Université de Caen Normandie

Rapport

Jeu de Tron multi-joueur

Étudiants :
Wassim DJEHA
Louheb KACED
Massinissa MAOUCHE
Rafik HALIT

Enseignants :
Bruno ZANUTTINI
Bonnet Grégory

Jury :
Bruno ZANUTTINI

29 mars 2024

Table des matières

1 Introduction

1.1 Description générale du projet

Le Jeu de Tron multi-joueur et coalitions s'inscrit dans le cadre de l'exploration des stratégies avancées dans le domaine des jeux à plusieurs joueurs,ce dernier est une variante multi-joueurs du classique jeu du serpent. Le but du projet comprendre comment les robots peuvent jouer intelligemment dans ce genre de jeu où ils doivent rivaliser et parfois coopérer pour gagner. On étudie les interactions stratégiques et les performances des algorithmes d'intelligence artificielle dans un contexte compétitif.

Dans ce jeu, chaque joueur contrôle un point mobile sur une grille fixe et doit naviguer pour éviter les collisions avec les murs, les adversaires ou les bords du plateau. Chaque déplacement laisse derrière lui un mur infranchissable, et l'objectif ultime est de survivre plus longtemps que les autres joueurs.

Le projet vise à étendre les possibilités du jeu en introduisant des aspects multi-joueurs plus complexes. Pour ce faire, nous allons implémenter des algorithmes tels que MAXN et Paranoid et aussi SOS pour la gestion des équipes, et explorer des stratégies de coalition où plusieurs joueurs s'unissent pour atteindre un objectif commun.

Une partie importante de ce projet consiste à évaluer l'impact de différentes profondeurs de recherche sur les performances des joueurs solitaires et des coalitions, en tenant compte de facteurs tels que la taille de la grille et le nombre de joueurs dans chaque équipe. Cette analyse permettra de mieux comprendre les mécanismes sous-jacents aux interactions entre les joueurs et d'identifier les stratégies les plus efficaces dans différentes configurations de jeu.

En résumé, ce projet offre une occasion unique d'explorer les complexités des jeux multi-joueurs et de développer des algorithmes pour relever les défis posés par ces environnements compétitifs.

2 Objectifs du projet

L'objectif principal de ce projet est de mettre en œuvre les algorithmes d'intelligence artificielle spécifiques, notamment MAXN, Paranoid et SOS, pour le jeu de Tron multi-joueur et coalitions. Ces algorithmes constituent les fondements théoriques sur lesquels reposera notre approche pour résoudre les défis posés par le jeu.

En plus de l'implémentation des algorithmes, ce projet vise à réaliser des expérimentations approfondies pour évaluer les performances des joueurs solitaires et des coalitions dans différentes configurations de jeu. Cela impliquera la manipulation de paramètres tels que la profondeur de recherche, la taille de la grille et le nombre de joueurs dans chaque équipe.

Un autre objectif clé est de répondre à la question scientifique du projet, qui porte sur l'influence de la profondeur de recherche sur les performances des joueurs solitaires et des coalitions, en tenant compte des variables telles que la taille des équipes et de la grille. Cette analyse permettra de dégager des conclusions importantes sur les stratégies les plus efficaces dans divers contextes de jeu.

En résumé, les objectifs du projet comprennent l'implémentation des algorithmes MAXN, Paranoid et SOS, la réalisation d'expérimentations pour évaluer les performances

des joueurs et des coalitions, ainsi que la réponse à la question scientifique sur l'impact de la profondeur de recherche dans différentes configurations de jeu.

2.1 Description des grandes étapes et fonctionnalités à implémenter

2.1.1 Partie 1 : Réalisation du diagramme UML

Dans cette phase initiale, nous avons conçu un diagramme UML pour décrire la structure de notre application. Ce diagramme a représenté les différentes classes, leurs attributs et méthodes, ainsi que les relations entre elles. Il a servi de guide pour le développement ultérieur du projet, en fournissant une vue d'ensemble claire de l'architecture à mettre en place.

2.1.2 Partie 2 : Modélisation et mise en place des classes de base

Dans cette étape, nous avons commencé par la modélisation et la mise en place des classes de base nécessaires pour notre jeu. Cela inclut les classes représentant les joueurs, l'état du jeu et les actions possibles. nous avons étendu notre modélisation pour inclure les classes relatives aux algorithmes et aux évaluations de l'état du jeu. De plus, nous avons également inclus les classes du package 'vue' pour la visualisation du jeu avec une interface graphique.

2.1.3 Partie 2 : Implémentation de l'algorithme MaxN et Paranoid

Dans ces sections, nous avons implémenté les algorithmes MaxN et Paranoid pour la prise de décision dans le jeu. MaxN explore les possibilités pour maximiser le score du joueur actuel, tandis que Paranoid simplifie le jeu en supposant une coopération des adversaires pour minimiser le score du joueur principal. Ces approches permettent une prise de décision stratégique efficace en tenant compte des actions des adversaires.

2.1.4 Partie 3 : Implémentation des heuristiques basiques

Dans cette partie, nous avons implémenté diverses heuristiques de base pour évaluer la situation des joueurs dans le jeu. Ces heuristiques comprennent la distance aux joueurs adverses, le nombre de cases vides environnantes, la distance minimale par rapport aux murs, la distance au centre et la distance minimale par rapport aux bords de la grille. Ces métriques sont utilisées pour estimer la qualité d'une position ou d'un coup dans le jeu, et sont intégrées dans notre algorithme pour prendre des décisions stratégiques.

2.1.5 Partie 4 : Implémentation de la méthode de Voronoi pour améliorer l'évaluation du jeu

Dans cette section, nous avons mis en œuvre la méthode de Voronoi pour améliorer l'évaluation de l'état du jeu. Cette approche divise l'espace de jeu en régions autour de chaque joueur, utilisant les tailles de ces régions comme base pour évaluer la situation de jeu.

2.1.6 Partie 5 : Introduction de la notion d'équipes dans le jeu avec l'implémentation de l'algorithme SOS

L'algorithme SOS permet d'amener une dimension stratégique supplémentaire au jeu. Plutôt que de simplement se concentrer sur la survie individuelle, les joueurs doivent désormais collaborer avec leurs coéquipiers pour atteindre un objectif commun. L'algorithme SOS intervient ici en assignant les joueurs à des équipes de manière équilibrée et stratégique, ce qui favorise un jeu plus dynamique et engageant. En intégrant cette fonctionnalité, le jeu gagne en profondeur stratégique et en intérêt pour les joueurs, en encourageant la coopération et la compétition entre les équipes.

2.1.7 Partie 6 : Mise en place des scripts bash pour le lancement des expérimentations

Dans cette section, nous avons élaboré des scripts bash permettant de lancer et d'analyser les résultats des tests dans notre environnement de jeu. Le premier script, nommé script.sh, offre une flexibilité en fixant deux paramètres tout en faisant varier un troisième, tel que la profondeur de recherche, pour évaluer les performances dans diverses configurations, scriptSOS.sh suit le meme principe que le premier. Le dérnier script, scriptProfondeur.sh, orchestre l'exécution des expériences avec différentes profondeurs, assurant ainsi une collecte de données efficace pour une analyse approfondie des performances du système.

2.1.8 Partie 7 : Utilisation de la librairie Matplotlib de Python pour représenter les résultats des expérimentations

Dans cette partie, nous avons employé la librairie Matplotlib de Python pour créer des graphiques illustrant clairement les résultats de nos expérimentations. Cette utilisation nous a permis de visualiser de manière efficace les performances de nos algorithmes dans divers scénarios de jeu, facilitant ainsi l'analyse des données expérimentales. En résumé, l'utilisation de Matplotlib a été cruciale pour une interprétation visuelle des résultats et pour éclairer nos décisions d'optimisation et d'amélioration du jeu.

3 Comment lancer le projet ?

3.1 Lancement du projet

Pour lancer le projet, suivez ces étapes :

1. **Compilation des classes :** Pour compiler toutes les classes, exécutez la commande suivante à partir du répertoire racine du projet :

   ```
   javac -d build src/model/algorithmes/*.java src/model/evaluation/*.java
   src/model/jeu/*.java src/model/main/*.java src/vue/*.java
   ```

2. **Exécution des classes exécutables :** Une fois que toutes les classes ont été compilées, vous pouvez exécuter chaque classe exécutable individuellement :
   — **Exécution de la classe principale DemoParametrable :**

```
java -cp build model.main.DemoParametrable
```

— **Exécution de la classe principale Main :** Pour exécuter la classe principale 'Main' en spécifiant les arguments nécessaires, utilisez la commande suivante :

```
java -cp build model.main.Main <taille_grille> <profondeur_recherche>
<nombre_joueurs>
```

— **Exécution de la classe principale MainSos :** Pour exécuter la classe principale `MainSos` en spécifiant les arguments nécessaires, utilisez la commande suivante :

```
java -cp build model.main.MainSos <taille_grille> <profondeur_recherche>
<nombre_joueurs_par équipe>
```

— **Exécution de la classe principale MainInter :** Pour lancer l'interface graphique du jeu, exécutez la classe principale `MainInter`. Utilisez la commande suivante :

```
java -cp build vue.MainInter
```

3. **Utilisation des fichiers JAR :** Vous trouverez les fichiers JAR dans le dossier `dist`. Une fois que vous avez les fichiers JAR, vous pouvez lancer le projet en utilisant les commandes suivante :

```
java -jar JeuTronConsole.jar
```

Cette commande exécutera le jeu sur le terminal en utilisant le fichier JAR pour la version console du jeu.

```
java -jar JeuTronVueGraphique.jar
```

Cette commande exécutera une partie du jeu avec une interface graphique en utilisant le fichier JAR pour la version avec visualisation graphique du jeu.

4 Règles du jeu et conditions de fin de partie

4.1 Règles du jeu

Le jeu de Tron est une variante multi-joueurs du célèbre jeu du serpent. Sur une grille de taille fixe, chaque joueur contrôle un point qui peut se déplacer dans quatre directions (up, down, left et right). À chaque déplacement, le joueur laisse derrière lui un mur infranchissable.

Les règles principales du jeu sont les suivantes :

— Chaque joueur commence avec une position initiale sur la grille.

— À chaque tour, chaque joueur peut choisir de se déplacer dans l'une des quatre directions possible.

— Les joueurs jouent simultanément, ce qui signifie qu'ils effectuent leurs mouvements en même temps lors de chaque tour.

— En se déplaçant, chaque joueur laisse derrière lui une trace (ou "mur") qui devient infranchissable pour tous les joueurs, y compris pour le joueur lui-même.
— Le but du jeu est d'être le dernier joueur survivant en évitant les collisions avec les murs et les bords de la grille.
— Si un joueur est bloqué et n'a aucune possibilité de déplacement, il est considéré comme mort.
— Le jeu se termine lorsque tous les joueurs, sauf un, sont éliminés ou lorsque tous les joueurs d'une équipe ont perdu.

4.2 Conditions de fin de partie

Les fonctions `isTerminal` et `isTerminalSos` déterminent si l'état actuel du jeu est terminal, c'est-à-dire s'il représente une fin de partie :

— **Mode solo :** La fonction `isTerminal` vérifie si l'état du jeu en mode solo est terminal en vérifiant si un seul joueur ou aucun joueur n'est en vie. Si un seul joueur est en vie ou aucun joueur n'est en vie, l'état est considéré comme terminal. dans le cas ou aucun joueur n'est en vie la partie est nulle.
— **Mode équipe :** La fonction `isTerminalSos` vérifie si l'état du jeu en mode équipe est terminal en vérifiant si tous les joueurs d'une équipe sont morts. Si tous les joueurs d'une équipe sont morts, l'état est considéré comme terminal.

5 Conception du projet

5.1 Architecture du projet

Voyons désormais comment nos classes et paquetages interagissent-ils entre eux.

5.1.1 Diagramme des classes

Dans le cadre de notre diagramme de classes, concernant la modélisation du jeu nous présentons les sous-paquetages 'jeu', 'algorithmes' et 'évaluation' du paquetage 'model' :

— **Paquetage jeu** : Ce paquetage comprend les classes essentielles pour la modélisation et la gestion de l'état du jeu. Ces classes sont cruciales pour représenter les éléments du jeu, gérer les actions des joueurs et évaluer les états du jeu.

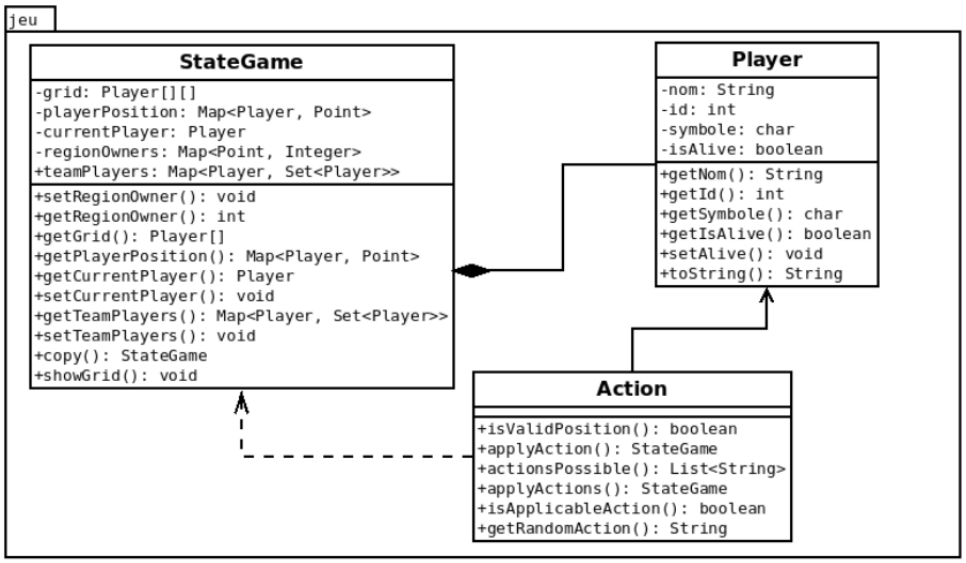


FIGURE 1 – Paquetage jeu.

— **Paquetage algorithmes** : Ce paquetage regroupe les différentes implémentations d'algorithmes utilisés dans le cadre du jeu. Ces algorithmes jouent un rôle crucial dans la prise de décision des joueurs , en évaluant les états de jeu et en choisissant les actions les plus stratégiques. Parmi les algorithmes disponibles, on retrouve notamment l'algorithme MaxN, l'algorithme Paranoid, l'algorithme SOS, et une abstraction de base, AbstractAlgorithmeSearch.

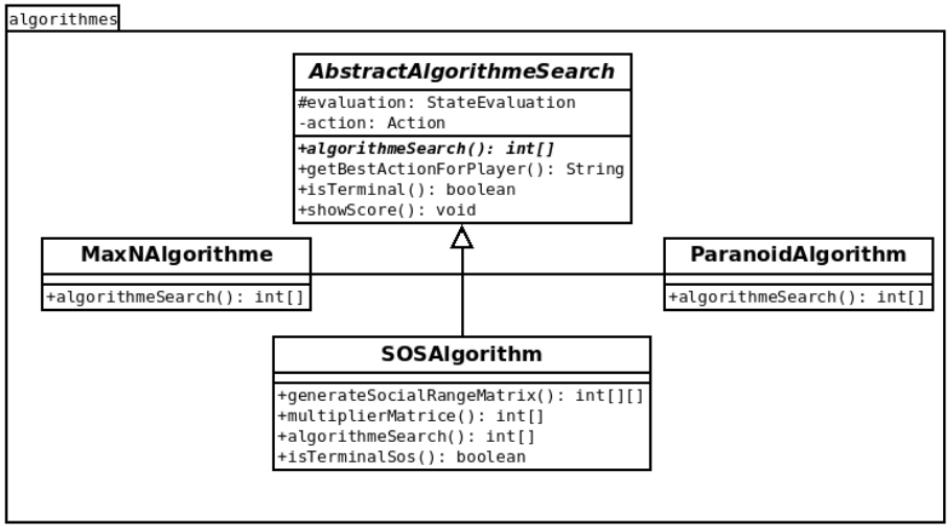


FIGURE 2 – Paquetage algorithme.

— **Paquetage evaluation** : Ce paquetage propose des mécanismes pour évaluer les états de jeu actuels, fournissant ainsi des informations essentielles aux algorithmes de prise de décisions. Il comprend une variété d'implémentations, telles que l'évaluation basique de l'état, la détermination des scores des joueurs, la stratégie Voronoi, ainsi qu'une implémentation de l'évaluation Voronoi, nommée VoronoiStateEvaluation.

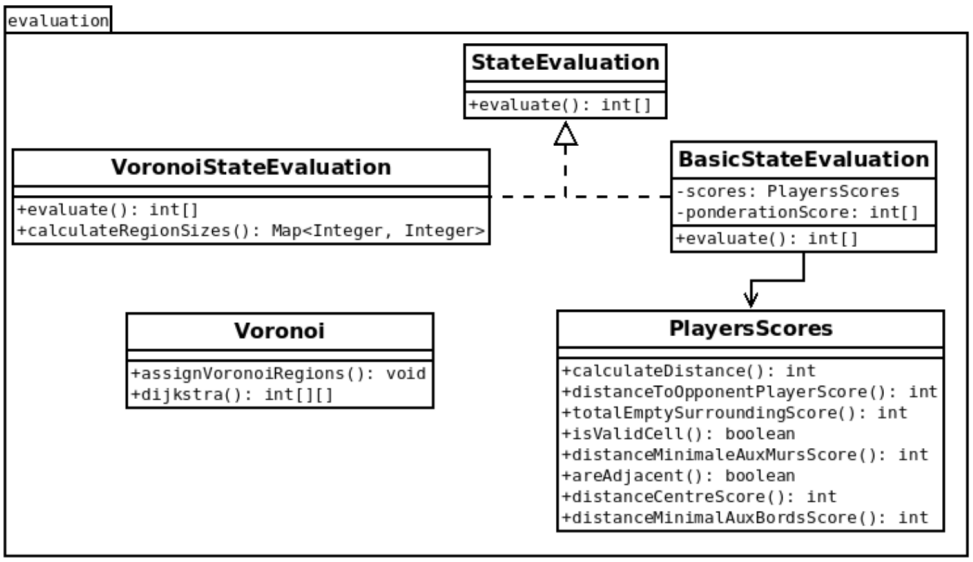


FIGURE 3 – Paquetage evaluation.

— **Paquetage vue** : Concernant la visualisation du jeu, nous présentons le paquetage 'vue', qui inclut les classes GameInterface, MainInter et SplashScreen.

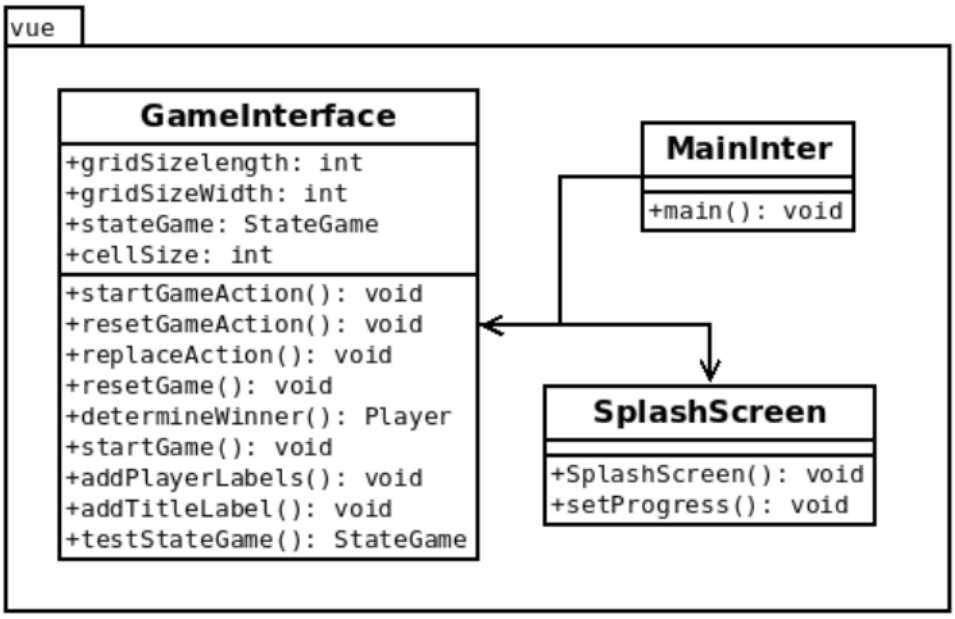


FIGURE 4 – Paquetage vue.

5.1.2 Interface graphique

L'interface graphique du jeu a été développée pour offrir une expérience visuelle aux utilisateurs. Cette interface permet de visualiser l'état actuel du jeu, d'interagir avec les contrôles et de suivre le déroulement de la partie en temps réel.

Composants de l'Interface :
L'interface graphique est composée de plusieurs éléments interactifs :
- **Grille de Jeu** : La zone centrale de l'interface affiche la grille de jeu, où les joueurs évoluent. Chaque joueur est représenté par un cercle coloré, positionné sur la grille en fonction de sa position actuelle.
- **Panneau de Contrôle** : En haut de l'écran, un panneau de contrôle permet aux utilisateurs d'interagir avec le jeu. Il contient des boutons pour démarrer, arrêter, redémarrer, replacer et quitter la partie.
- **Légende des Joueurs** : Sur le côté gauche de l'écran, une légende affiche les noms des joueurs ainsi que les algorithmes qui les contrôlent. Cela permet aux utilisateurs de distinguer les différents joueurs et de comprendre leur mode de fonctionnement.
- **Étiquette du Gagnant** : En bas de l'écran, une étiquette dynamique annonce le nom du joueur gagnant à la fin de la partie, ou indique s'il s'agit d'un match nul.

Fonctionnalités Principales :
L'interface graphique offre les fonctionnalités suivantes :
- **Démarrage et Arrêt de la Partie** : Les utilisateurs peuvent démarrer et arrêter la partie à tout moment en appuyant sur les boutons correspondants dans le panneau de contrôle.
- **Redémarrage de la Partie** : Un bouton permet de redémarrer la partie à partir de zéro, en réinitialisant la grille et en replaçant les joueurs aléatoirement.
- **Remplacement des Joueurs** : Un autre bouton permet de remplacer les joueurs actuels par de nouveaux joueurs positionnés aléatoirement sur la grille.
- **Visualisation en Temps Réel** : L'interface graphique met à jour la grille et les positions des joueurs en temps réel, permettant aux utilisateurs de suivre le déroulement de la partie pendant qu'elle se joue.

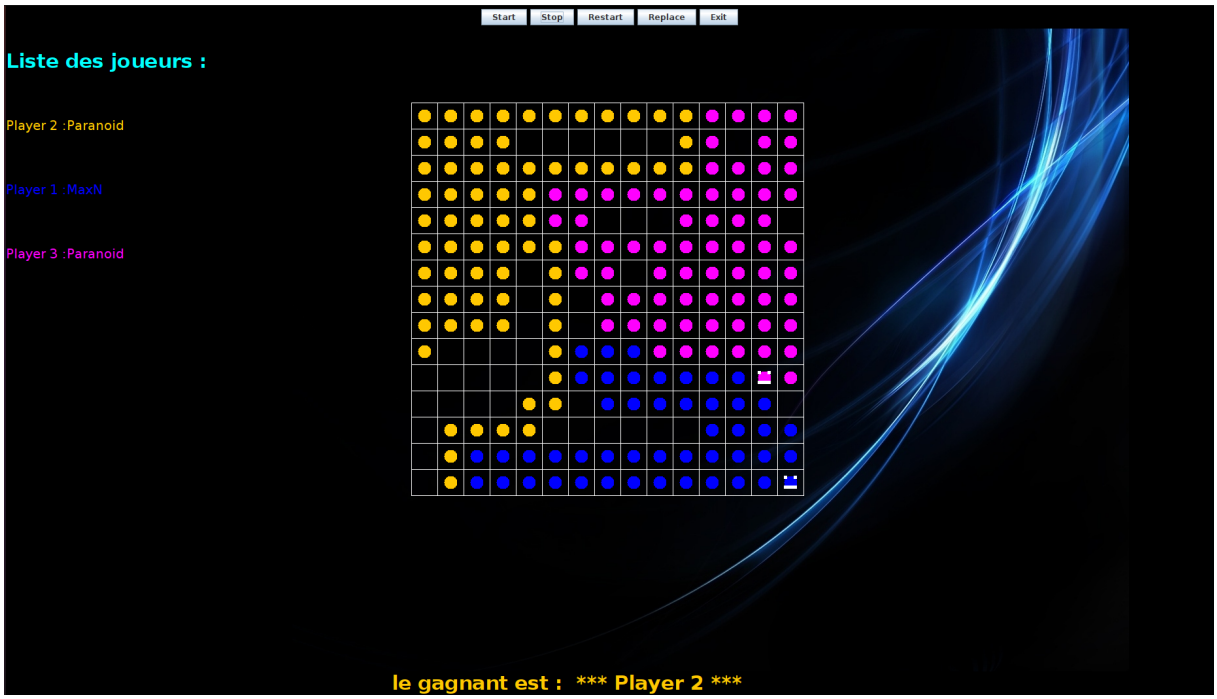


FIGURE 5 – Interface Graphique - Jeu de Tron

6 Algorithme MaxN

L'algorithme MaxN est une méthode de recherche utilisée pour déterminer la meilleure action à prendre dans un jeu à plusieurs joueurs. Dans cette section, nous détaillerons son fonctionnement et son implémentation dans notre projet.

L'algorithme MaxN effectue une recherche récursive dans l'arbre des possibilités de jeu, en évaluant les états du jeu à différentes profondeurs. Il cherche à maximiser le score du joueur actuel tout en prenant en compte les possibilités de déplacements pour les autres joueurs.

6.1 Description de l'algorithme

L'algorithme MaxN fonctionne comme suit :

1. À chaque niveau de profondeur de la recherche, il explore toutes les actions possibles pour le joueur courant.
2. Pour chaque action possible, il simule le résultat de cette action en appelant récursivement l'algorithme sur l'état suivant.
3. Il sélectionne l'action qui maximise le score du joueur courant.
4. Il retourne les valeurs évaluées pour chaque joueur.

6.2 Implémentation dans notre projet

Dans notre projet, l'algorithme MaxN est implémenté dans la classe `MaxNAlgorithme`. Cette classe hérite de la classe abstraite `AbstractAlgorithmeSearch` et implémente la méthode `algorithmeSearch`, qui effectue la recherche récursive selon l'algorithme MaxN.

Voici un pseudo code de notre implémentation de l'algorithme MaxN :

Function `MaxN(`*state, action, voronoi, depth, currentPlayer*`)` **:**
  if `IsTerminal(`*state*`)` *or depth* == 0 **then**
    voronoi.assignVoronoiRegions(*state*);
    return *evaluation.evaluate(state)*;
  end
  playerPositions ← *state*.getPlayerPosition();
  possibleActions ← `ActionsPossible(`*state, currentPlayer*`)`;
  numPlayers ← size of *playerPositions*;
  bestValue ← array of length *numPlayers*;
  for *possibleAction* ***in*** *possibleActions* **do**
    nextState ← copy of *state*;
    nextState ← *action*.applyAction(*nextState*, *possibleAction*, *currentPlayer*);
    value ← `MaxN(`*nextState, action, voronoi, depth - 1, nextPlayer(nextState, currentPlayer)*`)`;
    if *bestValue*[*currentPlayer.getId*()] < *value*[*currentPlayer.getId*()] **then**
      bestValue ← *value*;
    end
  end
  return *bestValue*;

Algorithm 1: Algorithme MaxN

6.3 Arbre MaxN

Ci-dessous, vous pouvez voir une représentation simplifiée de l'arbre de recherche MaxN pour 3 joueurs avec leurs scores :

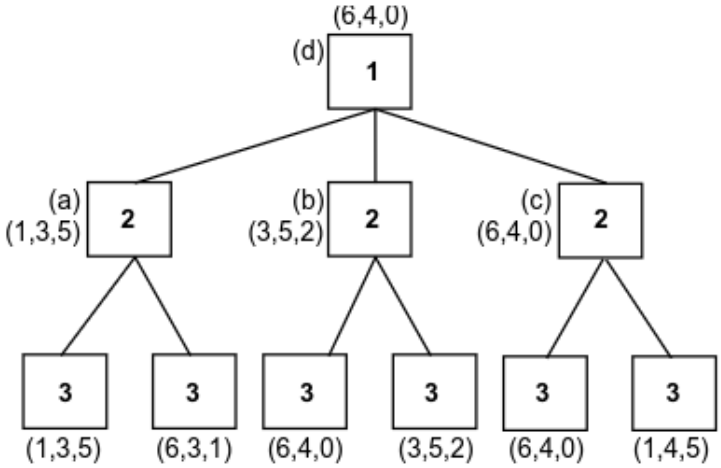


FIGURE 6 – Arbre de recherche MaxN pour 3 joueurs

7 Algorithme Paranoid

L'algorithme Paranoid dans le jeu de Tron repose sur une hypothèse paranoïaque où le joueur principal suppose que les autres joueurs vont coopérer pour minimiser son score. Ainsi, il simplifie le jeu en le ramenant à une confrontation directe entre lui-même et une coalition virtuelle des autres joueurs. Fondé sur cette supposition, le joueur principal prend des décisions visant à minimiser son propre risque, ce qui simplifie l'arbre de jeu et lui permet d'anticiper plus efficacement les actions de ses adversaires. Cette approche offre une stratégie plus claire et cohérente, réduisant la complexité du jeu tout en permettant une meilleure anticipation des réactions adverses.

7.1 Description de l'algorithme

L'algorithme Paranoid fonctionne de la manière suivante :

1. À chaque niveau de profondeur de la recherche, il explore toutes les actions possibles pour le joueur courant.
2. Pour chaque action possible, il simule le résultat de cette action en appelant récursivement l'algorithme sur l'état suivant.
3. Il maintient deux tableaux de scores : l'un pour maximiser le score du joueur principal (max player) quand c'est à lui de jouer et l'autre pour minimiser son score quand il simule les actions des adversaires qui vont essayer de le minimiser.
4. Il sélectionne les meilleures valeurs en fonction du joueur actuel et retourne ces valeurs évaluées pour chaque joueur.

7.2 Implémentation dans notre projet

Dans notre projet, l'algorithme Paranoid est implémenté dans la classe `ParanoidAlgorithm`. Cette classe hérite de la classe abstraite `AbstractAlgorithmeSearch` et implémente la méthode `algorithmeSearch`, qui effectue la recherche récursive selon l'algorithme Paranoid.

Voici un pseudo-code illustrant l'implémentation de l'algorithme Paranoid :

```
Function ParanoidAlgorithm(state, action, voronoi, depth, currentPlayer) :
    if IsTerminal(state) or depth == 0 then
        voronoi.assignVoronoiRegions(state);
        return evaluation.evaluate(state);
    end
    playerPositions ← state.getPlayerPosition();
    possibleActions ← ActionsPossible(state, currentPlayer);
    numPlayers ← size of playerPositions;
    bestValuesMaxPlayer ← array of length numPlayers;
    bestValuesMinPlayer ← array of length numPlayers;
    for possibleAction in possibleActions do
        nextState ← copy of state;
        nextState ← action.applyAction(nextState, possibleAction,
         currentPlayer);
        value ← ParanoidAlgorithm(nextState, action, voronoi, depth - 1,
         nextPlayer(nextState, currentPlayer));
        if currentPlayer.equals(state.getCurrentPlayer()) then
            if bestValuesMaxPlayer[currentPlayer.getId()] <
             value[currentPlayer.getId()] then
                bestValuesMaxPlayer ← value;
            end
        end
        else
            if bestValuesMinPlayer[currentPlayer.getId()] >
             value[currentPlayer.getId()] then
                bestValuesMinPlayer ← value;
            end
        end
    end
    if currentPlayer.equals(state.getCurrentPlayer()) then
        return bestValuesMaxPlayer;
    end
    else
        return bestValuesMinPlayer;
    end
```

Algorithm 2: Algorithme Paranoid

7.3 Arbre de recherche Paranoid

Ci-dessous, vous pouvez voir une représentation simplifiée de l'arbre de recherche Paranoid pour 3 joueurs avec les scores des joueurs :

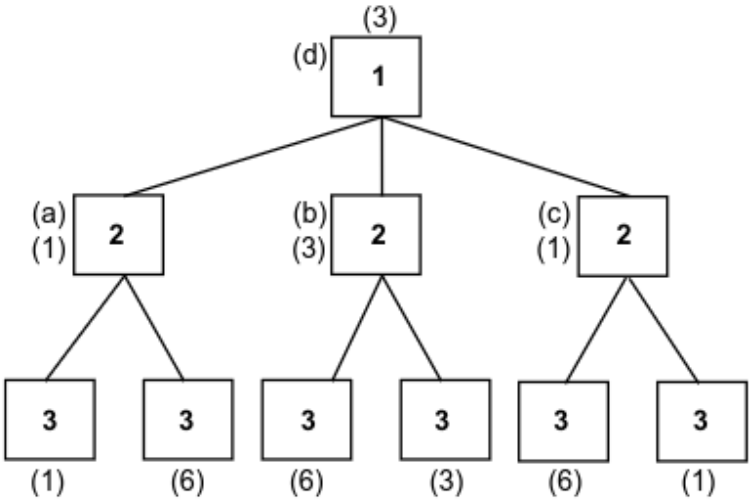


FIGURE 7 – Arbre de recherche Paranoid pour 3 joueurs

8 Algorithme SOS

L'algorithme SOS, (Socially Oriented Search), est un algorithme utilisé dans les jeux multi-joueurs pour assigner les joueurs à des équipes de manière équilibrée et stratégique. Son principe repose sur l'idée de favoriser à la fois la coopération et la compétition entre les joueurs d'une équipes, en prenant en compte les relations sociales entre ces dérniers. En utilisant une approche de recherche basée sur la théorie des jeux, l'algorithme SOS explore différentes combinaisons d'équipes et évalue leurs performances potentielles. En sélectionnant judicieusement les équipes pour chaque joueur, l'algorithme vise à maximiser à la fois la survie individuelle et la réussite collective. Ce faisant, il ajoute une dimension sociale et stratégique supplémentaire aux jeux multi-joueurs, enrichissant ainsi l'expérience de jeu pour les participants.

8.1 Déscription de l'algorithme

L'algorithme SOS fonctionne comme suit :

1. À chaque niveau de profondeur de la recherche, il explore toutes les actions possibles pour le joueur courant.
2. Pour chaque action possible, il simule le résultat de cette action en appelant récursivement l'algorithme sur l'état suivant.
3. Il utilise la matrice de relation entre les joueurs pour évaluer la pertinence de chaque action en termes de cohésion d'équipe.
4. Il sélectionne l'action qui maximise le score du joueur courant, en tenant compte des relations sociales entre les joueurs.
5. Il retourne les valeurs évaluées pour chaque joueur.

8.2 Géneration et utilisation de la matrice de relation entre les joueurs

Dans l'algorithme SOS, une composante clé est l'utilisation d'une matrice de relation entre les joueurs pour évaluer la cohésion d'équipe et guider les décisions stratégiques. Cette matrice représente les liens sociaux entre les joueurs, indiquant quels joueurs sont dans la même équipe et quels joueurs sont adversaires. Chaque ligne et chaque colonne de la matrice correspondent à un joueur, et les valeurs dans la matrice indiquent s'il existe un lien social entre les joueurs correspondants. Pour représenter ce lien on attribut les valeurs suivantes :

1. **Collaboration (valeur 1)** :
   Une valeur de 1 dans la matrice indique une collaboration entre les joueurs correspondants, signifiant qu'ils sont dans la même équipe.
2. **Individualisme (valeur 0)** :
   Indique que deux joueurs sont dans des équipes différentes, soulignant une absence de lien social direct entre eux.
3. **Agression (valeur -1)** :
   Indique un lien négatif entre deux joueurs, suggérant une relation agressive ou conflictuelle. Cette valeur peut être interprétée comme une compétition hostile entre les joueurs

Cette matrice est générée en fonction des règles spécifiques du jeu et peut être mise à jour en fonction du style de jeu choisit.

Pour utiliser cette matrice dans l'algorithme SOS, chaque fois qu'une action est évaluée, l'algorithme multiplie la matrice par un vecteur représentant les performances potentielles de chaque joueur. Cette multiplication permet d'évaluer l'impact de chaque action sur la cohésion d'équipe, en prenant en compte les relations sociales entre les joueurs. Ainsi, l'algorithme peut sélectionner l'action optimale qui maximise le score du joueur courant tout en favorisant les interactions positives au sein de l'équipe.

8.3 Implémentation dans notre projet

Dans notre projet, l'algorithme SOS est implémenté dans la classe SOSAlgorithm. Cette classe hérite de la classe abstraite AbstractAlgorithmeSearch et implémente la méthode algorithmeSearch, qui effectue la recherche récursive selon l'algorithme SOS

Voici un pseudo code de notre implémentation de l'algorithme SOS :

Function `SOS(`*state, action, voronoi, depth, currentPlayer*`)` **:**
  if `isTerminalSos(`*state*`)` *or depth* $== 0$ **then**
    voronoi.assignVoronoiRegions(*state*);
    return
     `multiplierMatrice(generateSocialRangeMatrix(`*state.getTeamPlayers()*`),`
     evaluation.evaluate(state)`)`;
  end
  playerPositions $\leftarrow$ *state*.getPlayerPosition();
  possibleActions $\leftarrow$ `actionsPossible(`*state, currentPlayer*`)`;
  numPlayers $\leftarrow$ size of *playerPositions*;
  bestValue $\leftarrow$ array of length *numPlayers*;
  for *possibleAction* ***in*** *possibleActions* **do**
    nextState $\leftarrow$ copy of *state*;
    nextState $\leftarrow$ *action*.applyAction(*nextState*, *possibleAction*,
     currentPlayer);
    value $\leftarrow$ `SOS(`*nextState, action, voronoi, depth - 1, nextPlayer(nextState,*
     currentPlayer)`)`;
    if $bestValue[currentPlayer.getId()] \leq value[currentPlayer.getId()]$ **then**
      $bestValue \leftarrow value$;
    end
  end
  return $bestValue$;

Algorithm 3: Algorithme SOS

8.4 Arbre SOS

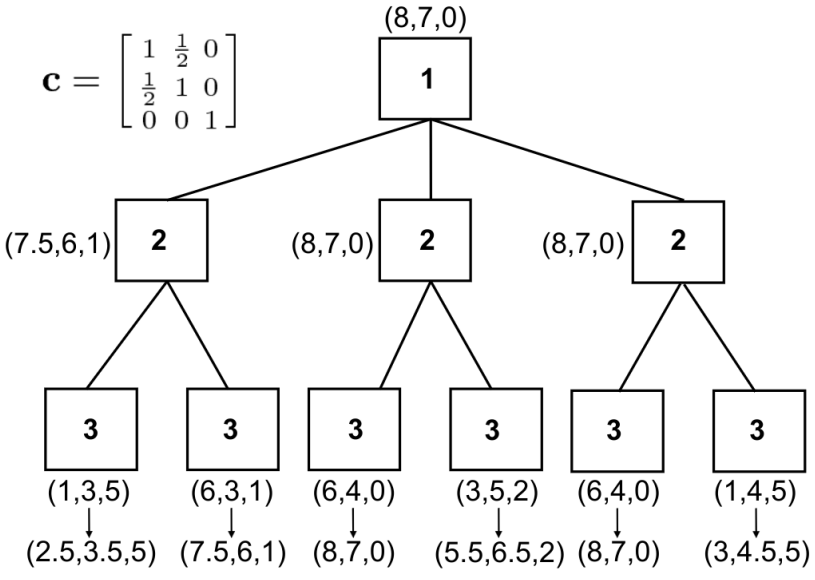


FIGURE 8 – Arbre de recherche SOS pour 3 joueurs , utilisant la matrice $\boldsymbol{c}$

Ci-dessus,l'arbre présente un exemple de recherche en profondeur guidée par la matrice c. Dans cette configuration, les joueurs 1 et 2 démontrent une coopération partielle mais privilégient leur propre intérêt, tandis que le joueur 3 se montre égoïste, ne tenant compte que de son propre bénéfice. L'algorithme SOS sélectionne alors la branche centrale comme étant la meilleure décision, offrant ainsi une utilité perçue de 8 pour le joueur 1.

9 Heuristiques et évaluation du jeu

9.1 Heuristiques de base

Les heuristiques de base sont des méthodes utilisées pour évaluer la situation d'un joueur dans le jeu en fonction de critères spécifiques. Dans notre implémentation, nous utilisons plusieurs heuristiques de base pour estimer la qualité d'un coup ou la position d'un joueur. Voici les principales heuristiques que nous avons utilisées :

- **Distance aux joueurs adverses :** Cette heuristique calcule la distance entre le joueur actuel et les autres joueurs adverses. Un score est attribué en fonction de cette distance, favorisant les positions où le joueur est éloigné des adversaires.
- **Nombre total de cases vides environnantes :** Cette heuristique compte le nombre de cases vides autour du joueur actuel. Un score est attribué en fonction de ce nombre, favorisant les positions entourées de nombreuses cases vides.
- **Distance minimale par rapport aux murs :** Cette heuristique calcule la distance minimale entre le joueur actuel et les murs de la grille. Un score est attribué en fonction de cette distance, favorisant les positions éloignées des murs.
- **Distance au centre :** Cette heuristique calcule la distance entre le joueur actuel et le centre de la grille. Un score est attribué en fonction de cette distance, favorisant les positions proches du centre de la grille.
- **Distance minimale par rapport aux bords :** Cette heuristique calcule la distance minimale entre le joueur actuel et les bords de la grille. Un score est attribué en fonction de cette distance, favorisant les positions éloignées des bords de la grille.

Ces heuristiques de base sont utilisées conjointement dans notre algorithme pour estimer la qualité d'une position ou d'un coup dans le jeu.

9.2 Méthode de Voronoi

Une autre approche d'évaluation de l'état du jeu , nous avons choisi d'utiliser la méthode des régions de Voronoi. Cette méthode nous permet de diviser l'espace de jeu en régions autour de chaque joueur, en attribuant à chaque joueur la région qui lui est la plus proche. les tailles des régions de Voronoi de chaque joueur sont ensuite utilisées comme base pour l'évaluation de l'état actuel du jeu.

- **Compréhension de la méthode des régions de Voronoi :** La méthode des régions de Voronoi est inspirée du concept des diagrammes de Voronoi, également connus sous le nom de tessellations de Voronoi. Elle tire son nom du mathématicien Georgy Voronoi et est largement utilisée dans divers domaines, y compris la cartographie, la biologie, et les jeux.

L'idée fondamentale derrière les régions de Voronoi est de diviser l'espace en régions disjointes, chaque région étant associée à un joueur spécifique. Cette division est telle que chaque point dans une région donnée est plus proche du joueur correspondant que de tout autre joueur sur la carte.

— **Implémentation dans notre évaluation de l'état du jeu :** Dans notre implémentation, nous utilisons la méthode des régions de Voronoi pour segmenter l'espace de jeu en régions autour de chaque joueur. Nous commençons par calculer les distances entre chaque case de la grille et les positions des joueurs à l'aide de l'algorithme de Dijkstra. Cette étape nous permet de déterminer le joueur le plus proche de chaque case.

Une fois que nous avons identifié les joueurs les plus proches pour chaque case, nous attribuons à chaque case le joueur correspondant en tant que propriétaire de la région de Voronoi associée. Si plusieurs joueurs sont à égale distance d'une case, cette dernière n'est pas attribuée à un joueur spécifique, reflétant ainsi une zone de conflit.

Voici un pseudo code de notre implémentation de l'algorithme de Dijkstra :

Input: L'état du jeu *state* et le joueur *player*
Output: Une matrice *distances* contenant les distances de Dijkstra
Function `dijkstra`(*state, player*)
    distances ← nouvelle matrice[*state.grid.length*][*state.grid*[0].*length*] ;
    directions ← {{−1, 0}, {1, 0}, {0, −1}, {0, 1}} ;
    currentGrid ← *state.grid* ;
    currentPosition ← *state.playerPosition*[*player*] ;
    row ← *currentPosition.x* ;
    col ← *currentPosition.y* ;
    for *i* ← 0 **to** *distances.length* − 1 **do**
        for *j* ← 0 **to** *distances*[0].*length* − 1 **do**
            distances[*i*][*j*] ← +∞ ;
        end
    end
    distances[*row*][*col*] ← 0 ;
    queue ← nouvelle file d'attente ;
    queue.enfiler(Point(*row*, *col*)) ;
    while *la file d'attente n'est pas vide* **do**
        current ← *queue*.défiler() ;
        x ← *current.x* ;
        y ← *current.y* ;
        distance ← *distances*[*x*][*y*] ;
        for *direction* ***in*** *directions* **do**
            newX ← *x* + *direction*[0] ;
            newY ← *y* + *direction*[1] ;
            if *estPositionValide(newX, newY, currentGrid)* ***et***
             distances[*newX*][*newY*] = +∞ **then**
                distances[*newX*][*newY*] ← *distance* + 1 ;
                queue.enfiler(Point(*newX*, *newY*)) ;
            end
        end
    end
    return *distances* ;
end

Algorithm 4: Algorithme de Dijkstra

— **Utilisation des tailles de régions pour l'évaluation :** Une fois les régions de Voronoi attribuées, nous utilisons les tailles de ces régions comme base pour évaluer l'état actuel du jeu. Chaque joueur se voit attribuer un score basé sur la taille de sa région de Voronoi respective. Cette approche permet de prendre en compte non seulement la position des joueurs, mais aussi la quantité d'espace qu'ils contrôlent, offrant ainsi une évaluation plus riche et nuancée de la situation de jeu.

— **Avantages et performances :** L'utilisation des régions de Voronoi comme métrique d'évaluation présente plusieurs avantages. Tout d'abord, elle prend en compte la position et l'espace contrôlé par chaque joueur, ce qui permet une évaluation plus précise de l'état du jeu. De plus, notre implémentation de l'algorithme de Dijkstra garantit une évaluation efficace même pour des grilles de jeu de grande taille.

10 Expérimentations et résultats

10.1 Expérimentations

Dans cette section, nous décrivons les expérimentations que nous avons menées pour évaluer les performances de notre environement du jeu.

10.1.1 Script

Dans cette sous-section, nous présentons les scripts utilisés pour mener nos expérimentations.

Script 1 : `script.sh` Ce script est utilisé pour lancer les expérimentations en fixant un paramètre spécifique et en faisant varier deux autres paramètres parmi la profondeur, le nombre de joueurs ou la taille de la grille. Voici une explication détaillée de son fonctionnement :

- Le script prend en entrée trois paramètres, en fixant un paramètres et en faisant varier les deux autres, le script permet de tester différentes configurations du jeu pour évaluer les performances du système dans des conditions variées.
- Après avoir exécuté le programme Java compilé avec la profondeur spécifiée et les autres paramètres, le script capture la sortie et extrait le nom du joueur gagnant.
- Les résultats de chaque expérimentation sont enregistrés dans un fichier de sortie.

Grâce à cette flexibilité, le script permet une exploration approfondie des performances du système en permettant de tester différentes combinaisons de paramètres, ce qui peut être utile pour l'analyse des résultats et l'optimisation du jeu.

Script 2 : `scriptSOS.sh` Ce script suit un fonctionnement similaire à "script.sh", mais ici, les expérimentations sont axées sur les équipes gagnantes du jeu simulé, et les paramètres variables incluent la taille de la grille, la profondeur, et le nombre de joueurs par équipe.

Script 3 : `scriptProfondeur.sh` Permet l'exécution automatisée d'expérimentations sur plusieurs paramètres essentiels du jeu. Voici les éléments principaux que nous varions dans nos expérimentations :

- **Profondeurs de Recherche :** Nous évaluons les performances de notre algorithme en variant les profondeurs de recherche. `depths`.
- **Tailles de Grille :** En testant différentes tailles de grille, nous examinons comment les performances de notre algorithme évoluent en fonction de la taille du terrain de jeu.
- **Tailles d'Équipe :** En variant les tailles d'équipe, nous évaluons comment notre algorithme se comporte dans des configurations de jeu avec différentes dynamiques d'équipe.

Ces scripts ont été conçus pour automatiser le processus d'exécution des expérimentations et pour recueillir les résultats de manière efficace et cohérente.

10.2 Résultats

Pour répondre à la question scientifique sur l'influence de la profondeur de recherche sur le jeu en fonction de la taille des équipes et de la taille de la grille, plusieurs expérimentations ont été réalisées. Les résultats de ces expérimentations sont présentés ci-dessus par des histogrammes et des graphes.

Dans ces expérimentations, nous avons d'abord varié la profondeur de recherche, puis la taille de la grille, et enfin la taille des équipes. Les observations issues de ces variations sont décrites ci-dessous.

10.2.1 Variation de la profondeur de recherche

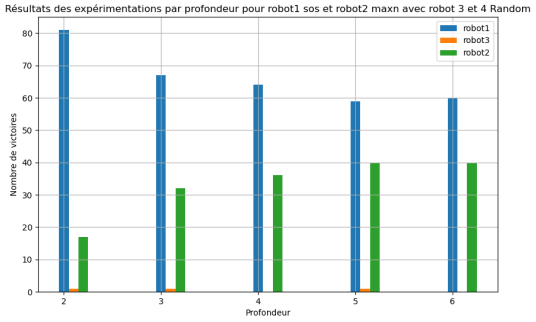
En ce qui concerne la profondeur de recherche, nous avons obtenu plusieurs résultats spécifiques pour chaque algorithme, mais nous ne pouvons pas conclure grand-chose à partir de la profondeur seule, car elle est liée à d'autres facteurs tels que la taille de la grille et la taille des équipes. Même la manière de positionner les joueurs influence cela. Cependant, nous pouvons remarquer certains effets. Par exemple, dans une expérimentation où nous avons fait jouer MaxN et SOS avec deux robots aléatoires, nous avons remarqué que lorsque la profondeur augmente, MaxN performance de MaxN et gagne davantage de parties, et la performance de Sos baisse. De même, dans le cas où nous avons fait jouer MaxN et Paranoid avec deux robots aléatoires, c'est MaxN qui est toujours plus performant quelle que soit la profondeur de recherche.

10.2.2 Variation de la taille de la grille

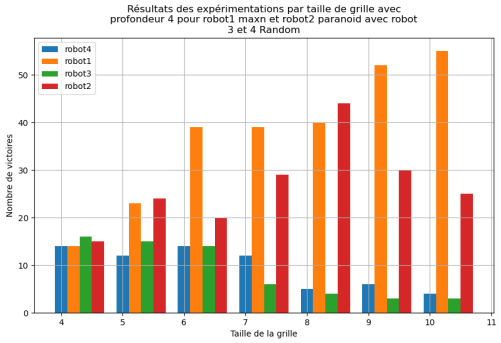
En faisant varier la taille de la grille, on a remarqué qu'à chaque fois que la taille de la grille augmente, MaxN gagne plus de parties, c'est-à-dire qu'il devient plus performant. Par exemple, nous avons réalisé des expérimentations où un robot a joué avec MaxN et un autre avec Paranoid, ainsi que deux robots aléatoires. À chaque augmentation de la taille de la grille, le nombre de victoires de MaxN augmente. Dans une autre expérimentation où MaxN, Paranoid, SOS et un robot aléatoire ont joué, nous avons également observé que MaxN gagne le plus souvent par rapport aux autres à mesure que la grille devient de plus en plus grande.

10.2.3 Variation de la taille des équipes

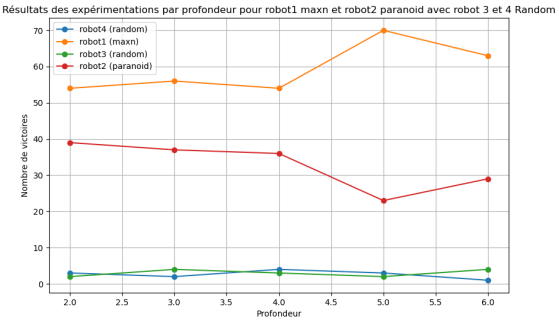
En ce qui concerne la variation de la taille des équipes, nous avons lancé plusieurs expérimentations où nous avons fait varier la taille des équipes pour différentes profondeurs. Nous avons remarqué que l'équipe qui utilise SOS gagne le plus souvent, ce résultat étant conforme à nos attentes. En effet, l'équipe qui joue avec SOS bénéficie d'une coordination entre ses joueurs, ce qui n'est pas le cas pour les autres équipes. Même si la taille de l'équipe change, l'équipe SOS reste la plus performante.



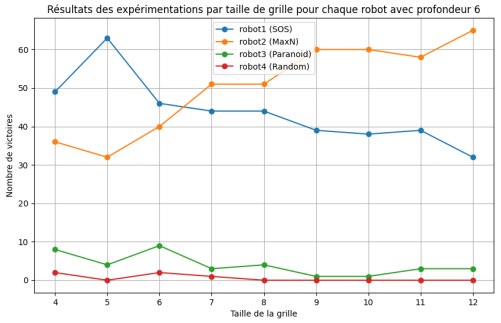
(a) **Résultats sur quatre joueurs : un MaxN, un SOS et deux aléatoires, sur 100 expérimentations pour chaque profondeur.La profondeur de recherche varie à chaque étape.**



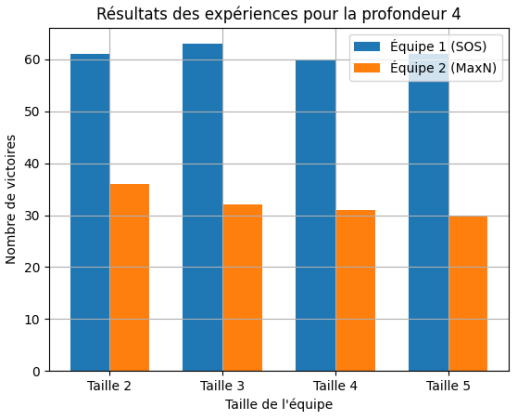
(b) **Résultats sur quatre joueurs : un MaxN, un Paranoid et deux aléatoires, sur 100 expérimentations pour chaque taille. À chaque étape, la taille de la grille a été variée.**



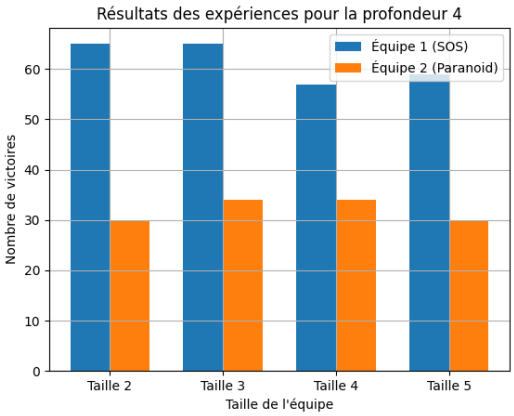
(c) **Résultats sur quatre joueurs : un MaxN, un Paranoid et deux aléatoires, sur 100 expérimentations pour chaque profondeur.La profondeur de recherche varie à chaque étape.**



(d) **Résultats sur quatre joueurs : un MaxN, un Paranoid et , un SOS et un aléatoire, sur 100 expérimentations pour chaque taille. À chaque étape, la taille de la grille a été variée.**



(e) **Résultats pour deux équipes : une avec SOS, l'autre avec Maxn, sur 100 expérimentations pour chaque taille. La taille des équipes varie à chaque étape.**



(f) **Résultats pour deux équipes : une avec SOS, l'autre avec Paranoid, sur 100 expérimentations pour chaque taille. La taille des équipes varie à chaque étape**

Enfin, nos expérimentations ont mis en lumière l'impact significatif de la profondeur de recherche, de la taille de la grille et de la taille des équipes sur les performances des algorithmes dans le jeu. Nous avons observé que MaxN tend à devenir plus performant avec une taille de grille plus grande, tandis que l'équipe utilisant SOS a démontré une efficacité constante, bénéficiant d'une meilleure coordination entre ses membres et montrant ainsi l'efficacité de l'algorithme SOS lorsqu'on joue avec des équipes.

11 Conclusion

11.1 Récapitulatif de la problématique et de la réalisation

Dans ce projet, nous avons abordé le défi de développer des algorithmes d'intelligence artificielle pour le jeu de Tron multi-joueur et coalitions. Notre objectif principal était de mettre en œuvre des stratégies efficaces basées sur des méthodes telles que MAXN, Paranoid et SOS, tout en explorant l'utilisation de la méthode Voronoi pour améliorer l'évaluation du jeu en plus des heuristiques de base. nous avons accordé une attention particulière aux expérimentations, les considérant comme un pilier essentiel pour évaluer et comparer les performances de nos joueurs solitaires et des coalitions. Ces expérimentations ont été menées dans diverses configurations de jeu, couvrant une gamme étendue de paramètres tels que la profondeur de recherche, la taille de la grille et le nombre de joueurs. Ces tests approfondis nous ont permis de recueillir des données précieuses sur la manière dont nos algorithmes se comportent dans des conditions variées.

11.2 Propositions d'améliorations

Bien que notre projet ait atteint ses objectifs initiaux, plusieurs axes d'amélioration peuvent être envisagés pour renforcer l'ensemble du projet et augmenter son efficacité globale :

- Nous pourrions explorer des variantes plus sophistiquées des algorithmes existants, telles que des améliorations de Maxn prenant en compte des facteurs de coupure alpha-bêta.
- L'amélioration de l'interface utilisateur pourrait offrir une meilleure expérience de jeu, avec des options plus riches et un design visuel plus attrayant.
- Nous pourrions également envisager d'explorer des techniques d'évaluation plus sophistiquées ou novatrices afin d'améliorer la précision et l'efficacité de l'évaluation des performances des joueurs dans le jeu.
- L'intégration de techniques d'apprentissage automatique pour l'adaptation dynamique des stratégies en fonction du comportement des adversaires, représentent des pistes prometteuses pour améliorer l'efficacité de nos joueurs dans des environnements de jeu complexes et changeants.

12 Références

1 **Maxn et Paranoid**
Utilisation : Implémentation des algorithmes Maxn et Paranoid
URL : `https://cdn.aaai.org/AAAI/2000/AAAI00-031.pdf`

2 **A Comparison of Algorithms for Multi-Player Games**
Utilisation : Implémentation des algorithmes Maxn et Paranoid
URL : `https://webdocs.cs.ualberta.ca/~nathanst/papers/comparison_algorithms.pdf`

3 **Algorithme Sos**
Utilisation : Implémentation de l'algorithme Sos
URL : `https://www.cs.umd.edu/~nau/papers/wilson2011modeling.pdf`

4 **Google AI Challenge post-mortem**
Utilisation : Implémentation de Voronoi pour l'évaluation du jeu
URL : `https://www.a1k0n.net/2010/03/04/google-ai-postmortem.html`

5 **Voronoi diagram - Wikipedia**
Utilisation : Implémentation de Voronoi pour l'évaluation du jeu
URL : `https://en.wikipedia.org/wiki/Voronoi_diagram`

6 **GitHub Repository - a1k0n/tronbot**
Utilisation : Implémentation de Voronoi pour l'évaluation du jeu
URL : `https://github.com/a1k0n/tronbot/`

7 **Matplotlib**
Utilisation : Utilisation de la librairie Matplotlib pour les expérimentations
URL : `https://matplotlib.org/`