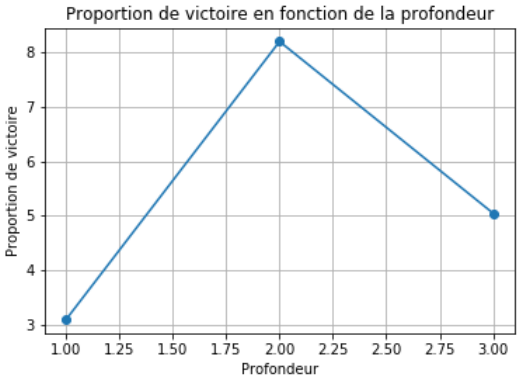

FIGURE 12 – Courbe de performance de l'algorithme Max^n en fonction de la profondeur

L'augmentation initiale des performances peut être attribuée au fait que l'algorithme est capable d'explorer davantage de mouvements et d'états de jeu possibles à mesure que la profondeur de recherche augmente. Il en résulte une évaluation plus précise des résultats potentiels de chaque mouvement, ce qui permet à l'algorithme de prendre de meilleures décisions. Toutefois, au-delà d'une certaine profondeur, l'augmentation de la complexité informatique de l'algorithme l'emporte sur les avantages de l'exploration d'espaces de recherche plus profonds. Cela peut conduire à un surajustement de l'algorithme à des états de jeu spécifiques, ce qui se traduit par une performance plus faible en moyenne.

Ce même effet se remarque également dans les expérimentations de Paranoid qui donne des victoire uniquement en profondeur 2

Il est important de noter que la profondeur optimale de l'algorithme de recherche peut varier en fonction du jeu spécifique auquel on joue, ainsi que des ressources matérielles et logicielles disponibles. Il est donc essentiel d'ajuster soigneusement le paramètre de profondeur de recherche afin d'obtenir les meilleures performances pour un jeu et un système donnés.

Dans l'ensemble, le phénomène de courbe en cloche observé dans nos expériences souligne l'importance d'équilibrer les ressources informatiques et la profondeur de recherche dans le développement d'algorithmes de jeu efficaces.

Il convient de rappeler que les autres expérimentations que nous avons menées, notamment sur l'influence du choix de la profondeur de recherche et du nombre de simulations, n'ont pas donné de résultats concluants. Les résultats ont été trop variables et ne permettaient pas de dégager de tendance claire.

Conclusion et perspectives

En conclusion, notre projet a exploré l'utilisation des algorithmes Maxn et paranoïaque dans le contexte d'un jeu multijoueur, et a montré que le choix de l'algorithme peut avoir un impact significatif sur la performance du joueur. Nos expériences ont démontré que l'algorithme paranoïaque est un concurrent de taille pour les jeux multijoueurs, surpassant l'algorithme Maxn dans certains cas.

Pour ce qui est de l'avenir, il existe plusieurs directions intéressantes dans lesquelles ce projet pourrait être étendu. L'une d'entre elles consisterait à étudier les performances des algorithmes Maxn et paranoïaque dans d'autres jeux multijoueurs que ceux que nous avons testés, tels que Tron. En particulier, le jeu Tron est un bon candidat pour une exploration plus approfondie car il présente un type de défi différent en ce qui concerne la stratégie et le mouvement, et nécessite donc des approches différentes de la conception et de l'optimisation des algorithmes.

Dans l'ensemble, notre projet a contribué à une meilleure compréhension des défis et des opportunités présentés par les jeux multijoueurs, et a démontré le potentiel des techniques algorithmiques avancées pour améliorer les performances des joueurs. Nous pensons que ce travail intéressera aussi bien les chercheurs que les développeurs de jeux, et nous sommes impatients de voir l'impact qu'il aura sur le domaine dans les années à venir.

Annexes

Table des figures

Liste des tableaux

Liste des Algorithmes

Bibliographie

Références

[1] Hans Leo Bodlaender, Antonius Jacobus Johannes Kloks, et al. Fast algorithms for the tron game on trees. 1990.

[2] Jean-Baptiste Boin. Cs221 project progress-light cycle racing in tron.

[3] Niek GP Den Teuling and Mark HM Winands. Monte-carlo tree adversarialsearch for the simultaneous move game tron. *Univ. Maastricht, Netherlands, Tech. Rep*, 2011.

[4] Sturtevant Nathan. A comparison of algorithms for multi-player games - university of alberta.

[5] Pierre Perick, David L St-Pierre, Francis Maes, and Damien Ernst. Comparison of different selection strategies in monte-carlo tree search for the game of tron. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 242–249. IEEE, 2012.

[6] Andy Sloane. Google ai challenge post-mortem, Mar 2010.