Rapport Aistrat
Projet Artishow 2024

Degieux Erwan, Neltner Pierre, Besson Nicolas, Weber Tom, Paviot Martin, Fraizier Jules

Encadrant : Jachiet Louis

I. Introduction

Notre objectif pour ce projet était de produire une IA la plus performante possible pour un jeu de stratégie donné : Chartishow.

La contrainte principale était l'interdiction pour notre groupe d'utiliser du machine learning pour nos IA, et de nous en tenir aux implémentations algorithmiques.

Il nous était également interdit de "tricher" avec notre IA. Aucune ressources additionnelles ni d'omniscience pour nos algorithmes.

L'objectif était ainsi de réaliser une IA qui pouvait se battre à la loyale avec un joueur humain et les autres IA. La seule chose pouvant donc nous distinguer étant la qualité de la logique.

Ce rapport commence par une présentation de l'état de l'art des IA de stratégies qui existent, puis nous présentons les 3 principales IA auxquelles nous sommes arrivés, qui abordent chacune l'IA de stratégie différemment, et qui possèdent chacune des forces et des faiblesses.

Table des matières

II. Etat de l'art des intelligences artificielles pour jeux de stratégies

1. Système global/interdépendant

Le fonctionnement le plus intuitif est celui où tous les éléments sont mélangés, interdépendants et se répondent les uns aux autres.

Ce serait le système le plus efficace et fonctionnel dans un monde idéal où la puissance de calcul et la capacité de programmation serait infinie. Cela semble cependant complexe à implémenter tel quel dans ce contexte au vu du nombre déjà assez grands de systèmes que l'on peut déjà considérer dans un jeu aux fonctionnalités simplistes.

2. Système par module (Jeux 4X comme Civilization)

Les tâches de l'IA peuvent être réparties à différents niveaux. Chaque niveau décide d'actions et d'objectif qu'il va soit exécuter directement, soit déléguer au niveau inférieur. Les niveaux sont les suivants :

- **Stratégique** (gestion des ressources, des objectifs de victoire...)
- **Opérationnel** (diplomatie, économie...)
- **Tactique** (géographie du terrain, analyse des ennemis..) *
- **Individuel** (pathfinding des unités, combats...)

Dans le cas de notre jeu, **le niveau opérationnel n'existe pas.** On peut cependant s'inspirer de cette répartition des prises de décisions :

- Le niveau stratégique décide de quoi défendre en priorité et où, ainsi que la priorité stratégique (attaque, défense, économie privilégié).
- Le niveau tactique décide de où placer les unités militaires, leurs assignations aux zones à défendre ou attaquer, ainsi que l'assignation des péons aux piles d'or.
- Le niveau individuel ne servirai qu'au pathfinding.

On peut laisser les unités faire leur tâche de manière **autonome**, sans les superviser à chaque tour. Pour éviter qu'elles meurent seule, il leur suffit de rappeler leur existence lorsqu'un ennemi se rapproche.

3. Systèmes de décisions en pratique pour les jeux

3.a. Notion d'heuristique

La majorité des algorithmes de décision ont besoin de trouver, à partir d'un état de la partie, **une valeur supposément objective indiquant l'intérêt de la configuration actuelle**.

L'étude de ces évaluation s'appelle **Heuristique**. Dans le cadre des jeux, l'heuristique dépend souvent de divers **critères** liés :

- au matériel (jetons, ...) ;
- à la mobilité (nombre de coups possibles) ;
- à la position (des pièces, ...).

Que l'on score **individuellement** et que l'on pondère par des poids variables au cours de la partie.

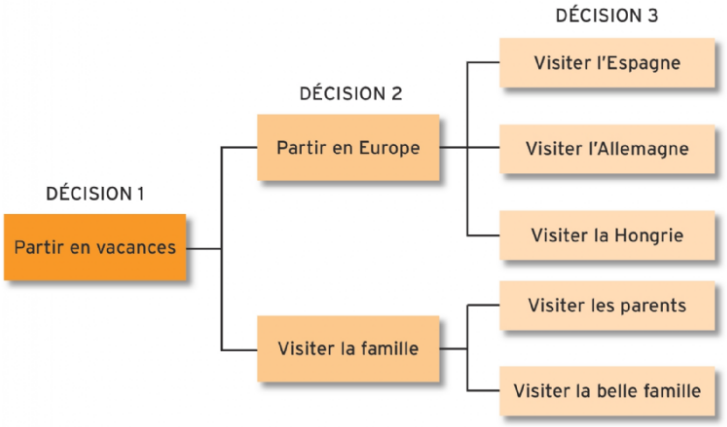

FIGURE 1 – Ilustration arbres de décision [3]

Score final = pmateriel × cmateriel + pmobilite × cmobilite + pposition × cposition

3.b. Les arbres de décision

Les arbres de décision fonctionnent de manière binaire par rapport aux états d'une partie. On commence à partir d'une racine puis des décisions seront prises successivement en fonction d'évaluations de l'état de la partie.

L'action finale sera celle prise lorsque l'algorithme tombera sur une feuille, c'est à dire un élément de l'arbre sans fils.

C'est un système de décision assez simple qui permet d'implémenter facilement des décisions à u.,n niveau individuel des unités et de définir facilement des stratégies (exemple : priorisation de la récupération de ressources plutôt que l'exploration pour les péons).

3.c. Min-Max (Puissance 4, Jeu de dame...)

Le Min-max est un algorithme ayant vocation à **explorer entièrement l'arbre des combinaisons possibles sur un certain nombre de coups. La décision sur quelle chemin est à privilégier se fait ensuite sous la logique suivante**

- On considère que comme nous, l'adversaire va **toujours effectuer le meilleur coup**.
- On cherche donc à l'entraîner sur le chemin qui est **le plus profitable pour nous malgré des bons coups de sa part**.

À chaque coup de l'adversaire, on attribue un **score** à l'état de la partie. Le noeud précédent, l'un de nos coups possibles, obtient alors pour score le **minimum** des résultats pour les coups de l'adversaire suivant, illustrant ainsi le principe de **décision optimale** pour l'adversaire.

A chaque noeud, le choix d'action à notre tour sera ainsi celui ayant la **valeur la plus élevée**, c'est donc la décision qui nous mène vers **la moins pire des configurations** une fois que l'adversaire a joué.

L'algorithme est cependant assez lourd et exponentiel en la profondeur de coups analysés. Il existe néanmoins des méthodes d'optimisations comme la méthode d'**élagage alpha-bêta** qui peuvent réduire le temps d'exécution en **éliminant rapidement des branches** dès qu'on obtient un score non significatif (exemple si on a une branche à 5 de score, toute autre branche dont on sait que le score maximal sera strictement inférieur à 5 pourra être coupée directement).

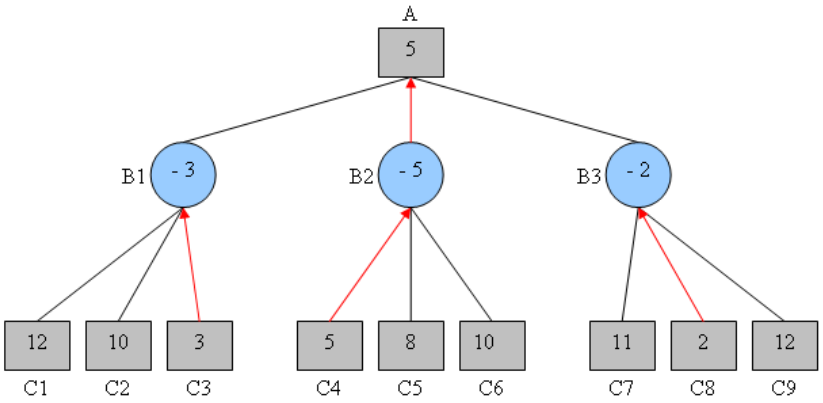

FIGURE 2 – Illustration du minimax [4]

L'algorithme est programmé de telle sorte à ce qu'on reçoive toujours la solution idéale pour le système de score que l'on utilise à la profondeur donnée, mais ce seulement si l'information offerte à l'algorithme est totale. En cas d'informations manquantes, la chose se complexifie.

Si l'on accepte de ne pas tout connaître sur la partie, l'usage de Min-max peut être utile dans des scénarios plus complexes en concentrant son usage sur des situations de petite taille (exemple des combats rapprochés dans la série des Civilisation).

3.d. Recherche arborescente Monte-Carlo

Monte-Carlo a une philosophie similaire à l'algorithme Min-Max, c'est à dire l'exploration des arbres des possibilités de jeu, mais **sans être exhaustif.**

- À partir d'une racine, on explore des enfants (quitte à les créer) **jusqu'à tomber sur une feuille** (un élément sans enfant).
- On cherche lors de cette phase à **maximiser la chance de résultat tout en donnant une chance aux branches pour l'instant pas idéales** mais prometteuses (Sélection).
- On crée ensuite un nouvel enfant pour lequel on joue **une partie aléatoire jusqu'à obtenir une position finale** pour laquelle on note si on a une réussite (Expansion et Simulation, on a un échec ici d'où 0 victoires / 1 partie jouée).
- On fait ensuite **remonter l'information jusqu'à la racine** et on recommence la phase de sélection.

La complexité et la finesse d'usage de l'algorithme se trouve surtout dans le **paramétrage de la phase de Sélection** entre le choix des positions maximisant la probabilité de victoires et celles ayant du potentiel.

C'est un algorithme moins gourmand que Min-Max mais avec des décisions qui ne peuvent pas être aussi bonnes que lui. **Il souffre notamment beaucoup de la présence d'entropie dans les actions possibles des joueurs et donc compliqué à utiliser dans une situation où les possibilités sont très nombreuses.**

3.e. Système de classeur (CPU Super Smash Brothers, priorité de construction en 4X...)

Un systeme de classeur est un ensemble de règles (qui sont en fait des actions possibles) constituées de :

- **Une condition** qui représente un état global ou local du jeu dans laquelle l'action associée est possible.

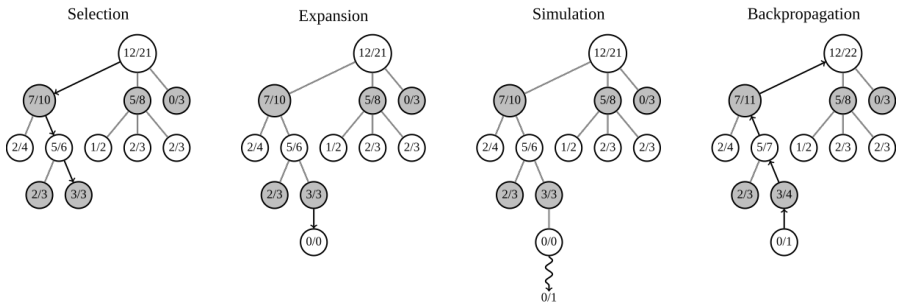

FIGURE 3 – Illustration de Monte-Carlo [1]

- **Une action** qui, dans le cas d'un ouvrier de notre jeu, peut être de se déplacer à droite, à gauche, en haut, en bas, construire un château...
- **Un poids** qui pondère la probabilité de choisir cette action parlis les autres actions valides. L'usage d'un systeme de classeur n'est donc pas déterministe.

La création du système de classeur peut se faire à la main ou à partir d'un **algorithme génétique sur les règles** (pour pouvoir implémenter un système plus fin voire évolutif via une boucle de retour).

Les conditions peuvent porter sur des variables globales représentatnt l'état de la partie et la grille localement autour de l'élément dont on veut déterminer l'action. On a essentiellement un arbre de décision avec un ajout d'aléatoire sur les différentes branches légales.

3.f. Système expert (Conseil pour Echec, Go...)

Ce système de décisions utilise **un ensemble de faits et de règles** (on peut faire une analogie avec la logique, les fait étant ici des valuation d'un ensemble de variable et les règles comme des règles de logique). On peut utiliser comme support la logique du premier ordre par exemple.

On utilise ensuite **des moteurs d'inférence** pour résoudre le problème et prendre des décisions. Ce système revient à **traduire l'état de la partie en formule logique** ayant déjà une évaluation partielle et à la résoudre.

Ce système nécessite une **connaissance très intime du jeux et des conséquences de chaque action** sous risque d'être potentiellement très mauvais.

3.g. Les Heatmaps (Civilization...)

Les Heatmaps consistent à générer des **"points chauds" d'intérêt** sur la carte d'un jeu en fonction d'une heuristique à **un niveau local ou global**.

Ces points chauds peuvent être utilisés comme **point itermédiaire d'Heuristique ou comme outil direct de prise de décision** : par exemple pour aiguiller des unités directement vers ces points chauds en priorité.

Par exemple l'illustration ci-dessous montre qu'à chaque case connue de la carte est attribué un code par rapport à **l'intérêt stratégique** de chaque position qui orientent ainsi le placement de chacune des unités et des colonies.

La reconnaisance de ces points chauds avec une Heuristique correcte peut permettre **d'orienter les unités de manière très qualitative sur la carte** en reconnaissant automatiquement tous les points d'intérêt.

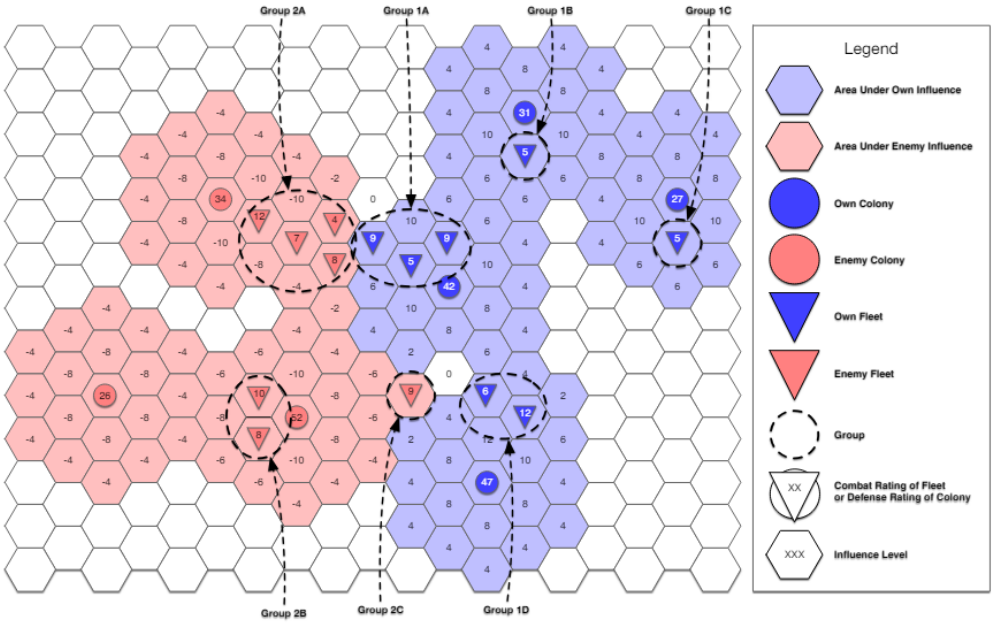

Figure 27. A possible grouping of units for analyzing strategic dispositions.

FIGURE 4 – Illustration issue de la thèse de Toni Lääveri Integrating AI for Turn-Based 4X Strategy Game [2]

3.h. La Triche

La triche est quelque chose qui est assez commun pour les IAs dans de nombreux jeux de toute catégorie : Civilization (4X), Starcraft (RTS), Super Smash Bros. (Combat), Elden Ring (Action-RPG)...

Celle-ci s'illustre de différentes manières :

- Ressources additionnelles au départ ou en continu dans le cas des 4X.
- Visibilité complète de la carte dans le cas de jeux de stratégies type Moba et des 4X.
- Réaction directe vis-à-vis des bouttons pressés par les joueurs dans le cas des ennemis de jeux d'action et des CPU de jeux de combat.
- etc...

Sans parler des purs changements statistiques offerts aux ennemis dans le cas de l'augmentation de la difficulté pour beaucoup de jeux, notamment des RPGs (Merciless dans les jeux Shin Megami Tensei/Persona par exemple)

Celle-ci est intégrée de manière très intime avec le fonctionnement interne des IA intégrées au jeu avec, dans le cas des IA à difficulté variable, une variation intiale sur le niveau d'intelligence de l'IA rapidement supplantée par la quantité de triche que l'IA s'autorise (notamment dans le cas de 4X comme Civilization). Dans les cas où la triche n'est pas "variable" comme des ressources, c'est le temps de réaction et le poids des informations "illégales" augmentant de la prise de décisions qui sont favorisés.

Notre IA ne pouvant pas tricher, nous n'aurons pas accès à ces béquilles pour pallier à une intelligence imparfaite et ses erreurs.

III. Nos implémentations

1. L'API client-serveur

1.a. Principe de l'API

Le premier travail effectué a été d'implémenter un système de communication avec le serveur que tout le reste du code pourrait utiliser dans toutes les versions en "boîte noire".

Philosophie des actions essentielles :

- Au début de la partie, on utilise une fonction `init` pour échanger les tokens de connexion avec le serveur.
- Au début de chaque tour, on envoie beaucoup de requêtes serveur avec `get\_data` pour savoir si c'est à nous de joueur. Si c'est le cas, on stocke en global toutes les données (en récupère la 'carte' du jeu). Tous les autres appels à l'API utilise cette map et donc n'appellent pas le serveur.
- À la fin du tour, on informe le serveur avec `end\_turn` qu'il peut dire à l'autre de joueur que c'est à son tour de jouer.

1.b. Fonctions de l'API

Cette API possède de nombreuses fonctions, dont des fonctions qui permettent de récupérer la carte, ou le contraire, de récupérer les positions de tel et tel type d'unité. Elle donne l'accès notamment à des informations sur nous ou l'ennemi (principalement sa position) et les cases qu'on ne voit pas (appelé `fog` dans le code). On peut également récuperer des données tels que la taille de la carte (calculée une seule fois au début de la partie), ou le joueur qui est en train de jouer pour des stratégies dépendant du coin de départ.

2. Stratégie globale

Pour tous nos algorithmes, le fonctionnement s'est essentiellement basé sur le **système 4X** avec 3 grands modules stratégiques :

- La gestion des péons (exploration, récupération de ressources, fuite).
- L'armée (attaque et défense).
- La production des nouvelles unités (construction de châteaux pouvant suplanter la gestion des péons, la production de nouveaux péons et de chevaliers).

3. Premier algorithme - Glutton

Le premier algorithme était basé sur un ensemble de fonctions de décisions en cascade :

- Chaque vague d'ordre opère de manière indépendante des autres et retire des unités de la banque totale, asséchant au fur et à mesure la quantité d'unités disponibles pour les suivantes.
- Le fonctionnement de chaque module était équivalent à un arbre de décisions ou chaque unité disponible était binairement utilisée ou laissée aux vagues suivantes jusqu'à une action par défaut.
- L'Heurstique était essentiellement globale puis appliquée à chaque unité pour prendre la décision de manière gloutonne à chaque étape.

Le système était initialement sans mémoire et utilisait directement les informations données par le serveur sans construction aditionnelle (essentiellement on ne prenait en compte que les coordonnées des unités sans classe associée) et souffrait ainsi de manque d'implication dans certaines actions. De même la gestion de l'armée était assez difficile car la séparation entre défense et attaque était rigide, menant parfois à des absurdités.

La logique de base a été finie assez vite mais les nombreuses itérations sur les arbres de décision internes à chaque module, ainsi que la restructuration des ordres des sous-modules entre eux a pris la majorité du temps de cette phase. (fuir avant de farmer des ressources, la gestion des algorithmes de déplacement des péons...).

Les 3 modules de décisions sont codés sur 4 modules pythons différents :

- Attack gère les prises de décisions liés à l'attaque de péons, châteaux et chevaliers ennemis. Il prédit les éventuels concéquences d'un combat et décide de qui attaquer en priorité.
- Defense gère les chevaliers associés à la défense, et les places en faces des menaces.
- Péons gère le farm des péons, leurs déplacements, leur explorations (avec un sous module d'exploration)...
- Castle gère la création d'unités sur les chateaux, et décide de quelle unité doit être utilisé construite.

Forces et faiblesses

- Forces :
  - ⋆ Gestion des péons que l'on considère presque optimal.
  - ⋆ Bonne aggressivité en début de partie.
- Faiblesses :
  - ⋆ Pas de mémoire entre les tours, ainsi si l'on perd la vision sur un chateau ennemis, l'on oublie son existance.
  - ⋆ Pas de stratégie globale, chaque chevalier décide indépendament de ses actions en attaque.
  - ⋆ La défense défend en priorité les points strategiques en avant même lorsque des chevaliers ennemis s'infiltres.
  - ⋆ Chaque chevalier se voit attribuer un rôle définitif à sa création, il y a donc un manque total de flexibilité. Certaines unités (défenseurs) sont inutilisés en fin de partie.
- Idées :
  - ⋆ Pour pouvoir améliorer cet algorithme et mener des actions coordonnées nous aurions besoin d'une mémoire sur plusieurs tours.

Le code nous a pris environ 350 heures de travail cumulé pour avoir le code actuel.

4. Refactor vers de l'orienté object - Memory

À partir de la semaine du 7 mai 2024, nous avons décider de faire un refactor total du code pour retravailler la sécurité et la mémoire du code.

En effet, la technique actuelle était très gloutonne, en ce que, à chaque tour, l'algorithme recalcule toutes les données, et puis décide le meilleur coup à jouer à ce tour.

Plus précisément, nous avons d'abord recodé toutes nos fonctions de la stratégie gloutonne avec la nouvelle structure orienté objet. Le fait que dans cette stratégie, les unités soient 'used' quand elles ont bougé a changé beaucoup de choses dans nos fonctions qui ne réfléchissaient pas de cette façon là. Cela a déjà pris pas mal de temps et nous avons du retravailler notre code pour qu'il fonctionne à nouveau comme dans la stratégie

gloutonne. On a enfin pu utiliser la structure orienté objet pour effectuer de nouvelles fonctions en tirant compte de notre mémoire.

Nous avons travaillé environ 250 heures sur cette branche.

4.a. Conservation des données

L'objectif de cette stratégie est de conserver la mémoire des événements et des données d'un tour au suivant, assurant ainsi une continuité et une cohérence dans le déroulement des actions. Nous avons donc trouvé la programmation orientée objet adéquate, en ce que nous pouvons ajouter des attributs pour stocker facilement les données. Ces objets forment des APIs faciles à utiliser et avec une facilité d'abstraction.

Ces données sont ensuite exploitées pour prendre des décisions. Elles permettent également de conserver des informations sur les unités, désignés comme `targets`. En effet, pour tuer des péons ennemis, on a besoin de le suivre, donc de garder des informations.

De plus, ces informations sont utilisées pour retenir des détails pertinents concernant les ressources financières sous forme d'or et les divers ennemis. Cette approche vise à optimiser la gestion et l'utilisation des données pour améliorer la prise de décision et renforcer la stratégie globale du jeu.

Les données stockées sont déjà triées (e.g.on trie déjà les bonnes et les mauvaises piles d'or avant de les stocker, même chose pour la répartition attaque défense).

API `Player` On a créé une classe joueur pour stocker les informations sur la partie (unités alliées, ennemis, piles d'or, etc.). On les conserve avec des informations supplémentaires. Par exemple, on stocke les piles d'or sous forme de good gold, bad gold, et on stocke la somme totale d'or.

À chaque fois qu'on farm une pile d'or, on met à jour ces valeurs. On stocke aussi les unités alliées et ennemies, et on les met à jour à chaque tour. Ainsi, lors que ces derniers sortent de la carte, on les garde dans la liste, et cela nous permet d'avoir plus d'information que ce que le serveur nous laisse avoir.

De plus, l'API permet de gérer facilement les initialisations pour forcer des cas de figures particuliers (en choisissant si les chevaliers du serveur vont à l'attaque ou à la défense par exemple).

API pour les unités Les unités aussi ont chacune leur classe (qui sont en fait des API). Ces classes permettent à notre code de jamais oublié des choses primordiales, comme update à la fois le serveur et les données de `player`, ou faire des vérifications de sécurité - comme la vérification de si on a suffisamment d'or pour créer l'unité, ou si la pile d'or n'a pas déjà été utilisée.

Exemple : utilisation de target Nos chevaliers ont donc maintenant une propriété `target`. Cette propriété est assignée par notre fonction `hunt`. Avec l'algorithme hongrois nos chevaliers se voient attribuées une cible qui est leur `target`. Quand des chevaliers ont la même `target`, ils sont gérés par `sync\_atk` qui va les déplacer dans le but d'attraper efficacement leur cible en la coinçant contre un bord. On peut en voir un exemple ici où le péon est coincé par les chevaliers et va se faire tuer au prochain tour.

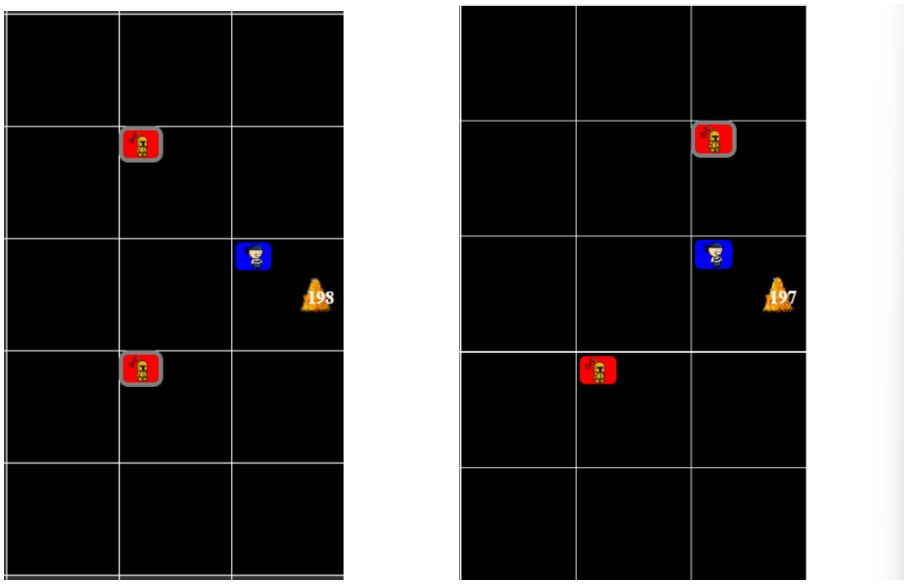

FIGURE 5 – Exemple d'attaqque synchronisée

4.b. Sécurité

On en a profité pour sécuriser tous les appels au serveur, en *crashant* si le serveur n'aime pas notre mouvement. En effet, dans la première version de notre code, nous partions du principe que notre code fonctionnait, mais on s'est rendu compte que ce n'était absolument pas le cas. On a pris donc beaucoup de temps à corriger toutes les erreurs de logique.

Ces erreurs consistent principalement en des mouvements illégaux (vers une case non voisine), de farm des piles d'or plus qu'une fois, ou encore bouger un même chevalier dans un même tour.

5. Heatmap et combats par Min-max

Pour une meilleure gestion de l'armée, il a été décidé de s'essayer à une combination d'une logique de Heatmap et de Min-max afin d'harmoniser et d'optimiser les mouvements des unités de l'armée.

Cet algorithme génère des heatmap tant d'un point de vue offensif que défensif et va utiliser deux algorithmes différents en fonction de la case la plus lumineuse :

- Si c'est une case d'attaque, on essaye de trouver les chevaliers ennemis et alliés à un rayon de 2 et on utilise un algorithme min-max pour leurs déplacements vers le point d'intérêt.
  - ⋆ Si aucun allié n'est trouvé, on regarde le chevalier allié le plus proche ainsi que tous les ennemis vers la case d'intérêt à une distance inférieure à 2 du chevalier et on min-max également.
- Dans le cas d'une défense, on essaye d'envoyer le chevalier le plus proche de manière sécurisée vers la case concernée.
- Dans tous les cas si on a un blocage et qu'aucun chevalier ne trouve de déplacement valide on marque la case comme inaccessible pour ce tour.
- Une fois l'un des algorithmes effectués, on actualise les "éclairages" de la Heatmap et on boucle jusqu'à épuiser les chevaliers disponibles (sachant que pour certains rester immobile sera leur action optimale qui sera utilisée, contrairement à auparavant.).

Forces et faiblesses

- Forces :
  - ⋆ Peu d'erreurs bêtes ou "d'oublis" de l'IA comparé aux implémentations précédentes, et meilleure stratégie long terme de nos chevaliers.
  - ⋆ Extrêmement solide une fois installé en endgame, notamment en terme de gestion des combats. Le minmax permet en effet d'aborder les combats de manières optimale.
- Faiblesses :
  - ⋆ Manque d'exploration en début de partie, c'est ainsi toujours l'adversaire qui fait le premier mouvement. Cela fait que même si nous faisons des échanges de combat équivalent, l'ennemi finit avec une meilleure économie car il possède plus de terrain.
  - ⋆ Temps de calcul très élevé en fin de partie (45 secondes par tour), voir plus dans certains cas isolés mais problématiques. Cela est dû au min-max, qui reste inefficace malgré certaines optimisations. D'autres tentatives d'optimisations ont été envisagé mais n'ont pas pus être implémentés par manque de temps.
- Idées :
  - ⋆ Forcer des bonus important à l'attaque pendant une première phase de la partie (jusqu'à voir n ennemis ou jusqu'au n^{eme} tour) afin de résoudre le manque d'agressivité.
  - ⋆ Beaucoup de constantes sont présentent dans le code mais nous manquons de test effectués pour déterminer les meilleures.

Ces algorithmes ont pris environ 55h de travail combiné pour 3 personnes sur la fin du projet.

6. Stratégies abandonnées

6.a. Mesure par zone

Il y a eu un mouvement de constructions d'un système pour mesurer différents états de la partie en fonction de 5 zones importantes de la carte :

- Ultra Arrière-garde
- Arrière-garde
- Centre
- Avant-garde
- Ultra avant-garde

Découpés en fractions d'environ $\frac{1}{8}$; $\frac{1}{4}$; $\frac{1}{4}$; $\frac{1}{4}$; $\frac{1}{8}$ relativement au joueur concerné

Avec pour objectif d'établir des stratégies basées sur le premier modèle mais en changeant l'ordre des opérations et les pondérations en fonction de l'avantage ou du désavantage dans chaque zone, notamment en cas de percée dans l'arrière-garde, et même encoder des réactions personnalisées pour gérer des problèmes que l'algorithme avait à cette époque.

L'idée de Heatmap a essentiellement remplacé ce concept car elle a réglé beaucoup des problèmes que cette solution cherchait à régler d'une manière plus élégante et généralisée.

Les essais ont pris environ 5h de programmation mais n'ont jamais vraiment été appliqués au-delà de la vérification des comptages.

6.b. Ligne de Front - Blitzkrieg

Une idée de stratégie plus “hard codé” nous est venu en tête, avec certes peu de chances que cette dernière marche vraiment. Mais c'était aussi l'intéret de tester notre version principale contre une autre version et ainsi faire ressortir des défaults visibles.

L'idée derrière la stratégie blitzkrieg était de former une ligne verticale de chevalier, et de faire avancer cette ligne sans distinction une fois que l'ennemis s'était présenté devant la ligne pour la briser. Comme il faut plus d'attaquant que de défenseur pour prendre une case, le reste des cases devaient être libres lors de la charge, et comme la ligne était déjà étandue, on pouvait espérer détruire rapidement l'économie de l'adversaire.

Malheuresement (mais sans grande surprise), la stratégie ne marche pas vraiment. Elle nécessite un temps au début de la partie pour former la ligne de front en arrière, chose possible uniquement sur les cartes où il n'y a pas de gold au centre.

Mais le projet à permis de remarquer certains défaults de la stratégie pricipale, notament le manque d'exploration des chevaliers inutilisé pour l'attaque (idée qui est maintenant implémentée), ce qui permettait à l'IA Blitzkrieg de se développer.

Le code a pris 6 heures pour être finalisé, et l'on a fait tourner des tests entre l'IA gloutonne et l'IA blitzkrieg pendant une à deux heures, en adaptant l'IA blitzkrieg et en notant les faiblesses de l'IA gloutonne.

6.c. Défense orientée chevalier ennemi

Une idée de défense pour résoudre les problème de chevaliers qui ne défendent pas sur les chevaliers ennemis était d'attribuer à chaque chevalier ennemi visible un chevalier allié. Ce chevalier doit se placer devant l'ennemi pour le bloquer et faire obstruction.

Cette idée à fait l'objet d'un début d'implémentation sur la base du systeme glouton. Elle n'as pas été terminée car avait besoins de la mémoire implémenté plus tard sur memory. Cette strategie qui aurait pu ralentir l'avancée des chevaliers ennemi aurait entrainé des pertes importantes de terrain du fait du délai entre le moment ou le chevalier ennemi est observé pour la première fois et le moment ou un chevalier allié est arrivé au contact.

Le temps passé à été de 3h.

6.d. Heatmap d'attaque ennemis en tant que Heatmap de défense

Il est possible de calculer, certes avec un manque d'information la heatmap d'attaque de l'ennemi. Utiliser cette heatmap en tant que heatmap de défense ou la combiner avec la heatmap de défense actuelle aurait pu produire de bon résultats.

Nous n'avons cependant pas eu le temps de l'implémenter.

7. Organisation du GIT

Le git a été initialement construit autour de la première version de l'algorithme puis branchée selon une hiérarchie spécifique :

- Depuis le main "première version", a été branch :
  - ⋆ Blitzkrieg

  - ⋆ L'implémentation en orienté objet
- Puis depuis l'orienté objet *OOP10* a été branch la Heatmap, qui récupérait régulièrement des fix venant des mises à jour de OOP10

Les différentes versions vivaient initialement parallèlement sur les diverses branches mais ont été rassemblées finalement sur la branche main pour le rendu final dans des sous-dossiers de strategies : - glutton pour la première version de l'algorithme - memory pour la version orientée objet finale - heatmap pour la stratégie utilisant la mémoire et les heatmaps pour les combats

La documentation plus détaillée des différents algorithmes est contenue dans le dossier docs sur lequel il est possible de générer une visualisation sur navigateur de la documentation complète.

Le dossier game-debug contient une version modifiée du jeu contenant des fonctionnalités additionnelles de debug, ainsi que des modifications à l'interface du jeu sur navigateur pour pouvoir jouer uniquement au clavier.

IV. Conclusion

Nous sommes plutôt content du résultat atteint à la fin du projet. Nos IA sont assez performantes et possèdes chacunes des forces et des faiblesse. Nous ne pensons pas qu'il existe de stratégie parfaite, car le manque d'information empêche de jouer parfaitement en fonction de l'adversaire pendant les premiers tours. Et comme le jeu est punitif, un petit retard accumulé au début peut avoir de grands impacts.

Nous aurions pu gagner un peu d'efficacité propre en modifiant légèrement le code glouton, mais nous estimions qu'il vallait mieux tenter de changer de manière d'aborder la stratégie. Ainsi, nous avons pu aborder tous les aspects de la stratégie que nous avons abordé dans notre état de l'art.

Avec plus de temps, nous aurions voulu tenter une "fusion" des IA en une seule, qui aurait changé sa stratégie en fonction de l'état de la partie.

Références

[1] *Illustration de Monte-Carlo.* URL : `https://upload.wikimedia.org/wikipedia/commons/thumb/0/04/MCTS.svg/1212px-MCTS.svg.png`.

[2] *Illustration de Monte-Carlo.* URL : `https://www.theseus.fi/bitstream/handle/10024/134060/Toni%5C_Laaveri.pdf?sequence=1`.

[3] *Illustration des arbres de décision.* URL : `https://s1.edi-static.fr/Img/FICHEOUTIL/2017/12/324014/355539.jpg`.

[4] *Illustration du min-max.* URL : `https://upload.wikimedia.org/wikipedia/commons/b/b4/Min_Max_Sample_3.png`.