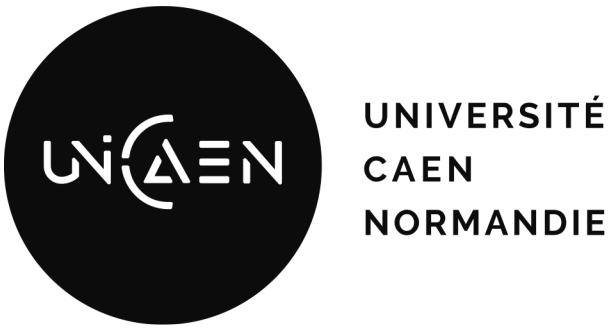
Rapport de projet - Conception logicielle 2
Jeu De Tron

Annou Rayane
Madjid Selmane
Mayas Ould Kaci
Ait Ali Belkacem Sonia

March 28, 2025



Contents

1 Introduction

1.1 Description générale du projet

Dans le cadre du projet de conception logicielle 2, nous avons opté pour le jeu de Tron, qui représente un terrain d'expérimentation idéal pour le développement et l'évaluation de stratégies d'intelligence artificielle (IA). Dans ce jeu, chaque joueur contrôle un point qui se déplace sur une grille, laissant derrière lui un mur infranchissable. L'objectif est d'être le dernier survivant, tout en évitant les collisions avec les murs, les adversaires et les bords du plateau.

Le jeu devient d'autant plus intéressant en mode multijoueur, car les défis stratégiques deviennent plus complexes. Cela nous amène à faire appel à des algorithmes comme MAXN , avant de les adapter à des équipes en nous inspirant de l'approche SOS (Socially Oriented Search).

1.2 Présentation du plan du rapport

Ce rapport commence par une présentation du jeu de Tron et de la problématique étudiée. Nous décrivons ensuite les structures de données, les algorithmes utilisés et les principales étapes de réalisation. Enfin, nous présentons la chaîne de traitement, les résultats obtenus et une analyse des performances des IA.

2 Objectifs du projet

L'objectif principal de ce projet est de concevoir une IA capable de simuler des parties de Tron multi-joueur afin d'étudier la performance des équipes dans différents scénarios. Pour cela, en plus d'implémenter une version classique du jeu, nous explorons l'impact de plusieurs facteurs, tels que la profondeur de raisonnement, la taille de la grille et les différences de taille entre les équipes. Ces expérimentations permettront d'évaluer comment les stratégies individuelles et collaboratives influencent les résultats et d'apporter une réflexion sur l'efficacité des algorithmes dans un environnement compétitif.

Dans ce cadre, nous avons implémenté les algorithmes, MAXN, MinMax et SOS, chacun apportant une approche différente à la prise de décision et à la gestion des interactions entre joueurs.

2.1 Problématique

La profondeur de recherche dans un algorithme de décision influence directement la qualité des choix effectués par une intelligence artificielle. Dans le cadre du jeu de Tron multi-joueur avec coalitions, il est essentiel d'évaluer comment cette profondeur impacte les performances des équipes en fonction de la taille du plateau et de la composition des joueurs. Une recherche plus profonde permet-elle une meilleure anticipation des mouvements adverses ? Favorise-t-elle également la coopération et la survie des coalitions ?

2.2 Fonctionnalités et étapes de réalisation

Voici les principales fonctionnalités implémentées et les étapes suivies pour le développement du projet :

Le développement du projet a commencé par la mise en place des bases du jeu : création de la grille, gestion des déplacements, et ajout des joueurs ainsi que des murs qu'ils laissent derrière eux. Ensuite, nous avons intégré les algorithmes d'intelligence artificielle : MaxN, MinMax.

Une fonction d'évaluation heuristique a été ajoutée afin que les IA puissent prendre des décisions plus pertinentes en fonction de leur environnement (par exemple en maximisant l'espace libre autour d'elles). Des scénarios variés ont été générés grâce à la classe `Scenario`, en faisant varier la taille de la grille, la profondeur de recherche ou encore le nombre de joueurs.

Pour observer le comportement des IA, nous avons développé une visualisation en console, puis exporté les résultats des simulations sous forme de fichiers CSV. Ces données ont ensuite été analysées à l'aide de scripts Python permettant de générer des graphiques illustrant l'impact des paramètres sur les performances des joueurs.

Enfin, une extension du projet est prévue pour intégrer l'algorithme SOS, qui introduit une logique de coopération entre les joueurs appartenant à une même équipe.

- **Gestion de la grille de jeu** : création d'une grille de taille variable, ajout des joueurs, gestion des déplacements et des murs laissés par les joueurs.
- **Déplacement automatique des joueurs** : chaque IA se déplace à son tour selon un algorithme donné, et laisse une traînée infranchissable derrière elle.
- **Implémentation des algorithmes d'IA** : intégration de MaxN, chacun adapté à un contexte différent (multi-joueur sans alliances).

- **Évaluation heuristique** : chaque IA prend ses décisions à l'aide d'une fonction d'évaluation basée sur l'espace libre disponible autour d'elle. Une version plus avancée de cette heuristique combine d'autres critères tels que le nombre de directions disponibles, la distance aux adversaires et la position par rapport au centre de la grille.
- **Détection des collisions et fin de partie** : la partie se termine automatiquement lorsqu'il ne reste qu'un seul joueur vivant.
- **Création et gestion de scénarios** : à l'aide de la classe `Scenario`, les parties sont générées avec des tailles de grille, profondeurs et nombres de joueurs différents.
- **Affichage visuel du jeu** : visualisation textuelle dans la console du déroulement des parties, tour par tour.
- **Génération de graphiques** : les résultats des simulations sont analysés via des scripts Python pour représenter visuellement l'impact de la profondeur, de la taille de la grille ou des équipes.
- **Préparation à l'intégration de l'algorithme SOS** : une extension du projet est prévue pour intégrer la logique de coopération entre joueurs d'une même équipe.

2.3 Travaux existants

Les algorithmes que nous utilisons sont des algorithmes classiques qui ont été largement étudiés dans le domaine de l'intelligence artificielle appliquée aux jeux :

- **MaxN :** généralisation du Minimax pour des jeux multi-joueurs sans alliances fixes, souvent utilisée dans les jeux à trois joueurs ou plus, comme le poker multi-joueur.
- **SOS (Socially Oriented Search) :** introduit la coopération entre joueurs d'une même équipe, permettant la modélisation de comportements collectifs et de stratégies coopératives.

3 Élements techniques

3.1 Descriptions des algorithmes

Présentation des algorithmes

- **MAXN** : Contrairement à Paranoid, MAXN est une généralisation du Minimax qui fonctionne pour plusieurs joueurs. Au lieu de considérer tous les autres comme des adversaires, chaque joueur maximise directement sa propre fonction d'évaluation, ce qui permet une prise de décision plus équilibrée dans un contexte multi-joueur. Cet algorithme est particulièrement adapté aux jeux sans alliances fixes.

```
Function MaxN(state, depth, player)
    If depth = 0 or isTerminal(state)
        Return Evaluation(state)  // [score1, score2
           , ..., scoreN]

    bestValue     [-   , -   , ..., -   ]

    For each action in Actions(state, player)
        nextState     ApplyAction(state, action)
        nextPlayer     (player + 1) mod N
        value     MaxN(nextState, depth - 1,
           nextPlayer)

        If value[player] > bestValue[player]
            bestValue     value

    Return bestValue
```

- **SOS (Socially Oriented Search)** : Cet algorithme introduit la notion de collaboration entre joueurs d'une même équipe. Contrairement aux deux précédents, il intègre une dimension sociale en permettant aux coéquipiers de coordonner leurs décisions afin d'optimiser leurs chances de victoire collective. Cela permet de mieux simuler des comportements coopératifs et d'évaluer l'impact des alliances stratégiques sur la performance globale des équipes.

```
Function SOS(state, depth, player)
    If depth = 0 or isTerminal(state)
```

```
        Return Evaluation(state)  // [score1, score2,
            ..., scoreN]

    bestValue     -
    bestVector     null

    For each action in Actions(state, player)
        nextState     ApplyAction(state, action)
        nextPlayer     (player + 1) mod N
        value     SOS(nextState, depth - 1, nextPlayer)

        teamScore     Sum(value for teammates of player)
        If teamScore > bestValue
            bestValue     teamScore
            bestVector     value

    Return bestVector
```

3.2 Structures de données utilisées

Dans notre projet, nous avons principalement utilisé les structures de données suivantes :

- **Tableau bidimensionnel** (`Entite[][]`) : utilisé pour représenter la grille du jeu. Chaque case contient soit un joueur, soit un mur, soit est vide (`null`).
- `ArrayList<Joueur>` : utilisée pour stocker dynamiquement la liste des joueurs présents sur la grille, dans la classe `Grille`.
- `Queue (LinkedList)` : utilisée pour explorer l'espace accessible autour d'un joueur, notamment dans l'évaluation heuristique (par une recherche en largeur - BFS).
- **Objets** (`Joueur`, `Mur`, `Scénario`) : les entités sont représentées sous forme d'objets héritant d'une structure abstraite pour être stockées et manipulées facilement.

Ces structures permettent de gérer efficacement les positions, les déplacements et l'analyse stratégique dans la simulation du jeu.

3.3 Description des données

Les principales données utilisées dans notre projet sont :

- **Grille :** Matrice bidimensionnelle représentant l'état du plateau de jeu. Chaque cellule peut contenir soit un joueur, soit un mur, soit être vide.
- **Joueur :** Caractérisé par :
  - Un numéro unique d'identification.
  - Une position (x, y) sur la grille.
  - Une profondeur de recherche (utilisée par les algorithmes d'IA).
- **Mur :** Représenté uniquement par sa position sur la grille après le passage d'un joueur, rendant cette case infranchissable.
- **Scénario :** Ensemble de paramètres définissant une configuration initiale d'une partie, comprenant :
  - La taille de la grille.
  - La liste des joueurs (avec leurs positions initiales et profondeurs de recherche respectives).

4 Architecture du projet

4.1 Description des paquetages non standards utilisés

Dans ce projet, nous avons utilisé les packages non standards suivants :

- **Jackson (Java)** : bibliothèque permettant de lire et écrire facilement des données au format JSON. Nous l'avons utilisée pour charger automatiquement les scénarios du jeu à partir de fichiers externes.
- **Matplotlib (Python)** : bibliothèque graphique permettant la visualisation des résultats obtenus lors des expérimentations. Elle nous a permis de tracer des graphiques montrant l'impact de la profondeur de recherche et de la taille de la grille sur les performances des IA.
- **CSV** : utilisé pour lire et traiter les fichiers de résultats CSV générés par les simulations du jeu.

4.2 Diagrammes des modules et des classes

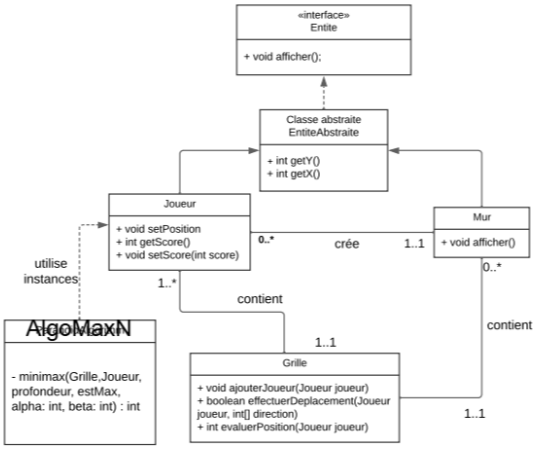


Figure 1: Diagramme des classes

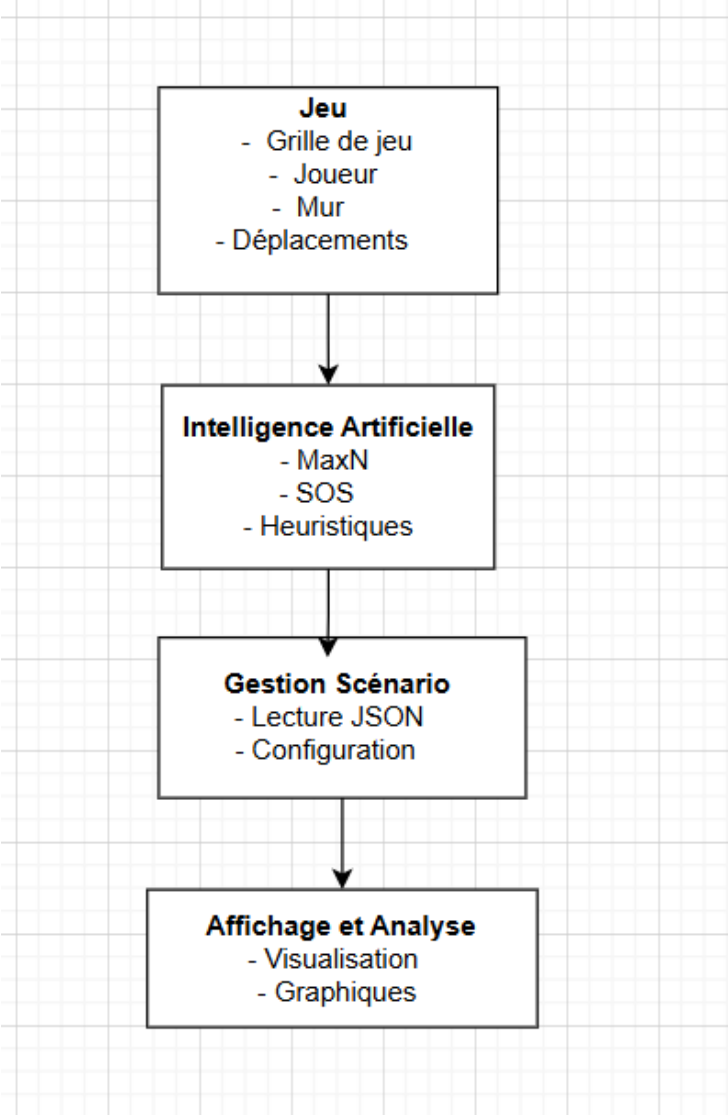


Figure 2: Diagramme des modules

4.3 Chaînes de traitement

La chaîne de traitement de notre projet suit les étapes suivantes :

- **Chargement du scénario** :
  - Lecture d'un fichier JSON via la classe `ScenarioLoader`.
  - Initialisation de la grille et placement des joueurs selon les données du scénario.

- **Prise de décision par les IA (tour par tour)** :
  - on choisit un algorithme de décision.
  - L'IA évalue les coups possibles à l'aide d'une heuristique (ex : espace libre ou nombre de directions disponibles, la distance aux adversaires et la position par rapport au centre de la grille).
  - Le meilleur mouvement est sélectionné.
- **Mise à jour de la grille** :
  - Le joueur se déplace dans la direction choisie.
  - Un mur est placé sur sa position précédente.
  - Détection de collision avec les murs, les bords ou les autres joueurs.
- **Boucle de jeu** :
  - Répétition des tours jusqu'à ce qu'un seul joueur reste en vie.
- **Sauvegarde des résultats** :
  - Enregistrement des informations de la partie (joueurs restants, nombre de tours, profondeur, etc.) dans un fichier CSV.
- **Analyse des résultats** :
  - Lecture des fichiers de résultats avec Python.
  - Génération de graphiques (via `matplotlib`) pour observer l'influence des paramètres comme la profondeur, la taille de la grille ou des équipes.

5 Expérimentations

Le nombre de victoires augmente globalement avec la profondeur jusqu'à un pic à 9, puis chute à 10. Cela montre qu'une profondeur intermédiaire à élevée améliore la stratégie, mais qu'une profondeur trop grande peut devenir contre-productive.

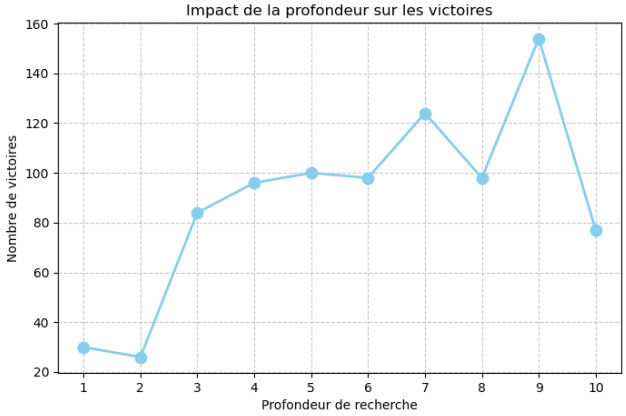


Figure 3: Impact de la profondeur de recherche sur les performances des IA

Le taux de victoire augmente avec la taille de la grille jusqu'à une taille de 15, puis diminue légèrement.

Cela suggère que la stratégie est plus efficace sur des tailles intermédiaires, tandis que les grandes grilles peuvent introduire une complexité supplémentaire rendant les victoires plus difficiles.

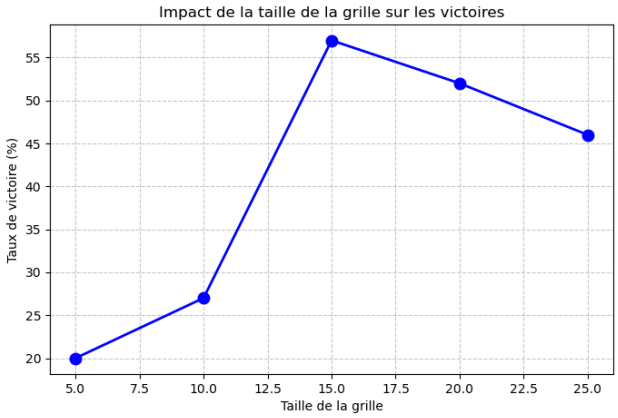


Figure 4: Impact de la taille de la grille sur les performances des IA

Réponse à la question scientifique

La profondeur de recherche a un impact significatif sur les performances des IA dans le jeu de Tron. Une profondeur plus élevée améliore généralement la capacité d'anticipation, ce qui se traduit par de meilleures décisions individuelles. Toutefois, cet effet dépend fortement de la taille de la grille. Sur des grilles trop petites, une grande profondeur est inutile car les possibilités sont rapidement limitées. Inversement, sur des grilles plus vastes, une profondeur plus importante devient un avantage. Ainsi, l'influence de la profondeur est bénéfique jusqu'à un certain seuil, au-delà duquel elle peut entraîner un surcoût en calcul sans gain stratégique notable.

6 Conclusion

En conclusion, notre étude montre que la profondeur de recherche influence fortement les performances des IA dans le jeu de Tron. Une profondeur trop faible nuit à la qualité des décisions, tandis qu'une trop grande peut devenir inefficace. De plus, la taille de la grille joue un rôle clé : une taille intermédiaire semble offrir le meilleur équilibre entre liberté et stratégie. Il est donc essentiel d'adapter les paramètres au contexte pour optimiser les résultats.