



# Projet Tutoré - Rapport de fin de projet

Conception et développement d'une simulation de capture  
de drapeau et mise en place d'agent autonome

HIRTZ - JEAN-BAPTISTE - JUNG - LE NALINEC - NAIGEON  
RA-IL2

Tuteur : Amine BOUMAZA  
2024/2025

<b>Introduction.....</b>	<b>3</b>
Présentation du projet.....	3
Rappel rapide des règles du jeu CTF :.....	3
Présentation de l'équipe et du rôle de chacun.....	4
Planning.....	4
<b>Analyse &amp; Conception.....</b>	<b>5</b>
Moteur.....	5
Architecture Moteur.....	6
Simulation par tours.....	7
Affichage.....	7
Limites du MVC classique.....	8
Refonte du modèle MVC.....	8
Spécificité de conception du MVC.....	9
Intelligence Artificielle.....	10
Evolution par rapport à l'étude préalable.....	10
<b>Réalisation.....</b>	<b>11</b>
Liste des fonctionnalités ajoutées.....	11
Réalisations logicielle.....	13
La simulation.....	14
L'apprentissage.....	18
L'éditeur de carte.....	20
Intelligence artificielle.....	21
Structure d'un agent.....	21
Entrées (perceptions).....	21
Sorties (actions).....	21
Modèle Aléatoire.....	22
Modèle Arbre de décision.....	22
Modèle Réseau de neurones.....	23
ECJ (Evolutionary Computation for Java).....	23
Fonction d'évaluation.....	24
Réseaux de neurones récurrent.....	25
Modèle : Humain.....	26
Fonctionnalités que l'on aurait aimé ajouter.....	26
Tests de validation.....	27
Difficultés rencontrées.....	27
ECJ.....	27
Bugs et "Edge-Case".....	28
<b>Conclusion.....</b>	<b>29</b>
<b>Annexe.....</b>	<b>30</b>
Lancement du projet.....	30



# Introduction

Ce document a pour objectif de présenter notre projet tutoré intitulé “Conception et développement d'une simulation de capture de drapeau et mise en place d'agent autonome”. Nous y détaillons le concept du jeu de capture du drapeau, notre analyse et la réalisation de l'application avec les différentes fonctionnalités.

## Présentation du projet

Notre projet tutoré a pour objectif de concevoir et développer une simulation de jeu de capture de drapeau (ou “capture the flag” en anglais, CTF) en Java. Ce jeu qui oppose deux équipes, consiste à s'introduire dans le camp adverse pour s'emparer de son drapeau et le ramener dans son propre territoire, tout en défendant son propre drapeau. Ce principe allie stratégie offensive et défensive, ainsi qu'une coordination étroite entre les membres de l'équipe.

Le but du projet tutoré est donc de développer une simulation informatique afin d'expérimenter diverses méthodes d'intelligence artificielle afin d'optimiser la performance des agents impliqués dans le jeu.

Ce document compile notre travail après 7 itérations, nous allons revenir sur nos avancements, nos réalisations et nos difficultés.

## Rappel rapide des règles du jeu CTF :

- Deux équipes s'affrontent, chacune possédant un drapeau placé dans son territoire.
- L'objectif est de capturer le drapeau de l'équipe adverse et de le ramener dans son propre territoire.
- Les joueurs peuvent être touchés et donc retirés temporairement de la partie lorsqu'ils ne se trouvent pas sur leur territoire.
- Si un joueur portant un drapeau est touché, il doit laisser le drapeau au sol.
- Une équipe gagne la partie lorsqu'elle ramène avec succès le drapeau adverse dans sa base.
- Il faut à la fois attaquer et défendre : protéger son propre drapeau tout en essayant de capturer celui de l'autre.

Nous avons rajouté des éléments supplémentaires dans notre simulation :

- Jeu à plus de 2 équipes.
- Plusieurs drapeaux par équipe (pour remporter la partie, il faut récupérer tous les drapeaux adverses).
- Zone neutre (si deux ennemis se rencontrent dans cette zone, les deux meurent).

## Présentation de l'équipe et du rôle de chacun

Nous sommes **5 étudiants** dans notre équipe et nous avons développé différentes fonctionnalités seul ou parfois ensemble :

- **Grégoire HIRTZ** a travaillé sur les agents, l'application et la simulation.
- **Evan JEAN-BAPTISTE** a travaillé sur le moteur de jeu, les perceptions et les agents contrôlés par des humains.
- **Damien JUNG** a travaillé sur l'apprentissage des agents et les perceptions.
- **Tibère LE NALINEC** a travaillé sur l'application, la simulation, le réseau de neurones et l'éditeur de carte.
- **Adrien NAIGEON** a travaillé sur le moteur de jeu, les perceptions, l'agent avec arbre de décision et l'agent avec le réseau de neurone.

## Planning

	Sujet	Simulateur	Perceptions	Arbre Décisions	Réseaux neurones	Bonus
Étude Préalable						
Itération 1						
Itération 2						
Itération 3						
Itération 4						Editeur de carte
Itération 5						Contrôles clavier
Itération 6						Contrôles manettes
Itération 7						

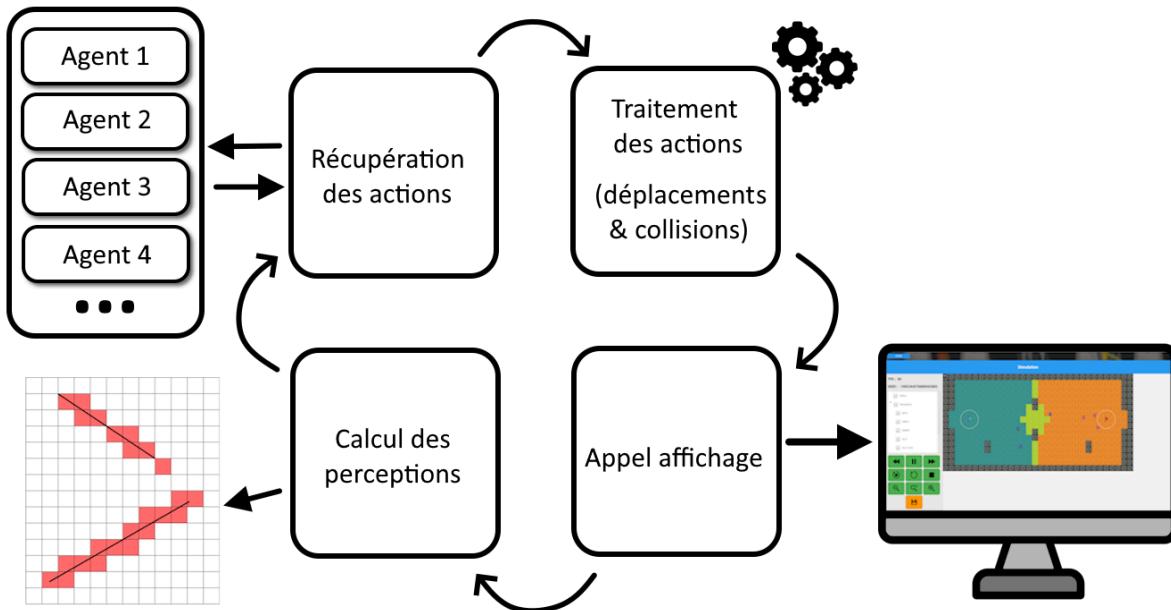


# Analyse & Conception

## Moteur

Le moteur constitue la partie centrale de l'application. Il intègre la boucle de simulation, chargée de gérer le TPS (Tours Par Seconde), ainsi que l'ensemble du code métier qui définit les règles de la simulation. En tant qu'élément charnière, nous avons porté une attention particulière à sa conception afin de garantir sa robustesse et sa pérennité.

Dès le départ, une contrainte majeure a guidé notre réflexion : l'usage du simulateur dans un contexte d'entraînement non supervisé. Cela impliquait un objectif clair : atteindre le plus grand nombre de tours par seconde possible. Pour y parvenir, nous avons fait le choix de dissocier complètement le moteur de l'affichage graphique, permettant ainsi l'exécution de la simulation sans interface utilisateur.



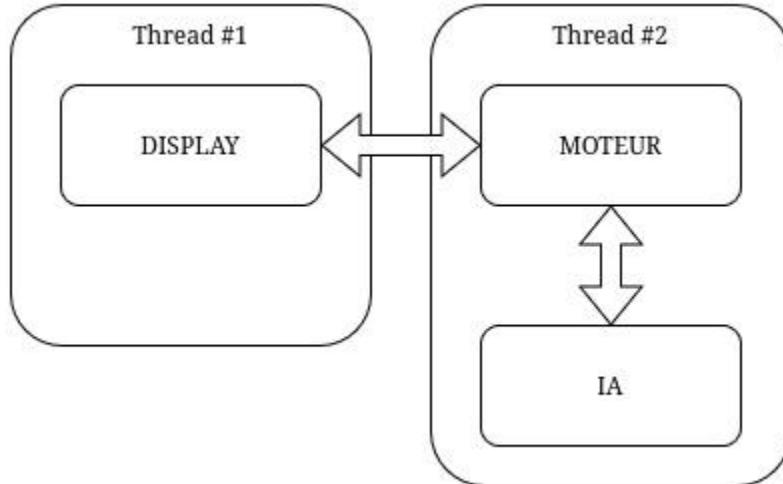
## Architecture Moteur

Lors de la construction de l'objet Java relatif au moteur, une référence à un objet Display peut être fournie. Cet objet, optionnel, est utilisé uniquement si l'on souhaite visualiser la simulation. Le moteur va appeler cet objet à chaque tour pour envoyer les informations du nouveau tour, le display va alors mettre à jour sa zone d'affichage dans l'interface (un Pane) qu'il lui a été attribué à la construction.

Nous avons ainsi mis en place deux boucles distinctes pour l'application, chacune exécutée indépendamment :

- une boucle pour le moteur, dédiée à la simulation et à l'application des règles qui tourne dans un thread séparé,
- une boucle JavaFX, responsable de l'interface utilisateur.

Grâce à cette architecture, la boucle JavaFX gère exclusivement les éléments de l'UI, tandis que la boucle du moteur prend l'initiative de mettre à jour l'affichage via la référence au Display, sans attendre la boucle graphique. Cela rend le moteur totalement autonome : on peut exécuter la simulation avec ou sans affichage, sans aucun blocage entre les threads.



## Simulation par tours

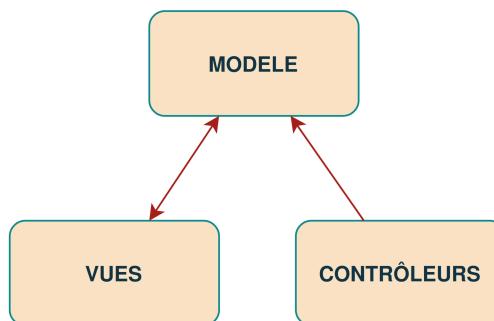
Le moteur fonctionne sur un modèle par tours, c'est-à-dire qu'à chaque mise à jour, le temps avance d'un pas fixe. Ce choix implique une baisse de fluidité lors de la visualisation, car les agents se déplacent par "sauts" visibles. En revanche, cela assure une stabilité totale : quelle que soit la vitesse d'exécution, la simulation reste entièrement déterministe, permettant notamment de rejouer une partie à l'identique à partir d'une graine aléatoire.

Ce fonctionnement simplifie également l'exécution rapide de la simulation : il suffit de ne pas limiter le nombre de TPS pour que le moteur tourne à la vitesse maximale autorisée par la machine.

## Affichage

Dès le début, nous avons naturellement opté pour l'utilisation du modèle MVC (Modèle-Vue-Contrôleur) et lors des premières itérations de développement, nous avons adopté une version classique de ce modèle :

- un modèle unique chargé de stocker et gérer l'ensemble des données de l'application,
- des vues s'actualisant à partir des données du modèle,
- des contrôleurs pour interpréter les actions utilisateur et modifier les données.



Cependant, cette architecture s'est rapidement heurtée à plusieurs limites, et c'est pourquoi le MVC a entièrement été refait dès l'itération 3 pour partir sur de meilleures bases.

## Limites du MVC classique

Le modèle centralisé, bien qu'efficace pour de petites applications, devient difficile à maintenir dans un projet plus ambitieux comme le nôtre. En effet, notre application se compose de plusieurs grandes vues indépendantes (simulation, éditeur de carte, entraînement), chacune composée de sous-vues complexes. Cette organisation rend le modèle global trop volumineux et son utilisation fastidieuse.

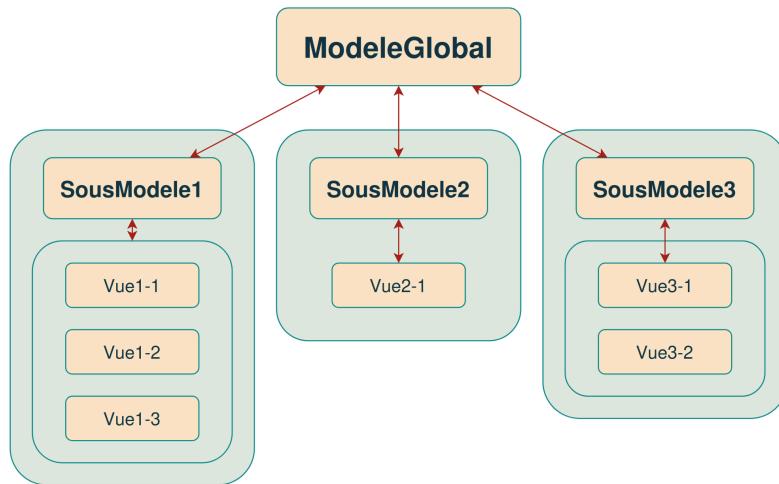
À cela s'ajoute une autre contrainte : JavaFX, bien qu'efficace pour créer des interfaces, n'offre pas facilement des options esthétiques modernes. Ces deux problèmes nous ont amenés à repenser entièrement notre approche du MVC.

## Refonte du modèle MVC

La solution que nous avons adoptée repose sur un principe simple : découper le modèle global en plusieurs sous-modèles spécialisés, chacun dédié à une grande partie de l'application.

Voici l'organisation que nous avons mise en place :

- un modèle global, contenant uniquement les informations communes à toute l'application,
- des sous-modèles spécifiques à chaque module majeur (simulation, éditeur, entraînement),
- des vues et contrôleurs associés à ces sous-modèles, limitant ainsi les dépendances croisées.



## Spécificité de conception du MVC

Tout d'abord, avec JavaFX, la création des vues devient rapidement fastidieuse si tout le monde ne respecte pas une rigueur stricte. En effet, créer les vues directement en Java pur nécessite de nombreuses lignes de code, ce qui complique considérablement la lecture et le débogage des fichiers liés aux interfaces. De plus, comme évoqué précédemment, l'utilisation d'un modèle unique rend cet élément très volumineux et difficile à lire, avec de nombreux attributs inutiles pour certaines vues, ce qui engendre plusieurs sources de problèmes.

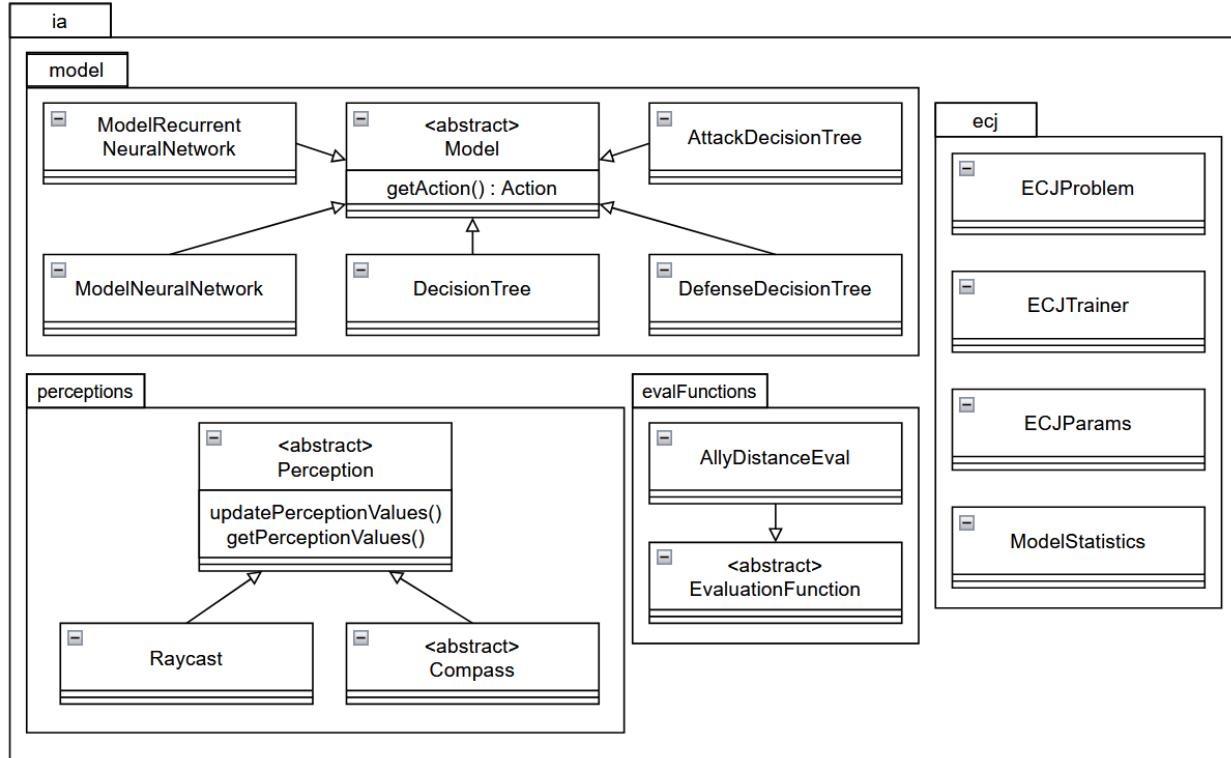
C'est pourquoi cette nouvelle version du MVC utilise JavaFXML. Cette refonte introduit une méthode statique permettant de charger automatiquement les fichiers FXML sans avoir à en spécifier manuellement le nom. Cela nous permet de concevoir l'ensemble des vues statiques avec JavaFXML, ce qui améliore grandement la lisibilité et la modularité du code.

Ensuite, nous avons mis en place un multiton pour les modèles. En effet, pour les trois sous-modèles, nous utilisons un multiton afin de récupérer l'instance correspondant au sous-modèle demandé. Ainsi, nous conservons une référence à chaque objet relatif à un sous-modèle, ce qui nous permet de passer de l'un à l'autre sans perdre les données associées. Pour réinitialiser un sous-modèle, il suffit simplement de le demander explicitement via une méthode dédiée du multiton.

Grâce à cette approche, nous pouvons naviguer facilement entre les différentes parties de l'application. Par exemple : un utilisateur peut lancer un entraînement, ne pas trouver la carte adéquate, revenir au menu principal, ouvrir l'éditeur de carte pour créer une nouvelle carte, puis relancer l'entraînement – et retrouver toutes ses données inchangées, y compris la sous-page sur laquelle il s'était arrêté.

# Intelligence Artificielle

La partie intelligence artificielle de notre projet englobe tous les concepts liés à la réception et au traitement de perceptions, mais également le code pour l'entraînement de nos réseaux de neurones.



Nous y avons regroupé les notions de modèles, et de perceptions, qui sont fortement liées au fonctionnement des agents, chaque agent dispose d'un modèle, et chaque modèle possède ses propres perceptions.

Les perceptions sont réparties en 2 catégories, les rayons (RayCast) et les boussoles (Compass), les rayons servent de vision directe, ils permettent aux agents de s'informer sur les objets et entités présentes dans leur environnement tandis que les boussoles servent plutôt à donner la direction et la distance de l'objet pointé.

## Evolution par rapport à l'étude préalable

Par rapport à l'étude préalable, nous n'avons pas beaucoup de modifications importantes. Nous avons gardé les fondements globaux que nous avions imaginé.

Nous avons seulement eu du mal à nous tenir au planning prévu. Nous avons mis plus de temps que prévu pour développer les fonctionnalités de l'application. Nous n'avons cependant pas pu développer certaines fonctionnalités supplémentaires par manque de temps.

# Réalisation

## Liste des fonctionnalités ajoutées

Voici une liste de toutes les fonctionnalités que nous avons développées dans notre projet, regroupées par catégories.

### Application :

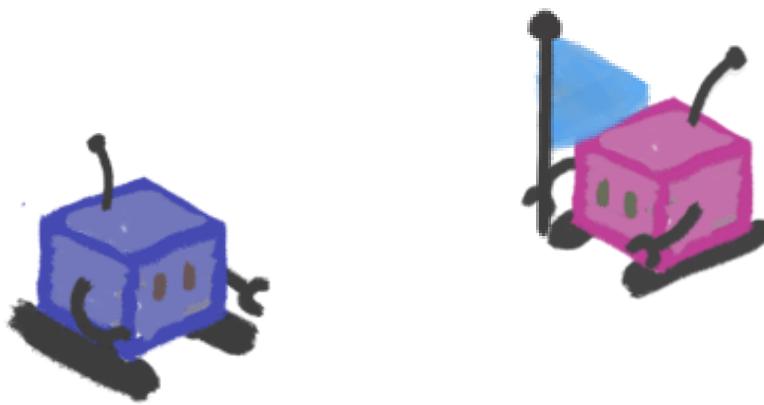
- Menu de simulation
  - Choix d'une carte
  - Choix des paramètres de la partie
  - Choix des modèles d'agents utilisés
  - Choix des joueurs humains si voulu (quel joueur utilise quel contrôleur)
  - Simulation avec interactivité (avance rapide, pause, restart, sauvegarde d'une partie, affichage des perceptions des agents et des zones de collisions)
  - Chargement d'une partie précédente
- Menu apprentissage
  - Choix des cartes à utiliser
  - Choix des paramètres des parties
  - Choix des perceptions
  - Choix du nom du modèle
  - Création du réseau de neurones (fonction d'activation, nombre de couche)
  - Apprentissage
    - Affichage de la fitness maximum, moyenne et minimum
    - Affichage de la "condition number"
    - Affichage du sigma
    - Sauvegarde automatique du modèle
- Menu Editeur de carte
  - Modification d'une carte existante
  - Création d'une nouvelle carte
    - Choix du type de case, à quelle équipe elle appartient
    - Choix du nom de la carte
    - Sauvegarde de la carte
    - Vérificateur de la validité d'une carte (toutes les cases présentes et utilisation de A\* pour avoir un chemin entre les cases importantes))
- Musiques différentes selon le menu et sons joués pendant une partie

## IA :

- Perceptions (boussoles et raycast)
- Modèle aléatoire
- Modèle arbre de décision
- Modèle réseau de neurones à entraîner (implémentation ECJ, fonction d'évaluation et récupération des statistiques d'apprentissage)

## Moteur & Display :

- Moteur et boucle de jeu
  - Gestion des collisions
  - Récupération des actions des agents
- Affichage de la carte et des joueurs et objets dessus



## Réalisations logicielles

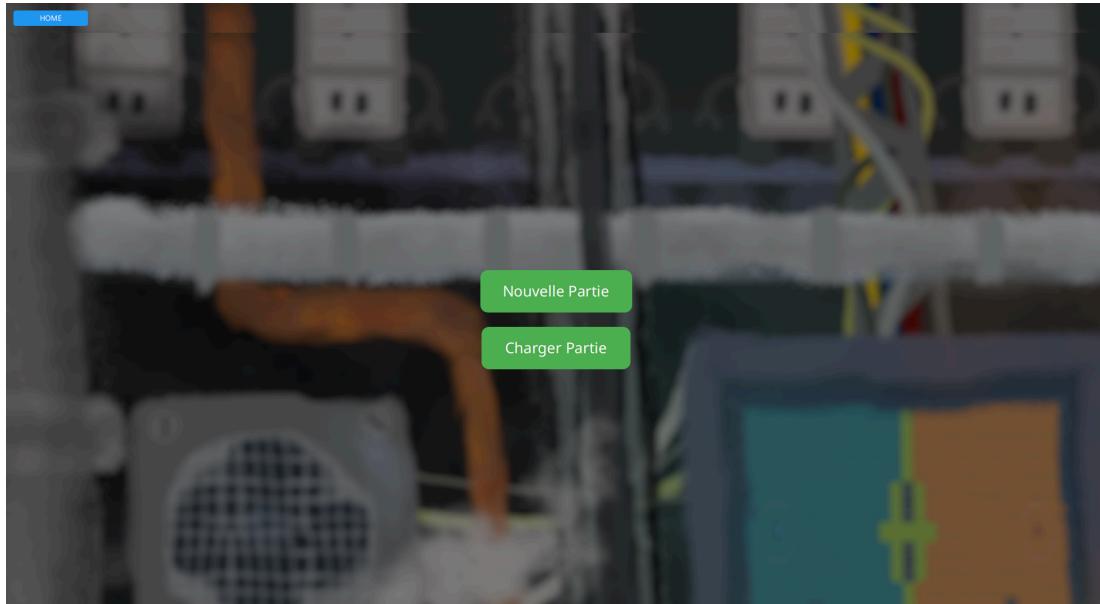
Nous avons présenté l'architecture logiciel dans la partie "Modèles UML utilisés". Voici ci-dessous quelques captures d'écran de notre application pour illustrer les fonctionnalités développées. Certaines captures d'écran ont été coupé pour mieux voir.



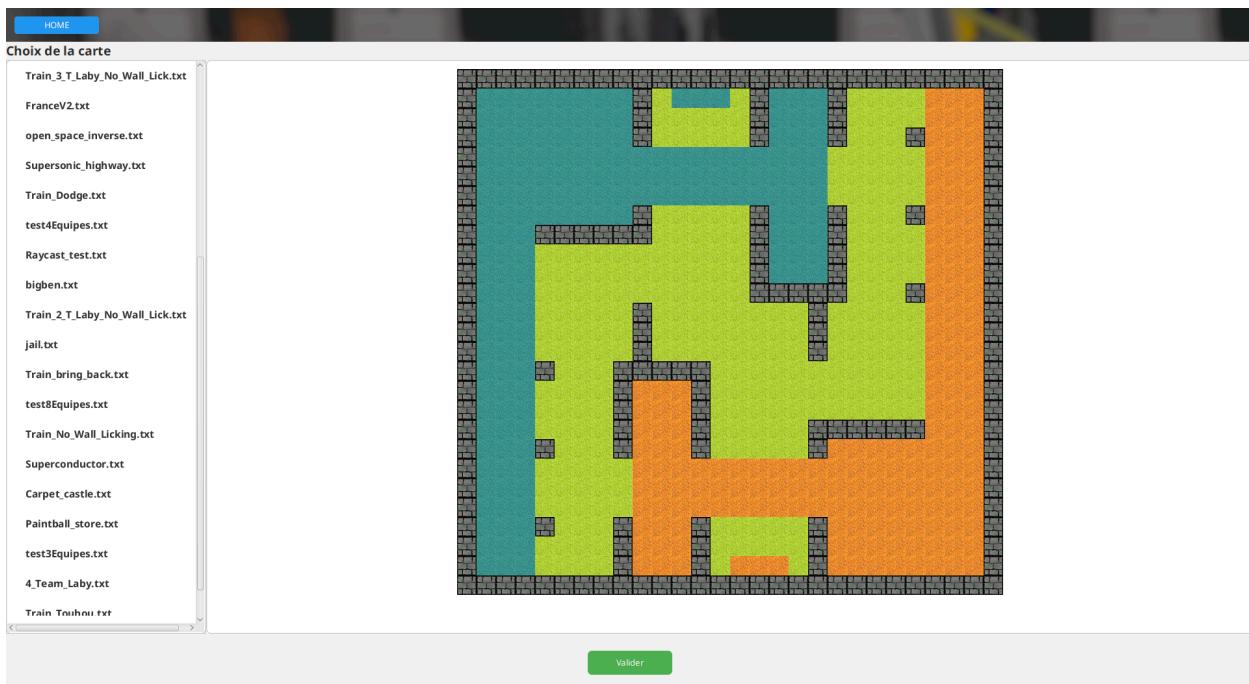
Notre application s'ouvre sur un menu permettant d'accéder aux 3 parties de l'application, la simulation, l'apprentissage, et l'éditeur de carte, un bouton présent au bas de l'interface permet de couper la musique du menu principal.

## La simulation

La première interface du menu de simulation permet de choisir entre créer une nouvelle simulation ou charger une partie précédemment sauvegarder.



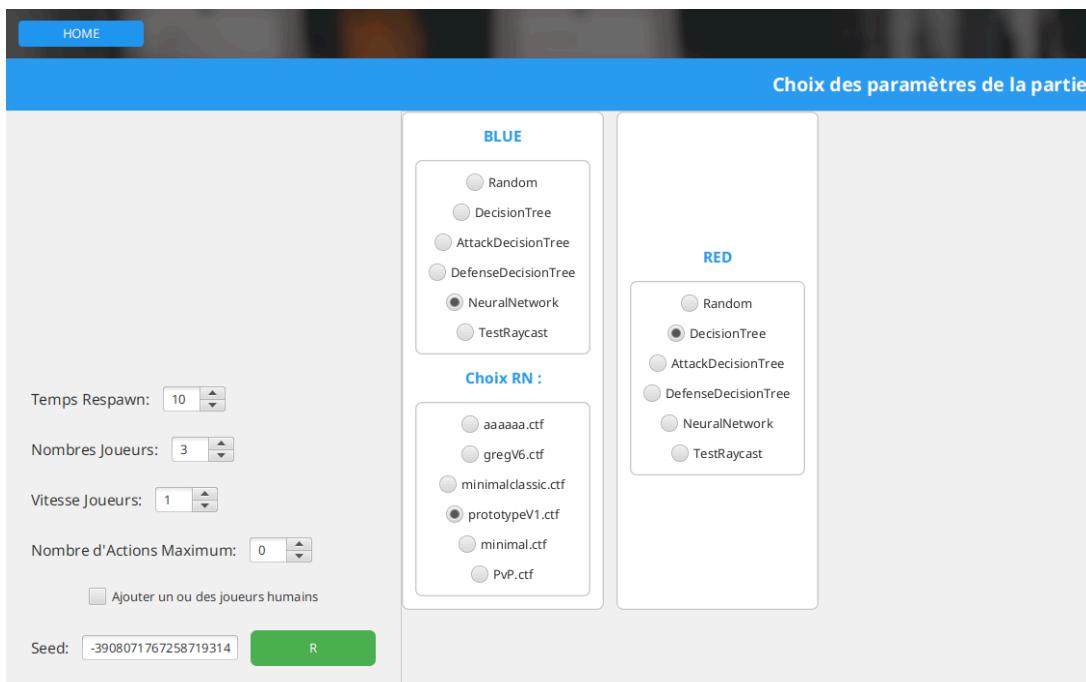
Lors de la création d'une nouvelle partie, l'application demande à l'utilisateur de choisir une carte parmi celles enregistrées en local.



Une fois la carte choisie l'utilisateur a la capacité de configurer la simulation, il peut modifier :

- La vitesse des agents
- Le temps de réapparition
- Le nombre d'agents par équipe
- Le nombre maximal de tour de la partie
- La seed utilisée

En plus de ces options, il doit choisir les modèles qui seront utilisés par les différentes équipes.



L'utilisateur peut également choisir de participer à la simulation en cochant l'option correspondante, ainsi qu'un menu dédié à la configuration des agents contrôlés par des joueurs.

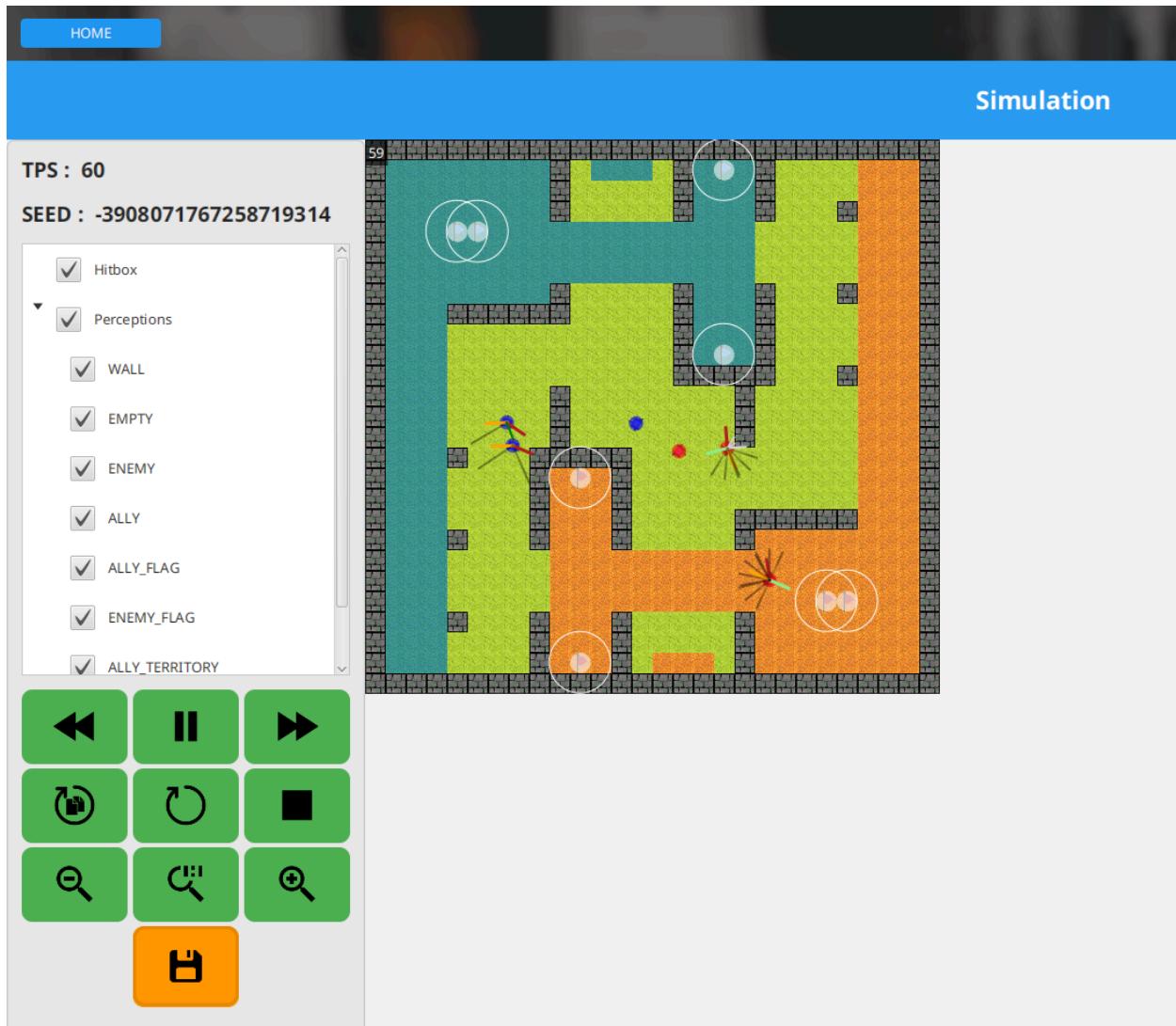
On peut voir qu'il est possible de contrôler chaque agent de chaque équipe, un même joueur pouvant prendre le contrôle de plusieurs agents en même temps, il est également possible d'ajouter des manettes, pour ce faire, une simple connexion bluetooth ou filaire suffit à ajouter la manette dans la liste des contrôles utilisables.



Une fois toutes étapes terminées, la partie peut démarrer, les différentes équipes peuvent enfin s'affronter.

La simulation se présente de cette manière, on y voit donc tous les agents encore en jeu, il est possible d'afficher la zone de collision des agents ainsi que les perceptions des agents à l'aide du menu présent à gauche. Il est d'ailleurs possible de repérer sur la carte les joueurs humains, car ce sont les seuls agents n'ayant aucune perception.

Il est également possible de décélérer, mettre en pause ou accélérer la simulation à n'importe quelle moment, mais aussi de la recommencer avec la même graine ou une graine différente. Le dernier bouton de la seconde rangée permet de retourner au menu, tandis que la l'avant-dernière rangée permet de modifier le zoom de la simulation, le tout dernier bouton sert à enregistrer la partie en local, afin de la rejouer dans les mêmes conditions plus tard.



UNIVERSITÉ  
DE LORRAINE

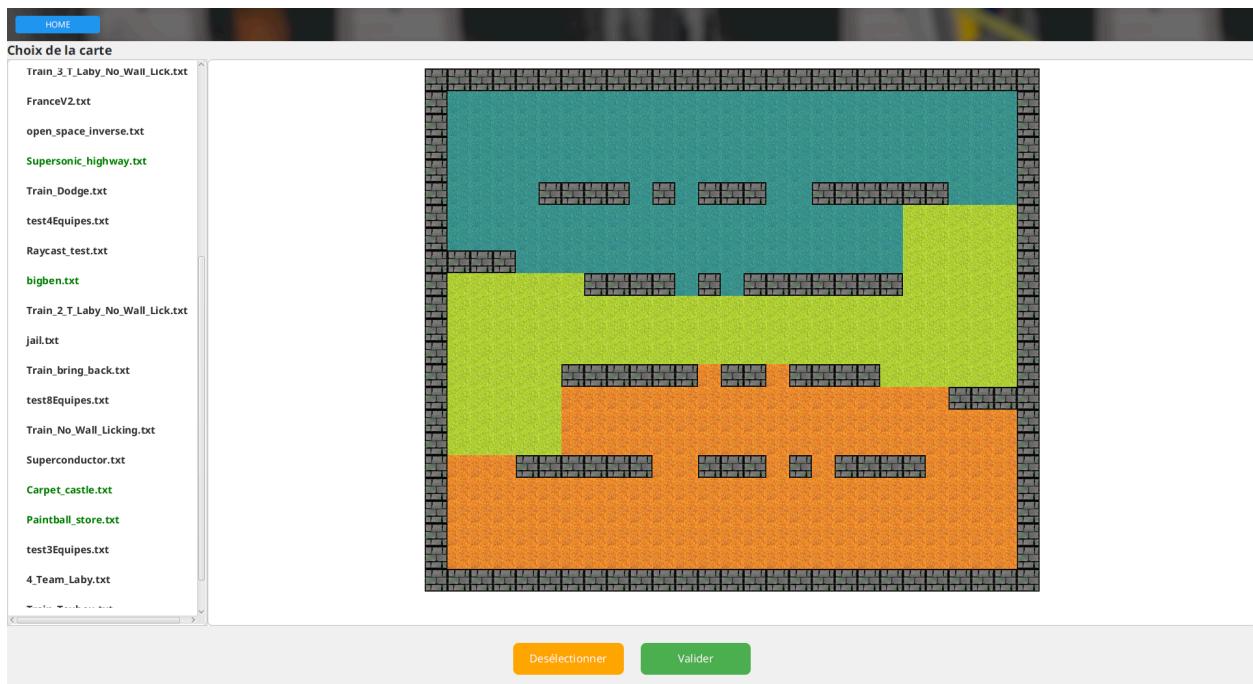


IUT nancy Charlemagne  
Département Informatique

## L'apprentissage

Le menu d'apprentissage s'ouvre sur une sélection de cartes, cette fois-ci l'utilisateur peut choisir plusieurs cartes, les cartes retenues pour l'apprentissage ont leur nom coloré en vert, il n'y a pas de limites explicites concernant le nombre de carte, l'apprentissage peut être fait sur toutes les cartes, comme sur une seule.

Toutefois il est important de noter que plus le nombre de cartes sera grand, plus l'apprentissage sera long.



Une fois les parties sélectionnées, l'utilisateur est invité à configurer les paramètres de l'apprentissage, on retrouve certains paramètres déjà présents lors du lancement d'une simulation, mais il est aussi possible de choisir le nombre de générations, et le nom sous lequel sera enregistré le réseau.

Le choix d'un modèle a été remplacé par le choix de plusieurs modèles, la partie droite de l'interface est dédiée à la configuration du modèle à entraîner, les 4 premières checkbox servent à affecter ou non les boussoles, tandis que la dernière permet de spécifier si le réseau est récurrent ou non, ce concept sera développé plus dans la section dédiée.

Le bouton central permet d'ajouter des perceptions de type rayon, et de choisir le nombre de rayons, leur taille, et l'angle entre le premier et le dernier rayon.

Enfin, la dernière partie de cette interface permet de choisir la fonction d'activation utilisée par le réseau, c'est aussi ici que la quantité de mémoire des réseaux récurrents est configurée. L'utilisateur peut également choisir le nombre et la composition des couches cachées.

HOME

### Choix des paramètres

Entrez ici le nom de votre modèle : demo

Temps Respawn : 10  
Nombres joueurs : 3  
Vitesse Joueurs : 1  
Nombre d'Actions Maximum: 0  
Nombre de générations : 100

Equipe adverse 1

- Random
- DecisionTree
- AttackDecisionTree
- DefenseDecisionTree
- NeuralNetwork
- TestRaycast

Choix des perceptions du bot

- Nearest Enemy Flag Compass
- Nearest Aly Flag Compass
- TerritoryCompass
- WallCompass
- Réseau récurrent

Ajouter un Raycast

Raycast 0 Taille des rayons 1 Nombre de rayons 2 Angle 1 Supprimer

Réseau de neurones  
Fonction d'activation  
 SoftSign  
 Hyperbolic  
 Sigmoid  
Taille de mémoire 2  
Neurones d'entrées : 19  
Nombre de poids : 184  
  
  
Neurones de sorties : 4

Valider

Une fois l'apprentissage lancée, le score du meilleur, pire et ma moyenne des scores de chaque génération est représenté sous la forme d'un graphique, il est également possible de consulter l'évolution du sigma, décrivant la taille de la zone de recherche, et l'indice de conditionnement permettant de décrire la forme de cette même zone.



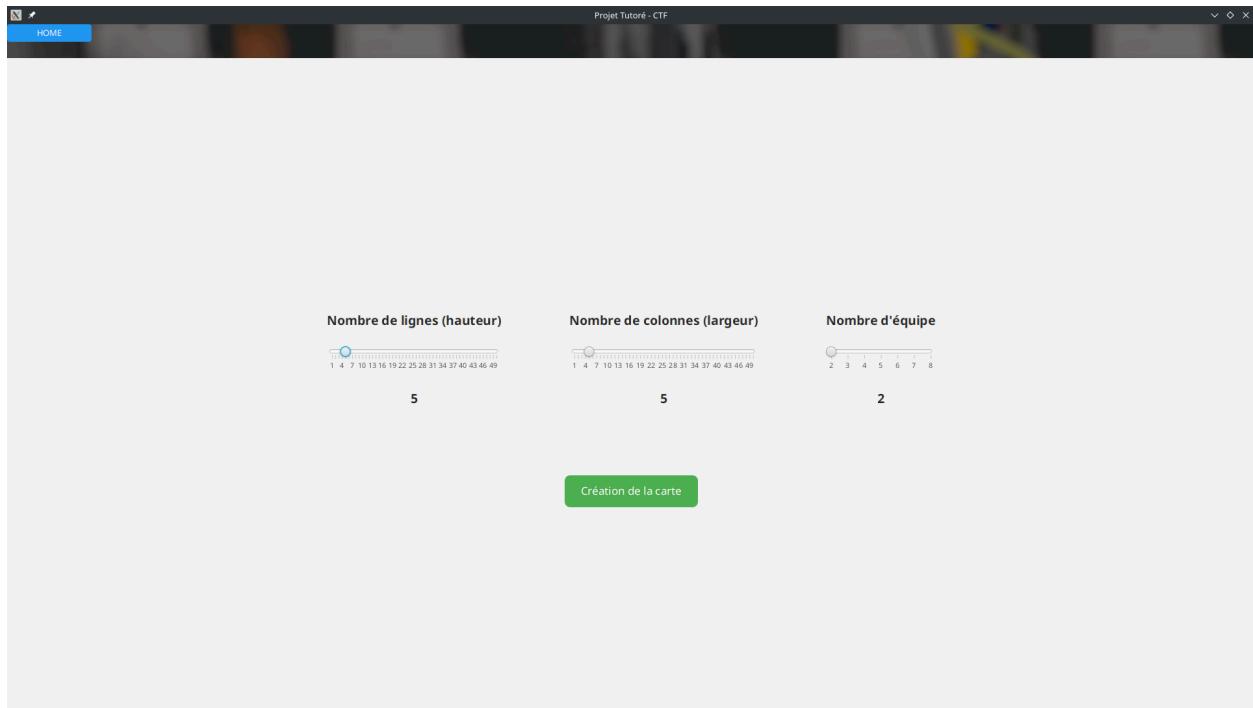
UNIVERSITÉ  
DE LORRAINE



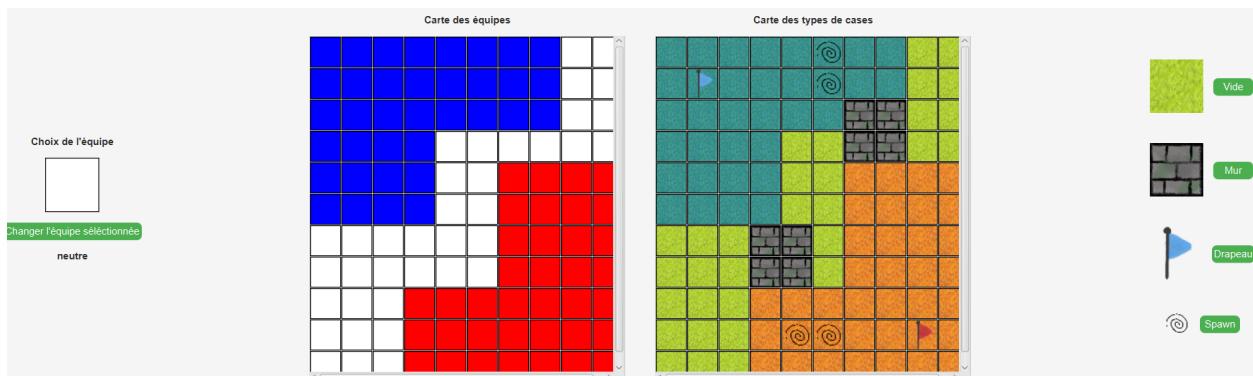
IUT nancy Charlemagne  
Département Informatique

## L'éditeur de carte

Afin de rendre plus accessible la création et la modification de carte, lors de la création d'une carte, la première étape demande d'en définir la taille et le nombre d'équipes qui vont s'affronter dessus.



Une fois ces paramètres définis, l'utilisateur peut éditer les grilles afin de construire une nouvelle carte.



# Intelligence artificielle

Concernant la partie IA, nous avons travaillé sur deux types d'IA (sans compter le modèle aléatoire) : un arbre de décision codé à la main et des réseaux de neurones.

Nous allons tout d'abord détailler la structure d'un agent avant de parler des IA.

## Structure d'un agent

Nous avons structuré les agents comme des robots : ils perçoivent l'environnement via des capteurs et effectuent des actions en fonction. Notre objectif est donc de développer le contrôleur le plus optimal possible pour s'approcher d'un agent jouant parfaitement en fonction des entrées fournies.

### Entrées (perceptions)

Les perceptions d'un agent correspondent à une liste de capteurs qui prend en entrée le monde et retourne les informations que l'agent aura le droit d'utiliser pour se diriger. Nous avons deux types de perceptions : les boussoles et les rayons. Les boussoles donnent la direction et la distance vers des éléments spécifiques (drapeau le plus proche, ennemi le plus proche, mur le plus proche..) tandis que les rayons, comparable à des yeux, permettent à l'agent de voir son environnement proche, les rayons sont décrits selon plusieurs valeurs : le type de l'objet (ou agent) touché et la distance à cet objet.

Entrées dans le réseau par types de perception		
Rayons	Boussole (drapeau)	Boussole (autre)
<ul style="list-style-type: none"><li>• Mur : 0 ou 1</li><li>• Agent : -1; 0 ou 1</li><li>• Drapeau : -1; 0 ou 1</li><li>• Distance : [0; 1]</li></ul>	<ul style="list-style-type: none"><li>• Angle : [-1; 1]</li><li>• Distance : [0; 1]</li><li>• Capturé : 0 ou 1</li></ul>	<ul style="list-style-type: none"><li>• Angle : [-1; 1]</li><li>• Distance : [0; 1]</li></ul>

### Sorties (actions)

A chaque tour de simulation, les agents vont percevoir le monde à l'aide de leurs perceptions, ensuite l'agent va convertir ces perceptions en deux sorties : la vitesse et la direction. Ces sorties sont des valeurs comprises entre -1 et 1 et représentent la proportion de la vitesse (ou rotation) à laquelle l'agent veut avancer (ou tourner). Donc par exemple, si l'agent possède une vitesse de rotation de 10 degrés, et qu'il décide de tourner d'une valeur de -0.5, alors le moteur fera tourner l'agent de 5 degrés vers la gauche.

## Modèle Aléatoire

Le modèle aléatoire est un contrôleur n'utilisant aucune perception pour fonctionner : il conserve une valeur de rotation qui varie à chaque évaluation d'un montant aléatoire. Cela permet à l'agent d'avoir un comportement stable au cours du temps. Cet agent nous permet d'avoir une base de comparaison neutre pour nos futurs modèles.

## Modèle Arbre de décision

Suite à cela, nous avons développé un modèle utilisant un arbre de décision qui est capable de gagner dans des situations simples. Il se déplace vers le drapeau ennemi le plus proche en esquivant les ennemis et les murs sur le chemin, puis rapporte le drapeau dans son territoire. L'arbre présenté ici est un arbre d'attaquant, mais nous avons également conçu un arbre de défenseur qui patrouille autour du drapeau allié le plus proche et se déplace vers les ennemis qu'il perçoit. Initialement, nous assignons des rôles aléatoires aux agents mais cela apportait de variation et d'incertitude dans les parties, et donc dans l'évaluation des agents pendant les entraînements, nous avons donc choisis de rendre leur rôle dynamique : l'agent commence la partie en tant que défenseur, puis quand il détecte et élimine un ennemi, il change de rôle pour aller contre-attaquer.

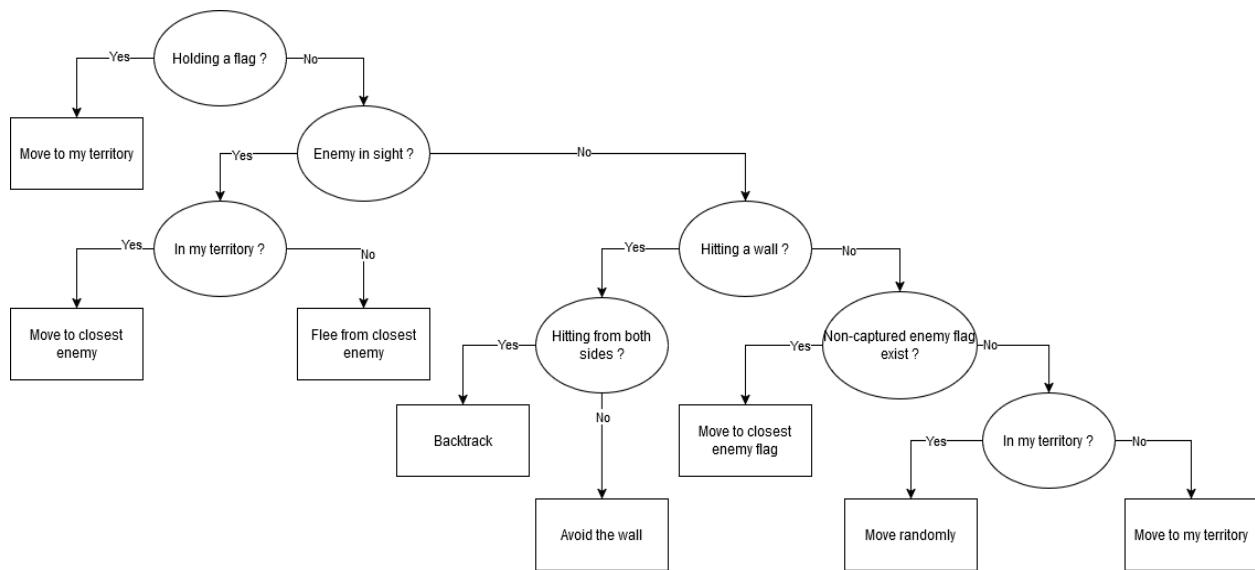


Diagramme représentant l'arbre décision d'un attaquant

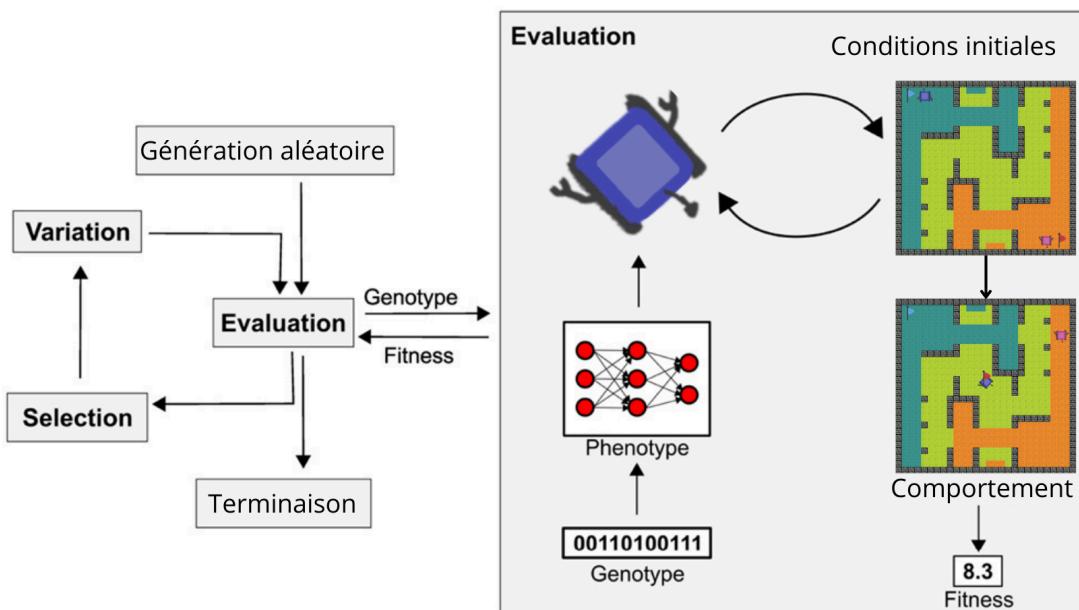
L'arbre de comportement est très performant et gagne rapidement une partie sur des cartes simples. Cependant sur des cartes plus compliquées comme des labyrinthes. Il n'arrive pas à trouver le chemin vers le drapeau car il se dirige tout droit en esquivant les murs.

## Modèle Réseau de neurones

Pour les réseaux de neurones, nous utilisons la bibliothèque “MLP” fournie par nos professeurs pour le module d’initiation à l’IA, nous avons plus tard essayé d’utiliser la bibliothèque “DL4J”, cependant cette bibliothèque est optimisée pour un contexte dans lequel elle serait la seule bibliothèque utilisée, or ce n’est pas le cas dans notre projet puisqu’elle sert seulement d’outil, ce qui a mené à des performances catastrophiques lié au coût de construction d’un réseau DL4J et au fait qu’il utilise de multiples threads (alors que ces derniers sont déjà utilisés par la bibliothèque d’évolution).

### ECJ (Evolutionary Computation for Java)

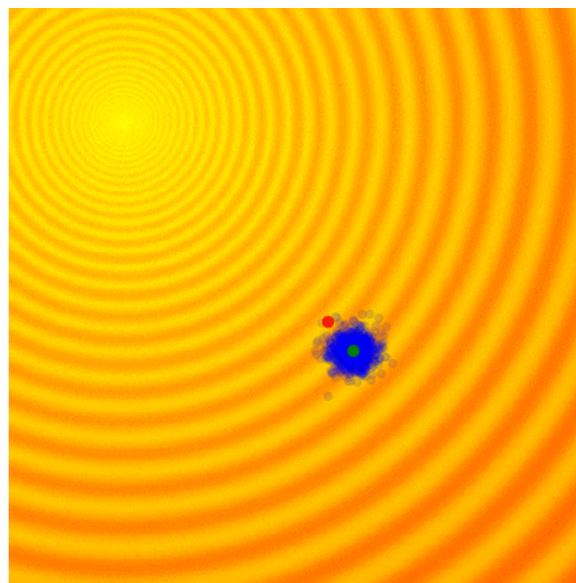
Pour l’apprentissage des poids du réseau, nous utilisons la bibliothèque ECJ, une bibliothèque utilisée dans la recherche pour les algorithmes évolutionnaires. Cette bibliothèque met à disposition de nombreux algorithmes d’évolution, et notre tuteur nous a amené à utiliser les stratégies d’évolution, un modèle qui consiste à créer des populations successives pour optimiser le génotype (dans notre cas le réseau de neurone).



Plus particulièrement, nous utilisons l'algorithme CMA-ES (“Covariance matrix adaptation evolution strategy”), qui est un algorithme inspiré de la descente de gradient : CMA-ES utilise un seul individu et va générer  $\lambda$  enfants répartis autour de l'individu selon la matrice de covariance. CMA-ES va alors sélectionner les  $\mu$  meilleurs enfants et mettre à jour l'individu en interpolant les enfants sélectionnés.

Voici un schéma du fonctionnement de CMA-ES :

- Le point vert représente la solution actuelle (l'individus de la population)
- Les points bleus représentent les enfants de la solution
- Le point rouge représente le meilleur enfant d'une génération



## Fonction d'évaluation

La fonction d'évaluation ("fitness") est utilisée pour évaluer les performances d'un agent lors du processus d'évolution. L'objectif de l'algorithme d'évolution va donc être de trouver le réseau de neurones minimisant la fitness. La fonction d'évaluation que nous utilisons est la suivante :

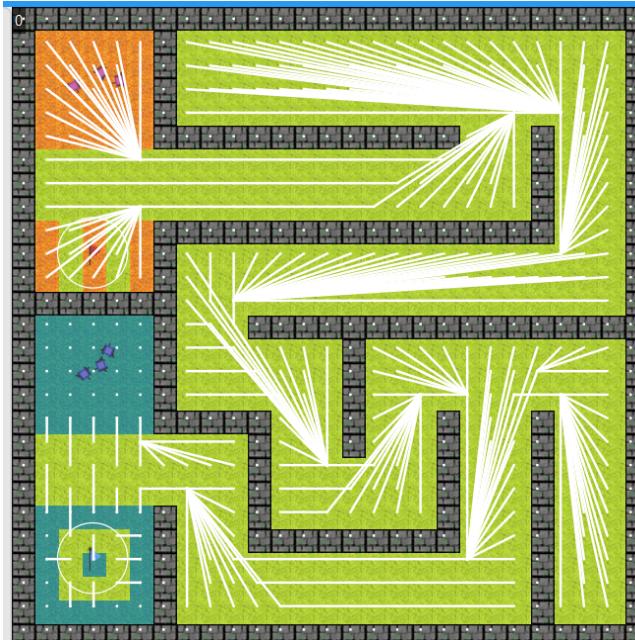
$$\text{Fitness} = \left( \sum_{i=1}^{\text{nbDrapeaux}} D_{\text{agent } i} + D_{\text{territoire } i} \right) + \left( 1 - \frac{\text{durée finale}}{\text{durée max}} \right)$$

où :

- $D_{\text{agent } i}$  est la distance minimale à laquelle les agents de l'équipe à réussi à s'approcher du drapeau i
- $D_{\text{territoire } i}$  est la distance minimale à laquelle l'équipe à réussi à approcher le drapeau i de son territoire
- durée max représente la durée maximale d'une partie
- durée final représente la durée restante à la fin de la partie : une partie se terminant par une capture de tous les drapeaux aura une durée finale > 0, alors qu'une partie se terminant par le temps aura une durée finale = 0

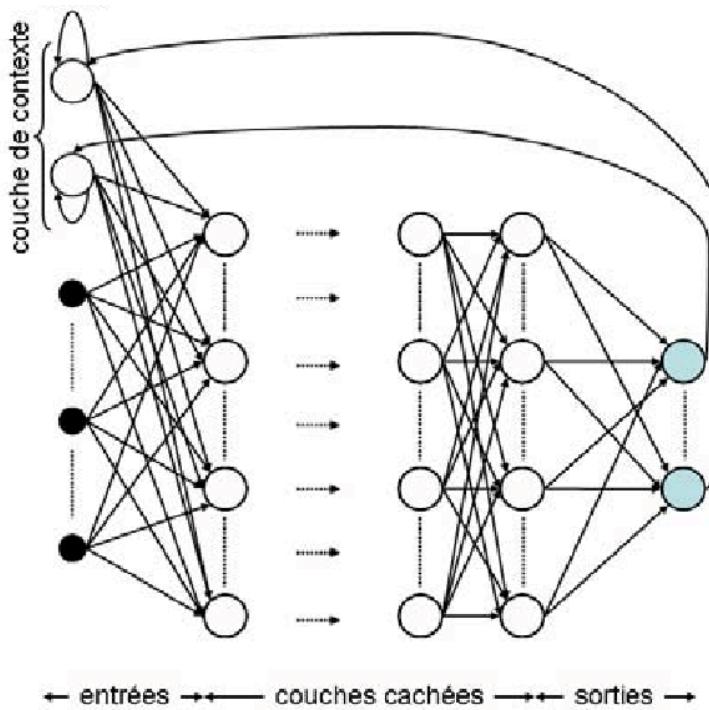
Initialement, nous intégrions le score des équipes adverses dans la fonction d'évaluation, l'objectif était de pousser les agents à attaquer et défendre en même temps. Cependant, comme il est plus simple d'apprendre à tourner autour de son drapeau qu'à attaquer, les agents ne faisaient que maximiser leur note de défense sans jamais essayer d'attaquer. C'est pourquoi nous avons retiré la note ennemie de la fonction pour ne conserver que la distance alliée.

Les distances pour la fonction d'évaluation sont calculées en utilisant une table des distances pré-calculée par l'algorithme dijkstra pour une meilleur précision dans les labyrinthes :



## Réseaux de neurones récurrent

Au cours de la 6ème itération nous avons implémenté des réseaux de neurones récurrents : des réseaux de neurones ayant pour entrée les sorties de l'inférence précédente. Ainsi dans la forme classique, le réseau possède 2 entrées supplémentaires, la vitesse et rotation au tour précédent; cependant nous avons ajouter un paramètre de "mémoire", qui ajoute X entrées/sorties au réseau. Ces entrées et sorties sont utilisées uniquement par le réseau et lui servent à encoder des informations qui seront utilisées aux tours suivants, dans l'optique de donner une mémoire au réseau. Cependant par manque de temps nous n'avons pu faire d'étude comparant l'efficacité d'un réseau avec et sans cette mémoire.



## Notes sur l'apprentissage

Au cours du projet, nous n'avons cessé d'améliorer l'environnement d'apprentissage pour maximiser la qualité de jeu de nos agents, voici les axes non mentionnés plus tôt sur lequel nous avons travaillé :

Nous avons commencé par faire apprendre nos agents sur une seule carte, contre un seul type d'agent et sur une seule partie, or cela n'a pas apporté suffisamment de situations variées au modèle et par conséquent les performances de l'agent n'étaient pas très satisfaisantes. Nous avons donc amélioré cela pour que l'agent joue X parties contre tous les modèles sélectionnés dans l'interface (de tout type : aléatoire, arbre de décision ou réseau de neurone) sur toutes les cartes sélectionnées. Ainsi la fitness devient la moyenne des notes obtenues sur toutes les parties jouées, ce qui oblige le modèle à généraliser.

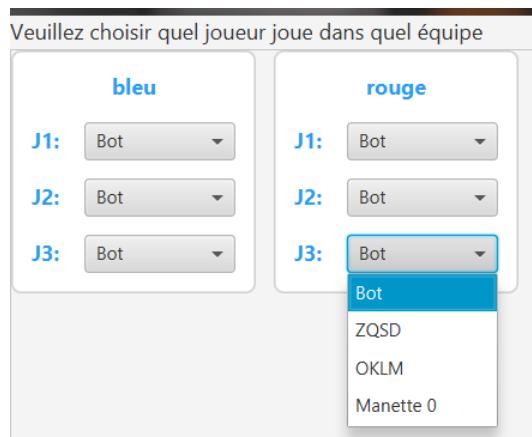
Grâce à cela, nous pouvions entraîner des agents sur 4 cartes, contre 3 adversaires en jouant 10 parties, soit 120 parties par agent à testé. Ce nombre important de parties à considérablement augmenté le temps d'apprentissage (contrairement à la taille du réseau à apprendre qui elle n'influe que sur le nombre de générations nécessaire pour apprendre le réseau), nous faisant passer à plusieurs jours d'apprentissage.

Jouer plus de parties améliore la qualité de jeu des agents, cependant ce n'était encore pas satisfaisant : malgré des réseaux avec environ 400 poids, ils ne faisaient que longer les murs pour atteindre les drapeaux plutôt que d'utiliser les boussoles. Nous avons donc réduit le nombre d'entrées du réseau en retirant des informations superflues (l'angle à laquelle les rayons sont tiré : l'agent peut apprendre ces informations par coeur, et nous n'avions donc pas à donner cette information au réseau) et réduit la limite de temps par partie (nous sommes passé de parties durant 30000 tours à seulement 8000). Ces modifications nous ont permis de lancer des apprentissages avec des réseaux plus gros (plus de couches cachées et plus grosses) et sur beaucoup plus de cartes : notre réseau le plus performant, "GregV12", fonctionne avec 20 entrées (3 rayons et 2 boussoles) puis 2 couches cachées de 15 et 6 neurones, soit 402 paramètres et a appris contre 3 modèles sur 9 cartes, en faisant 7 parties à chaque fois soit 189 parties par modèle à évaluer. Ce réseau contenant plus de neurones cachés et ayant joué dans plus de situations différentes a atteint un bien meilleur niveau de jeu en seulement 3 heures d'apprentissage. L'algorithme a convergé en environ 400 générations.

Nous n'avons pas eu le temps de lancer un apprentissage sur un réseau encore plus grand, sur plus de cartes et contre des agents à réseau de neurones mais nous pensons que les résultats auraient été encore meilleurs.

## Modèle : Humain

En plus de tous ces modèles agents autonomes nous en avons développé un modèle contrôlé par un humain. Ce modèle utilise les mêmes interfaces qu'un modèle d'ia, le joueur utilise donc un clavier ou une manette pour contrôler les actions de l'agent. L'image ci-dessous représente le menu permettant de sélectionner les contrôles utilisés par chaque joueur.



## Fonctionnalités que l'on aurait aimé ajouter

Nous avions prévu dès l'étude préalable un ensemble de fonctionnalités que nous voulions ajouter après avoir finis le projet de base. Certaines de ces fonctionnalités ont été ajoutées comme l'éditeur de carte et la possibilité de jouer soi-même contre des robots ou contre d'autres humains.

Voici la liste des fonctionnalités que nous n'avons pas eu le temps de développer.

Application :

- Application responsive (adaptative à l'écran utilisé), que l'on a essayé d'implémenter sans succès.
- Écran de benchmark pour visualiser la performance des agents sur les cartes (sur un grand nombre de parties, combien de fois gagne un agent particulier).
- Visualisation d'un réseau de neurone en direct (voir prototype ci-dessous)

IA :

- Communication entre robots (stratégie de coopération pour récupérer un drapeau ou attaquer des adversaires).
- Coévolution, faire évoluer plusieurs réseaux de neurones concurrents afin d'obtenir une "course à l'armement" et faire en sorte que les avancés "intelligentes" d'un réseau inspire le deuxième pour qu'il devienne meilleur lui aussi et ainsi de suite.

Jeu :

- Ajout d'objets récupérables (bonus de vitesse, bonus d'immunité, téléporteur, etc...).
- Vue isométrique (initialement prévue mais abandonné lorsque l'on a choisi de passer d'un monde discret à un monde continu)

## Tests de validation

Dès la première itération, nous avons mis en place plusieurs classes de tests, notamment sur le comportement de l'agent aléatoire et sur la validité des cartes. Par la suite, notamment à l'itération 6, nous avons également ajouté des tests spécifiques au chargement et au déchargement des modèles de réseaux de neurones.

Cependant, au fil du développement, nous n'avons pas généralisé la couverture de tests automatisés à l'ensemble de l'application. L'une des raisons principales tient au fait que notre application est largement axée sur l'interface graphique avec JavaFX, rendant l'écriture de tests automatisés complexes et peu accessibles pour nous à ce stade. Cela nous a poussés à privilégier les tests manuels durant les phases de développement, en vérifiant les comportements directement via l'interface.

Cela dit, avec du recul, nous reconnaissons que davantage de tests automatisés, même sur les modules les plus logiques ou critiques (comme les modèles d'agents, la logique du moteur ou certaines fonctions utilitaires), auraient pu améliorer la robustesse globale de notre projet. En effet, nous avons parfois rencontré des problèmes de régressions ou de casse de code suite à des merges ou des ajouts de fonctionnalités. Une couverture de tests plus étendue aurait sans doute permis de limiter ces erreurs plus efficacement.

Nous retenons donc pour nos futurs projets l'importance d'un meilleur équilibre entre tests manuels et tests automatisés, même dans le cadre d'applications à forte composante visuelle.

## Difficultés rencontrées

### ECJ

Nous avons eu des difficultés lors de l'installation d'ECJ (pas de tuto simple), lors de l'apprentissage de son fonctionnement (pas d'informations claires, une doc vieille et pas à jour). Mais une fois installé et compris, nous avons pu nous en servir pour l'apprentissage de réseaux.

La première étape a été de créer des réseaux très petit ne disposant que de quelques entrées, les premiers modèles étaient donc composé d'une perception vers la boussole du drapeau ennemis le plus proche, ce premier test nous a permis de voir que ECJ était capable de faire apprendre un réseau pour le faire suivre une boussole.

Nous avons ensuite entrepris de faire apprendre à des réseaux plus grands et plus complexes. Ces premières expérimentations nous ont permis de découvrir que nous utilisions la mauvaise fonction de transfert pour nos réseaux de neurones.

En effet, jusque là nous utilisions la Sigmoïde, cependant cette fonction ne sort des résultats qu'entre 0 et 1, or les agents peuvent fournir des valeurs entre -1 et 1, nous avons pu entraîner

un petit réseau qui avait pour perception uniquement 2 rayons de taille 1, capable de reconnaître un mur, de le contourner par la droite, mais incapable de le contourner par la gauche.

Après avoir changé la fonction de transfert pour n'utiliser que la tangente hyperbolique (ou tanh) nous avons pu constater de meilleurs résultats.

### Bugs et “Edge-Case”

Une grande partie des problèmes rencontrés viennent de cas très précis que l'on ne remarque qu'en itérant un grand nombre de fois des parties aléatoires. Tel qu'un agent dans un cas très précis qui peut se retrouver entre 4 murs ou des agents qui envoient des drapeaux hors zone. Cela nous pousse à raffiner de plus en plus les fonctionnalités déjà existantes et d'envisager aux mêmes moments des cas qui n'auraient pas été pris en compte plus tôt.

En fin de projet, il nous reste donc quelques problèmes non résolus mais qui ne sont pas forcément problématiques pour une utilisation simple de l'application.

# Conclusion

Ce projet tutoré représente un véritable aboutissement des compétences que nous avons acquises tout au long de notre formation. Il nous a permis de mettre en pratique des notions complexes en conception logicielle, programmation orientée objet, développement d'interfaces graphiques, et intelligence artificielle, tout en relevant des défis techniques ambitieux et stimulants.

Le développement d'une simulation complète du jeu *Capture The Flag*, intégrant des agents autonomes et un moteur de jeu performant, nous a offert une opportunité unique d'explorer en profondeur des domaines variés, comme la refonte d'architecture logicielle (MVC), la modélisation de comportements d'agents, ou encore l'apprentissage automatique par évolution génétique (CMA-ES).

Au-delà des aspects techniques, ce projet nous a également confrontés à la réalité du travail en équipe sur un projet long : planification rigoureuse, gestion des priorités, coordination, prise de décisions collectives, documentation continue, et adaptation aux imprévus. Nous avons veillé à répartir au mieux les tâches, à tirer parti des forces de chacun, et à maintenir une dynamique de groupe constructive et efficace tout au long du semestre.

Si toutes les fonctionnalités envisagées au départ n'ont pas pu être finalisées, les objectifs principaux ont été atteints. L'application que nous avons développée est fonctionnelle, riche, modulable et constitue une base solide pour d'éventuelles extensions ou expérimentations futures dans le domaine de l'IA.

Nous sortons de ce projet avec une vision plus concrète et professionnelle du développement logiciel, et une meilleure compréhension des défis liés à la mise en œuvre de systèmes intelligents. Ce travail représente pour nous une réussite collective, et une belle démonstration de nos capacités techniques, méthodologiques et humaines en vue de notre entrée dans le monde professionnel ou la poursuite d'études.

Cependant, avec le recul, nous sommes conscients que certains aspects, notamment la préparation de nos présentations orales, pourraient encore être perfectionnés. C'est une piste d'amélioration que nous retenons pour l'avenir : savoir valoriser nos compétences à l'oral avec autant de rigueur que dans le développement technique. Cette remise en question constructive nous accompagnera dans nos prochains projets.

# Annexe

## Lancement du projet

Pour lancer le projet, il suffit de **cloner le dépôt GitHub** à l'adresse suivante :  
<https://github.com/hirtz-gregoire/RA-IL2-ProjetTutore-CTF>

Une fois le projet ouvert dans votre **IDE** (nous recommandons **IntelliJ**), il vous suffit de **charger la configuration Maven** via le fichier `pom.xml`. L'ensemble des dépendances nécessaires sera automatiquement téléchargé et configuré.

Le projet devrait alors être prêt à être compilé et exécuté. Pour lancer l'application, il suffit d'exécuter le fichier `Main.java` situé dans le dossier `src`.