Hex-Ta(c)tique
Projet de programmation 2

Auteurs

AL AYOUBI Ibrahim
BIGEY Raphaël
BONETTI Timothée
LEJEUNE Ivan

Encadrant

DA-SILVA Sébastien

12 mai 2024

Table des matières

1 Présentation du sujet

Les jeux de société ont toujours été un moyen de divertissement populaire, mais au-delà de leur aspect récréatif, ils constituent également des domaines d'étude intéressants pour les chercheurs en informatique, en mathématiques et en intelligence artificielle. Dans ce contexte, et dans le cadre de notre projet de programmation de L3 Informatique, nous avons choisi de nous intéresser à la théorie des jeux combinatoires. Nous avons ainsi mis en œuvre les jeux de stratégie combinatoire abstraits du Hex et de l'Awalé, reconnaissant ainsi leur valeur non seulement comme des défis ludiques, mais aussi comme des sujets de recherche passionnants pour explorer les interactions stratégiques et algorithmiques entre les joueurs.

Avant de poursuivre notre discussion sur notre motivation à étudier ces jeux, il est pertinent de clarifier ce que l'on entend par « jeu de stratégie combinatoire abstrait ». Ces jeux, généralement de société, sont des jeux :

- opposant généralement deux joueurs ou deux équipes (ou bien un joueur humain seul contre un ordinateur « intelligent »)
- dans lequel les joueurs ou équipes jouent à tour de rôle.
- dont tous les éléments sont connus (jeu à information complète).
- où le hasard n'intervient pas pendant le déroulement du jeu.

En d'autres termes, dans les jeux de stratégie combinatoire, la victoire dépend entièrement des actions des joueurs et de leur capacité à anticiper et à contrer les mouvements de l'adversaire.

L'étude de ces jeux nous motive à mieux comprendre les algorithmes de recherche et d'optimisation, à de nous familiariser avec les techniques de programmation avancée. À l'avenir, ces résultats pourront être utilisés dans un cadre plus général pour la résolution de problèmes plus complexes, par exemple pour les échecs ou le go.

Il existe plusieurs approches possibles pour la résolution de jeux de stratégie combinatoire. Parmi les approches les plus courantes, on trouve les algorithmes de recherche en profondeur, les algorithmes de recherche de chemin, les algorithmes de recherche de meilleure réponse, les algorithmes de *Monte-Carlo*, les algorithmes de renforcement, etc. Pour ce projet, nous avons choisi d'implémenter un algorithme d'intelligence artificielle en particulier pour la résolution des jeux du Hex et d'Awalé : l'algorithme *MinMax* avec élagage alpha-bêta. Les principaux avantages de cet algorithme sont les suivants :

- L'algorithme *MinMax* avec élagage alpha-bêta est un algorithme de recherche qui permet de trouver la meilleure stratégie pour un joueur dans un jeu à deux joueurs. Cet algorithme est très efficace pour les jeux de stratégie combinatoire comme le Hex. L'élagage alpha-bêta permet de reduire la complexité de cet algorithme.

Le cahier des charges détaillé est disponible en annexe avec les règles detaillées des jeux implementés.

2 Technologies utilisées

Pour la réalisation de ce projet, nous avons utilisé les langages de programmation suivants :

- Python pour l'implémentation des algorithmes de résolution des jeux et la logique du jeu,
- HTML, CSS et JavaScript pour l'implémentation de l'interface graphique des jeux,
- Git pour la gestion du code source et le suivi des versions,
- Visual Studio Code pour l'écriture du code et le débogage,
- GitHub pour l'hébergement du code source et la collaboration,
- LaTeX pour la rédaction du rapport,
- Lucidchart pour la création des diagrammes UML.

Nous avons choisi Python pour l'implémentation des algorithmes de résolution des jeux et la logique du jeu, car c'est un langage de programmation très populaire et très puissant. Il offre de nombreuses bibliothèques et modules pour le développement d'applications complexes. Python est également un langage de programmation très simple et très lisible, ce qui facilite la compréhension du code et la collaboration entre les membres de l'équipe. À noter que Python est un langage relativement lent et que l'utilisation d'un autre langage pour le backend pourrait réduire les temps de calcul, mais pour un projet de cette envergure, cela ne nous a pas semblé très pertinent.

Nous avons choisi HTML, CSS et JavaScript pour l'implémentation de l'interface graphique des jeux, car ce sont des langages de programmation efficaces et simples à comprendre. Ils permettent de créer des interfaces graphiques intéractives et ergonomiques. HTML est un langage de balisage qui permet de structurer les pages web. CSS est un langage de style qui permet de mettre en forme les pages web, et JavaScript est un langage de programmation qui permet de rendre les pages web intéractives.

Nous avons choisi UML pour la modélisation des classes et des cas d'utilisation. En effet, UML est un langage de modélisation très complet, il permet de représenter de façon claire et précise la structure et le comportement des systèmes informatiques.

Nous avons choisi Git et GitHub pour la gestion du code source et le suivi des versions, car ce sont des outils très puissants avec lesquels nous sommes très familiers.

LaTeX a été choisi pour la rédaction du rapport. C'est un langage de composition de documents très puissant et flexible, qui permet de créer des documents de grande qualité typographique. Étant donné que nous avons déjà utilisé LaTeX pour d'autres projets, nous avons préféré continuer à l'utiliser pour ce projet.

Lucidchart a été choisi pour la création des diagrammes UML, car c'est un outil assez simple qui répond parfaitement à nos besoins. Il permet de créer des diagrammes UML et de les exporter dans différents formats.

3 Développements Logiciel : Conception, Modélisation, Implémentation

3.1 Description des modules utilisés

Pour la réalisation de ce projet, nous avons développé une application web qui permet de jouer aux jeux du Hex et de l'Awalé avec la possibilité d'ajouter facilement d'autres jeux combinatoires. Ce logiciel est composé de plusieurs modules, dont les principaux sont les suivants :

- **Module Hex :** Ce module contient les classes et les fonctions nécessaires pour jouer au jeu du Hex. Il contient notamment la classe `HexBoard` qui représente le plateau de jeu et les joueurs du jeu de Hex. Ce module contient également les fonctions pour l'implémentation de l'algorithme *MinMax* avec élagage alpha-bêta pour la résolution du Hex.
- **Module Awalé :** Ce module contient les classes et les fonctions nécessaires pour jouer au jeu de l'Awalé. Il contient la classe `AwaleBoard` qui représente le plateau de jeu et les joueurs du jeu de l'Awalé. Ce module contient également les fonctions pour l'implémentation de l'algorithme de *MinMax* pour la résolution du jeu de l'Awalé. (à noter que la fonction *MinMax* est la même pour les 2 jeux. Seule les fonctions d'évalutaions changent.)
- **Module Interface Graphique :** Ce module contient les fichiers HTML, CSS et JavaScript nécessaires pour l'implémentation de l'interface graphique des jeux du Hex et d'Awalé. Il contient notamment les fichiers `home.html`, `hex.html` et `awale.html` qui permettent à l'utilisateur de choisir le jeu auquel il veut jouer et les paramètres de la partie. Il contient également des fichiers JavaScript pour la gestion des événements et des intéractions avec l'utilisateur.
- **Module Tests :** Ce module contient les fichiers de tests unitaires pour les classes et les fonctions des modules Hex et Awalé. Il contient notamment les fichiers `test_hex.py` et `test_awale.py`, qui permettent de tester les classes et les fonctions des modules Hex et Awalé.

Fonctionnalités de l'interface graphique L'interface graphique de notre logiciel comprend une page d'accueil qui permet à l'utilisateur de choisir le jeu auquel il veut jouer (Hex ou Awalé), une page principale (home) pour chaque jeu. Cela permet à l'utilisateur de choisir les paramètres de la partie comme la taille du plateau ou le mode de jeu. La page home permet aussi de consulter les règles du jeu que nous avons sélectionné. Lorsque l'on clique sur le bouton correspondant au mode de jeu choisi, nous sommes dirigés vers la page de jeu. Sur les deux jeux, le joueur à la possibilité de prévisualiser le prochain coup grâce à des fonctions de "hover" lorsqu'il survole avec la souris certains éléments. Lors d'une partie de Hex, il est possible de changer le thème graphique de la page avec la touche "H". Cela modifie la page css chargée dans le navigateur. L'interface graphique est implémentée en HTML, CSS et JavaScript. Celle-ci communique avec les modules Hex et Awalé via des appels de fonctions JavaScript à des fonctions Python via des requêtes json et le module Flask.

Structures de données Les principales structures de données définies dans le cadre de ce projet sont les classes `HexBoard` et `AwaleBoard`. Elles représentent respectivement les plateaux de jeu du Hex et de l'Awalé. Ces classes contiennent les attributs et les méthodes nécessaires pour représenter les plateaux de jeu et les joueurs, et pour effectuer les opérations de jeu (placement de pions, déplacement de pions, etc.) Les données sont fournies en entrée des programmes sous la forme de chaînes de caractères qui représentent les paramètres de la partie (taille du plateau, mode de jeu etc.) Les procédures de lecture et de validation des entrées sont effectuées par les fonctions des modules Hex et Awalé qui vérifient que les paramètres de la partie sont valides avant de commencer la partie.

Statistiques Voici quelques statistiques sur le logiciel développé dans le cadre de ce projet :

- Nombre de modules (sans les tests) : 39
- Nombre de classes : 11 (principalement dans les modules Hex et Awalé)
- Nombre de fonctions : 68 (en Python) et 99 (en JavaScript) pour 167 en tout
- Nombre de lignes de code : 1 405 (en Python), 2008 (en JavaScript), 553 (en HTML), 1 331 (en CSS) et 480 (en LaTeX) pour un total de 5 777 lignes de code

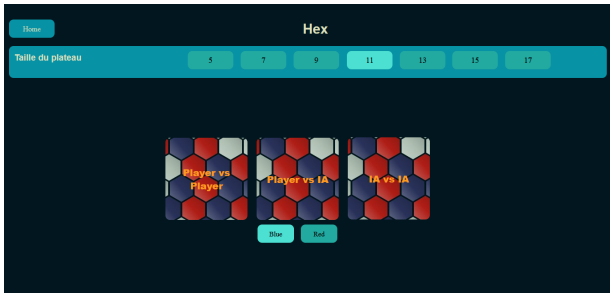

FIGURE 1 – Page Home du Hex

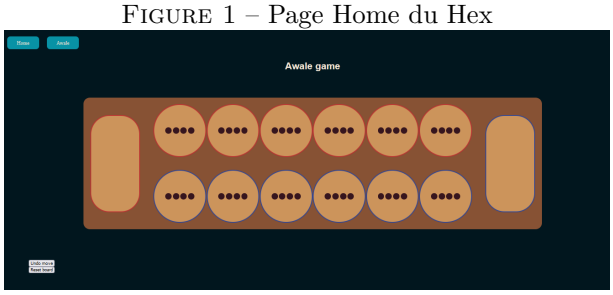

FIGURE 2 – Page du jeu Awalé

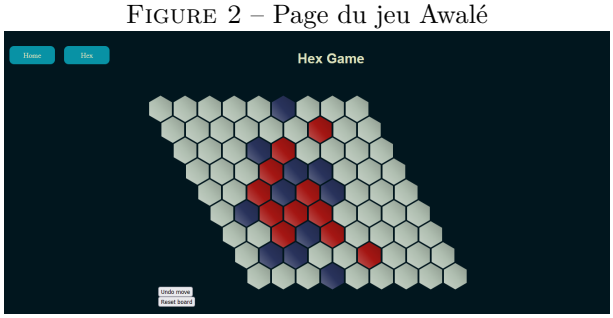

FIGURE 3 – Page du jeu Hex

3.2 Intéraction entre fichiers

L'intéraction entre les fichiers de l'application est la même lors d'une partie de Hex ou d'Awalé. Le fichier `app.py` héberge l'application flask et gère l'intéraction entre le frontend et le backend pour les deux jeux.

Initialisation de la partie La sélection du jeu et des paramètres de la partie se fait dans le fichier `home_hex.html` pour le Hex (respectivement `home_awale.html` pour l'Awalé). Ces informations seront envoyées au fichier `app.py` lorsqu'un clic souris sera effectué sur l'un des boutons. Le bouton sélectionné appellera dans le fichier `app.py` la fonction adéquate renvoyant la page html en prenant compte des choix effectués plus tôt. En parallèle, une instance du plateau de jeu désiré sera créée dans le fichier `app.py`.

Déroulement d'une partie Lorsqu'une partie est jouée, l'information des pièces placées au Hex, ou des graines déplacées sur l'Awalé est récupérée par le fichier JavaScript adapté : `game_hexia.js` si la partie est une partie de Hex JcIA, ou encore `game_awale.js` si la partie est une partie d'Awalé JcJ. Cette information est gérée par le capteur d'événements hex.onclick pour le Hex et pit.onclick pour l'Awalé. Si un tel événement est entendu, le fichier JavaScript correspondant envoie une requête json au fichier `app.py` afin de savoir si le coup est valide. La validité du coup est gérée par une fonction appelée dans la classe du plateau de jeu adéquat. Si le coup est valide, alors celui-ci est joué et rajouté à l'instance du plateau de jeu. Il est ensuite renvoyé au fichier JavaScript qui va s'occuper du nouvel affichage graphique (En modifiant le CSS ou le html). Si une erreur est attrapée, elle est renvoyée au fichier JavaScript qui va afficher une alerte sur l'écran du joueur, indiquant la non-validité du coup. Lors du placement de

chaque hex (déplacement de graines pour l'Awalé), la fonction *check_winner* de la classe du plateau est appelée afin de vérifier si un joueur a gagné. Si c'est le cas, alors la fonction questionnée par la requête json renvoie la variable `game_over` égale à **True**. En récupérant cette variable, le fichier JavaScript va pouvoir procéder à une animation indiquant au(x) joueur(s) que la partie est terminée.

3.3 Fonctions asynchrones

L'envoie des différentes requêtes des fichiers JavaScript à `app.py` se fait grâce aux différents appels à la fonction *fetch*. La méthode globale *fetch* démarre le chargement d'une ressource sur le réseau et retourne une promesse qui est résolue dès que la réponse est disponible. Cependant, le fait que la réponse ne soit renvoyée que lorsque celle-ci est disponible nous a gêné lors de l'implémentation des modes JcIA et IAvIA pour les deux jeux. En effet, nous souhaitons que notre programme s'exécute de façon synchrone (ligne après ligne). Or, le *fetch* peut prendre un certain temps avant de résoudre la promesse faite, cela est notamment dû à la lenteur des fonctions d'évaluation. Ainsi, pour éviter que le reste du programme ne s'exécute avant la résolution de la prommesse, nous avons décidé d'ajouter le décorateur `async` à notre fonction *window.onload*. Ce simple décorateur nous a alors permis d'utiliser le mot-clé `await` devant les requêtes *fetch*. Celui-ci interrompt l'exécution de la fonction asynchrone et attend la résolution de la promesse passée. Ainsi, les requêtes se voient toujours résolues avant d'exécuter la suite du programme. Cela permet d'éviter que le joueur puisse jouer plusieurs coups pendant que l'IA « réfléchie », ou encore le fait qu'une des deux IA dans le IAvIA joue plusieurs fois avant que l'autre ne joue.

3.4 Diagrammes UML

3.4.1 Diagramme du module Hex et Awalé

Le diagramme suivant représente les classes et les fonctions du module Hex et du module Awalé. Il montre les classes `HexBoard` et `AwaleBoard` qui représentent les plateaux de jeu du Hex et de l'Awalé, ainsi que les fonctions pour l'implémentation des différents algorithmes de jeu. Il contient également les classes d'exceptions pour gérer les erreurs et les exceptions dans le jeu.

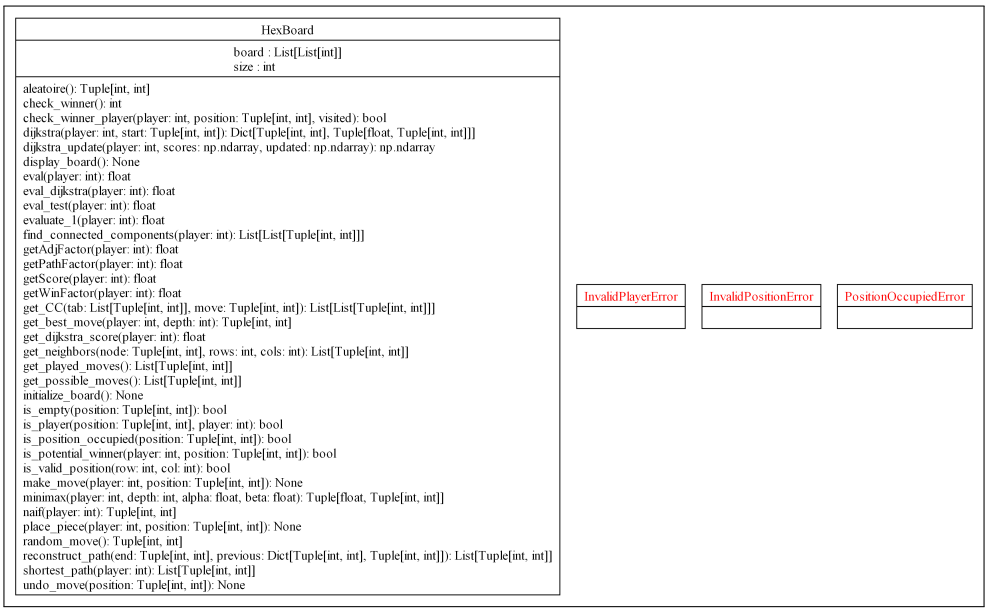

FIGURE 4 – Diagramme UML du module Hex

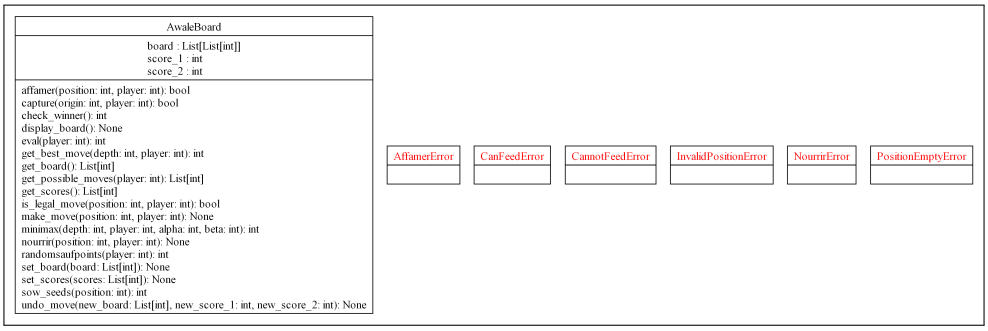

FIGURE 5 – Diagramme UML du module Awale

Ces diagrammes UML ont pû être générés automatiquement à partir du code source grâce à 'pyreverse' qui est un outil de génération de diagrammes UML pour Python.

3.4.2 Diagramme de fonctionnement de l'application

Le diagramme suivant représente le fonctionnement de l'application. Il montre les différentes classes et fonctions de l'application, ainsi que les interactions entre elles. Il montre comment l'interface graphique communique avec les modules de jeu pour afficher le plateau de jeu et les coups joués par les joueurs. Les détails ne sont pas affichés pour des raisons de lisibilité, de même que seul le module Hex est représenté en détail.

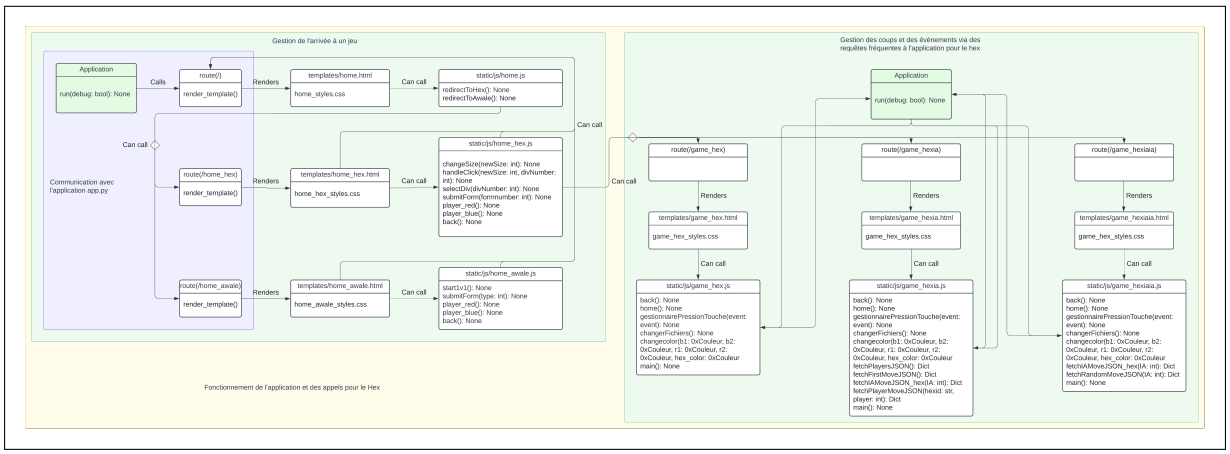

FIGURE 6 – Diagramme UML de l'application

4 Moteur de jeux combinatoires abstraits et algorithmes

Implémenter un algorithme pour jouer à des jeux combinatoires contre l'ordinateur est l'un des objectifs principaux du projet Hex-Ta(c)tique, mais plusieurs questions naturelles ont alors vu le jour : quels sont les algorithmes qui permettent à un ordinateur de jouer à un jeu combinatoire abstrait ? Quels sont leurs points forts et faibles ? Dans cette section, nous allons présenter deux algorithmes principaux qui aident à répondre à ces questions : la recherche arborescente *Monte-Carlo* (Monte-carlo Tree Search) et l'algorithme du *MinMax* avec élagage alpha-bêta :

4.1 La recherche arborescente Monte-Carlo

Présentation La recherche arborescente *Monte-Carlo* ou *Monte-Carlo* (MCTS) est un algorithme de recherche heuristique. Il est principalement utilisé dans le cadre de mise en place d'intelligence artificielle pour des jeux tels que le go, mais pas uniquement. En effet, il peut aussi être utilisé pour les moteurs de jeux des échecs comme le moteur AlphaZero de Google qui est l'un des leaders en termes d'intelligence de jeux aux échecs. Cet algorithme peut même être implémenté dans des jeux où le hasard apparaît par exemple au poker.

MCTS - Fonctionnement succinct *MCTS* est un algorithme qui explore l'arbre des possibles. La racine est la configuration initiale du jeu. Chaque nœud est une configuration (une situation en jeu) et ses enfants sont les configurations suivantes. *MCTS* conserve en mémoire un arbre qui correspond aux nœuds déjà explorés. Une feuille de cet arbre est soit une configuration finale (donc un des joueurs a gagné), soit un nœud dont aucun enfant n'a encore été exploré. Dans chaque nœud, on stocke deux nombres : le nombre de simulations gagnantes, et le nombre total de simulations.

À chaque itération *MCTS* va chercher la feuille la plus prometteuse, comprendre la suite de coups qui possède la meilleure heuristique, et ensuite, depuis cette feuille, créer grâce aux règles du jeu, une nouvelle feuille au hasard dont le but est d'atteindre une configuration finale. Dans le cas où on trouverait une configuration gagnante pour un joueur, on va incrémenter le nombre de simulations gagnantes, pour les nœuds correspondant au joueur et dans les parents de la feuille que l'on vient de créer. L'algorithme répète cette opération un certain nombre de fois avant de choisir un coup, i.e. : choisir le coup qui a le plus de simulations gagnantes et le moins de simulations totales.

4.2 MinMax

4.2.1 Présentation

L'algorithme *MinMax*, comme *MCTS*, est un algorithme décisionnel. Cependant, celui-ci s'applique sur des jeux à somme nulle : c'est-à-dire un jeu où la somme des gains et des pertes de tous les joueurs est égale à 0. Ainsi, le gain de l'un constitue obligatoirement une perte pour l'autre. L'algorithme va utiliser cette propriété dans sa recherche pour trouver le meilleur coup possible. En effet, l'ordinateur va passer en revue toutes les configurations possibles pour un nombre limité de coups (que l'on appellera profondeur), et leur assigner une valeur qui prend en compte les bénéfices pour le joueur et pour son adversaire. Le meilleur coup est alors celui qui minimise les pertes du joueur, tout en supposant que l'adversaire cherche au contraire à les maximiser (d'où le nom *MinMax*).

4.2.2 MinMax - Fonctionnement succinct

Comme le *MCTS*, le *MinMax* va explorer l'arbre des possibles où chaque nœud représente une configuration du jeu. L'algorithme va explorer toutes les possibilités et associer à chaque feuille une valeur positive ou négative. Ces feuilles sont soit des nœuds terminaux (c'est-à-dire que l'un des deux joueurs a gagné), soit des nœuds à la profondeur de recherche maximale. Un nœud va se voir assigner une valeur positive si la position associée favorise l'ordinateur (le joueur maximisant) et négative dans le cas contraire.

En ce qui concerne les nœuds feuilles non-terminaux, c'est-à-dire ceux représentant une configuration de jeu non gagnante, mais bloqués par la profondeur de recherche, c'est une fonction d'évaluation qui va estimer leur valeur heuristique. La qualité de cette fonction va déterminer « l'intelligence » de l'ordinateur.

Cette estimation et la profondeur de recherche déterminent donc la qualité et la précision du résultat final du *MinMax*.

Les nœuds qui ne sont pas des feuilles vont hériter d'une des valeurs de leurs enfants en fonction de s'ils sont le minimisant, ou le maximisant. Si le rôle du nœud est de maximiser sa valeur, alors celui-ci va recevoir la plus grande valeur associée à ses fils. Sinon, son rôle est de minimiser sa valeur, et donc il va recevoir la plus petite valeur associée à ses fils. Ainsi, les nœuds conduisant à un résultat favorable, comme une victoire pour le joueur maximisant, ont des scores plus élevés que les nœuds plus favorables pour le joueur minimisant. Les valeurs des nœuds qui mènent à une victoire sont -inf ou + inf ou 0 en fonction de : la victoire, la défaite ou l'égalité pour le joueur maximisant.

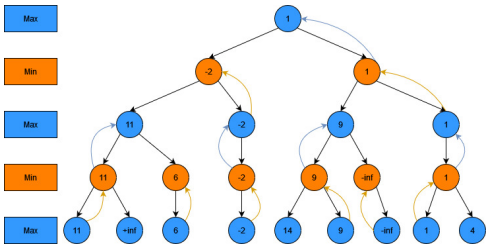

FIGURE 7 – Ici, le joueur bleu cherche à maximiser ses gains avec une profondeur 4.

4.2.3 Élagage alpha-bêta

L'algorithme *MinMax* effectue une exploration complète de l'arbre de recherche jusqu'à un niveau donné. L'élagage alpha-bêta permet d'optimiser cet algorithme sans en modifier le résultat. Pour cela, il ne réalise qu'une exploration partielle de l'arbre. En effet, on observe qu'il n'est pas utile d'explorer les sous-arbres qui conduisent à des valeurs qui ne participeront pas au calcul de la valeur associée à la racine de l'arbre. Dit autrement, l'élagage alpha-bêta n'évalue pas des nœuds dont on peut penser, si la fonction d'évaluation est à peu près correcte, que leur qualité sera inférieure à celle d'un nœud déjà évalué.

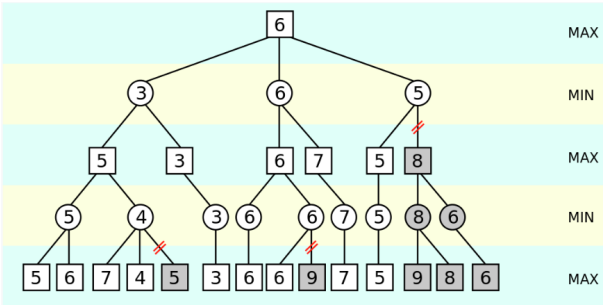

FIGURE 8 – *MinMax* avec élagage α-β.

Plusieurs coupures ont pu être réalisées. De gauche à droite :

1. Le nœud MIN vient de mettre à jour sa valeur courante à 4. Celle-ci, qui ne peut que baisser, est déjà inférieure à α=5, la valeur actuelle du nœud MAX précédent. Celui-ci cherchant la plus grande valeur possible, ne la choisira donc de toute façon pas.
2. Le nœud MIN vient de mettre à jour sa valeur courante à 6. Celle-ci, qui ne peut que baisser, est déjà égale à α=6, la valeur actuelle du nœud MAX précédent. Celui-ci cherchant une valeur supérieure, il ne mettra de toute façon pas à jour sa valeur que ce nœud vaille 6 ou moins.
3. Le nœud MIN vient de mettre à jour sa valeur courante à 5. Celle-ci, qui ne peut que baisser, est déjà inférieure à α=6, la valeur actuelle du nœud MAX précédent. Celui-ci cherchant la plus grande valeur possible, ne la choisira donc de toute façon pas.

4.2.4 Remarque

À l'aide de l'algorithme *MinMax*, il est possible de donner une preuve de l'existence d'une stratégie gagnante au jeu du Hex notamment. Nous détaillons cette preuve en annexe.

4.3 Autres algorithmes

4.3.1 Dijkstra et BFS

Dans le projet, lorsqu'un joueur gagne au Hex, nous avons décidé de mettre en valeur son chemin gagnant. Dans un premier temps, nous devions détecter qu'un joueur ait remporté une partie. Pour cela nous avons implementé l'algorithme du parcours en largeur (*BFS*). Nous avons alors interprété le tableau dans lequel les coups des joueurs sont sauvegardés comme une graphe. Le parcous en largeur va générer un graphe couvrant depuis un bord. Ce graphe couvrant ne va passer que par les coups placés par un joueur. Si le graphe généré atteint le bord opposé, alors un chemin relie les deux bords, et donc le joueur a gagné. Nous appellons donc cet algorithme à chaque fois qu'un coup est placé. Sa complexité est $O(|V| + |E|)$. Notons ici que nous appelons l'algorithme seulement sur l'arbre correspondant aux coups joués par un joueur, donc en pratique, l'appel à cet algorithme est très rapide.

L'algorithme *BFS* nous assure qu'un joueur a gagné, mais il peut exister plusieurs chemins distincts reliant les deux bords. En effet, il peut y avoir plusieurs points de départ (points sur l'un des bords du gagnant) et points d'arrivée (points sur le bord opposé). Afin d'afficher le plus court chemin, nous avons décidé d'intégrer une version modifiée de l'algorithme de *Dijkstra*. Nous avons opté pour une solution naïve. En effet, nous appellons l'algorithme plusieurs fois (une pour chaque point de départ possible). Ensuite nous comparons la taille des chemins trouvés. Nous obtenons alors le plus court chemin, que l'on a choisi de mettre de jaune afin de faciliter sa visualisation.

À l'origine, dans le pire des cas, la complexité de l'algorithme de *Dijkstra* est $O(|V| + |E| \times \log(|V|))$. Cependant, notre implémentation est plus complexe, notre algorithme possède alors une complexité de $O(n \times (|V| + |E| \times \log(|V|)))$, avec n la taille de notre plateau. Notons que ce cas n'arrive en pratique jamais. En pratique le chemin le plus court est trouvé de façon instantanée.

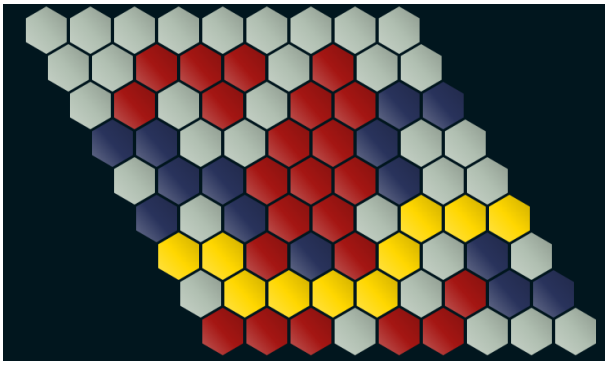

FIGURE 9 – Ici bleu a gagné, le chemin le plus court est trouvé

5 Observations de l'algorithme MinMax sur nos jeux

Une fois l'implémentation de l'algorithme *MinMax* faite sur nos deux jeux combinatoires, nous pouvons nous demander, est-ce que cet algorithme est adapté pour le jeu du Hex et de l'Awalé ? Nous observons que les performances du *MinMax* sont très différentes d'un jeu à l'autre. Dans cette section nous étudierons ces différences.

5.1 Hexgame : Performances décevantes

L'algorithme *MinMax* développé sur le jeu du Hex n'arrive quasiment jamais à battre un humain, et cela, pour deux raisons principales : la faible profondeur, et la difficulté à trouver une fonction d'évaluation satisfaisante.

Trop grande complexité Le premier problème provient du nombre de coups possibles à chaque tour. En effet, sur un plateau classique de 13×13 le premier joueur a 169 coups possibles. Au coup suivant, il y en a 168 etc. . .Cela rend les calculs lents. Si l'ordinateur joue en premier, et que la profondeur de calcul est de 6, celui-ci doit calculer pas moins de $2.1298467e+13$ (soit $169 \times 168 \times 167 \times 166 \times 165 \times 164$) fois la fonction d'évaluation. Cela n'est pas réalisable en temps réaliste. Dans notre implémentation, la taille des côtés du plateau varie entre une longueur de 5 cases et de 17. Nous avons remarqué que pour un plateau de dimension 11×11, et une profondeur demandée de 4, l'algorithme mettait déjà plusieurs secondes à calculer le meilleur coup. Notons que ces calculs ne prennent pas en compte l'élagage alpha-bêta qui optimise significativement le *MinMax*. Même si une portion des nœuds n'est pas évaluée, le calcul reste trop lent pour pouvoir demander une profondeur plus intéressante de 6 ou de 8 par exemple. On peut aussi noter que la performance de l'élagage alpha-bêta dépend de la qualité de la fonction d'évaluation, qui, comme nous allons le voir, n'est pas assurée.

Fonction d'évaluation : Un vrai casse-tête Le *MinMax* n'arrivant pratiquement jamais à une feuille terminale, nous avons compris que la fonction d'évaluation du Hex devait être performante et peu coûteuse en temps. Cependant, faire comprendre à l'ordinateur si une position est gagnante ou non s'est révélé très difficile. En effet, le Hex est un jeu possédant de nombreuses stratégies, et relier toutes ces stratégies en une seule fonction d'évaluation n'est pas chose aisée. Ainsi, nous avons rapidement cherché à faire comprendre à l'ordinateur quelle stratégie adapter au fur et à mesure de la partie. Parmi elles, la stratégie de faire des ponts (voir la figure 3) nous a posé beaucoup de problèmes. Un pont permet au joueur de s'assurer de pouvoir connecter deux pièces lors d'un prochain coup. En effet, si le joueur adversaire cherche à bloquer l'une des deux cases, nous avons juste à ajouter notre jeton afin de créer un chemin entre ces deux pièces. Nous pouvons noter que lorsque la profondeur de réflexion du *MinMax* est grande, il arrive parfois à remarquer que créer un pont est le meilleur coup. Mais nous revenons au problème n°1 : la complexité. Voir la section sur les fonctions d'évaluations pour voir quelles idées nous avons essayé. Notre fonction finale est capable de battre des humains sur de petits plateaux. Mais en général, lorsque le plateau est de grande dimension, la fonction d'évaluation prend de nombreuses secondes, voire minutes avant de jouer.

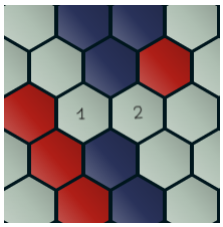

FIGURE 10 – Ici bleu a créé un pont [1].

1. Si rouge joue sur la case 1, bleu joue sur la case 2, et relie ainsi les deux jetons bleus (respectivement si rouge joue sur la case 2)

5.2 Awalé : Bonnes performances

Une fonction d'évaluation efficace Le jeu de l'Awalé possède un avantage significatif par rapport au Hex, le nombre maximum de coups possible pour un joueur est de 6. Ainsi, l'arbre de recherche est bien plus petit que celui du Hex. L'algorithme *MinMax* est donc efficace en termes de temps et de performances avec une grande profondeur (la profondeur initiale est de 6 ou 8).

Le choix de notre fonction d'évaluation est arrivé naturellement pendant nos recherches. Nous avons implémenté la simple fonction cherchant à maximiser les points de l'ordinateur tout en minimisant les points de son adversaire. Cette fonction combinée à une profondeur de taille 6 en fait un adversaire redoutable que peu d'humains ont réussi à battre.

5.3 Conclusion

L'algorithme *MinMax* est donc capable d'être efficace lorsque le nombre de nœuds est petit à chaque itération. On peut alors lui mettre une profondeur relativement grande afin d'explorer toutes les branches de cet arbre. Cependant, lorsque le nombre de nœuds devient gigantesque, comme dans le Hex, l'algorithme ne devient plus très bon. Il est en effet très lent et peu performant. Dans la bibliographie est disponible un lien vers un papier scientifique cherchant une bonne fonction d'évaluation pour le Hex. Dans celui-ci, il est écrit que les ordinateurs battent les humains pour des petits plateaux, font jeu égal pour le plateau 9x9 mais sont moins bons qu'un humain pour un plus grand plateau. On y voit aussi la difficulté de trouver une bonne fonction d'évaluation.

5.4 Fonctions d'évaluation

L'évaluation d'une position est une étape cruciale dans la conception d'un programme de jeu. Elle permet de déterminer la force relative des positions des deux joueurs et d'orienter le choix du coup à jouer à la fois pour une IA et pour un joueur humain qui conceptualise la stratégie à suivre. Dans le cadre de notre projet, nous avons implémenté plusieurs fonctions d'évaluation pour le jeu Hex.

5.4.1 Fonction d'évaluation initiale

La première fonction d'évaluation implémentée était très simple. Elle attribuait un score de 1 à chaque case du plateau occupée par une pièce du joueur et un score de -1 à chaque case occupée par une pièce de l'adversaire. Le score final de la position dépendait alors de la différence entre le nombre de cases occupées par le joueur et le nombre de cases occupées par l'adversaire autour de la case centrale du plateau. Cette fonction d'évaluation a été implémentée pour tester le fonctionnement du jeu et de l'IA, mais elle s'est avérée trop simpliste pour être efficace.

5.4.2 Fonction d'évaluation basée sur la proximité au bord

La deuxième fonction d'évaluation implémentée prenait en compte la proximité des cases au bord du plateau. Elle attribuait un score plus élevé aux cases proches du bord, car elles sont plus stratégiques et plus difficiles à bloquer. Cette fonction d'évaluation a permis d'améliorer les performances de l'IA, mais elle n'était pas suffisante pour prendre en compte les stratégies de blocage et de connexion des pièces.

De plus, lors de l'implémentation de cette fonction, nous avons rencontré des difficultés pour catégoriser les cases qui étaient similaires en termes de proximité au bord. Cela a conduit à des situations où l'IA plaçait ses pièces dans un ordre aléatoire, ce qui n'était pas optimal.

5.4.3 Fonction d'évaluation basée sur les voisins

La troisième fonction d'évaluation implémentée comptait le nombre de voisins de chaque joueur et essayait de bloquer l'ajout d'un autre voisin pour ne pas former une chaîne en plus. En bref, l'idée de cette fonction était de vérifier l'intérêt à bloquer les chaînes de l'adversaire plutôt que de former les siennes. Cette fonction d'évaluation nous a permis de mieux comprendre les stratégies de blocage et de connexion des pièces, mais elle n'était toujours pas suffisante et était facile à contrer.

5.4.4 Fonction d'évaluation basée sur la position actuelle

La quatrième fonction d'évaluation implémentée évaluait la position actuelle sur le plateau de jeu pour déterminer la force relative des positions des deux joueurs. Elle prenait en compte les composantes connectées des pions de chaque joueur, leur proximité au centre du plateau et la possibilité de victoire imminente. C'était un grand pas en avant par rapport aux fonctions d'évaluation précédentes car elle changeait activement de stratégie en fonction de la position actuelle sur le plateau. Elle était donc moins 'prévisible' pour l'adversaire et moins facile à contrer. Néanmoins, cette fonction d'évaluation était toujours basée sur des critères statiques et ne prenait pas complètement en compte les stratégies de blocage et de connexion des pièces.

5.4.5 Fonction d'évaluation basée sur l'algorithme de Dijkstra

Un des gros problèmes rencontrés lors de l'implémentation des fonctions d'évaluation était qu'on implémentait des 'idées' de stratégies sans vraiment les formaliser. Cela a conduit à des fonctions d'évaluation très complexes et difficiles à optimiser sans pour autant être efficaces. On a donc décidé d'implémenter une fonction d'évaluation basée sur l'algorithme de Dijkstra pour mettre à jour les scores des cases sur le plateau en trouvant les chemins les plus courts du point de départ du joueur vers chaque case. Cette fonction d'évaluation permettait de déterminer les cases qui rapprochaient le joueur de la victoire et de les prioriser dans le choix du coup à jouer.

On avait alors une fonction d'évaluation qui 'visualisait' les chemins possibles pour chaque joueur et qui permettait de déterminer les coups les plus stratégiques à jouer. Cependant, cette fonction d'évaluation était très complexe (en termes de temps de calcul) car elle nécessitait de parcourir une grande partie

du plateau pour chaque coup. Cela a conduit à des temps de calcul très longs et à des performances globales de l'IA qui n'étaient pas satisfaisantes.

5.4.6 Le problème de la complexité

La dernière fonction d'évaluation implémentée, bien que plus efficace que les précédentes, souffrait toujours du problème de complexité. En effet, les fonctions d'évaluation étaient de plus en plus complexes et nécessitaient de plus en plus de calculs pour déterminer le score d'une position. Plusieurs choix se présentent alors pour résoudre ce problème :

- Réduire la complexité des fonctions d'évaluation en simplifiant les critères pris en compte.
- Optimiser les fonctions d'évaluation pour réduire le temps de calcul.
- Implémenter des algorithmes de recherche plus efficaces pour déterminer le coup à jouer.

Dans le cadre de notre projet, nous avons choisi de simplifier les fonctions d'évaluation et d'optimiser les algorithmes de recherche pour améliorer les performances de l'IA. Cela a permis de réduire la complexité du programme et d'obtenir des résultats plus satisfaisants en termes de temps de calcul et de performances globales de l'IA. Mais, comme pour tout problème de recherche, il n'y a pas de solution parfaite et il est important de trouver un compromis entre la complexité des fonctions d'évaluation et les performances de l'IA :

- Réduire la complexité des fonctions d'évaluation peut entraîner une perte de précision dans l'évaluation des positions et une moins bonne performance de l'IA.
- Optimiser les fonctions d'évaluation peut permettre d'améliorer les performances de l'IA, mais cela nécessite beaucoup de travail et atteint rapidement des limites : il est difficile d'optimiser une fonction d'évaluation complexe sans la simplifier.
- Implémenter des algorithmes de recherche plus efficaces peut permettre d'améliorer les performances de l'IA, mais cela nécessite également beaucoup de travail et de recherche pour trouver les algorithmes les plus adaptés au jeu et à l'IA.

5.4.7 Conclusion

En conclusion, cette partie aborde les différentes approches d'évaluation implémentées dans le projet. Les évaluations ont évolué pour être plus précises et tenir compte de la proximité du bord, des stratégies de blocage et de la recherche de chemins optimaux vers la victoire, tout en essayant de réduire le temps de calcul et la complexité du programme. Cependant, le problème de la complexité reste un défi majeur dans la conception d'une IA pour le jeu Hex, et il est donc toujours possible d'améliorer les fonctions d'évaluation et les algorithmes de recherche pour obtenir de meilleurs résultats. Dans le cadre de notre projet, nous avons choisi de passer ce temps à améliorer d'autres aspects du jeu, mais il reste encore beaucoup à faire pour obtenir une IA optimale pour le jeu Hex.

6 Gestion du Projet

Pour la gestion du projet, nous avons suivi un cycle de développement similaire à celui de la méthode Scrum. Nous avons donc défini des sprints courts de quelques jours chaque semaine, avec une réunion hebdomadaire le mercredi pour faire le point sur l'avancement du projet. Ces réunions étaient l'occasion de discuter des tâches effectuées, de celles à venir, et de résoudre les problèmes rencontrés. Nous avons également utilisé le logiciel Jira pour gérer les tâches et les sprints, et pour suivre l'avancement du projet.

6.1 Diagramme de Gantt

Officiellement, nous n'avons pas utilisé de diagramme de Gantt pour la gestion du projet, mais nous avons tout de même réalisé un planning prévisionnel des tâches à effectuer. Ce planning a été réalisé en début de projet, et a été mis à jour régulièrement pour refléter l'avancement du projet. Il a été utilisé pour définir les tâches à effectuer pour chaque sprint, et pour suivre l'avancement du projet. Un exemple de diagramme de Gantt est présenté en annexe 14.

6.2 Organisation

Avec ce fonctionnement, chaque semaine nos avions de nouvelles tâches à accomplir, celles-ci pouvaient être par exemple ajouter une fonctionnalité ou le débogage, l'ajout d'éléments graphiques, etc... Les plus grosses tâches étant l'ajout de nouveaux jeux, pour celles-ci nous avons procédé comme ceci : Une première version du jeu est ajoutée avec sa classe Python. Cette première version contient les fonctionnalités de base et est jouable dans le terminal dans le but de faire des tests. L'étape suivante est de créer une page web d'accueil et une autre pour le jeu, ainsi que de créer le plateau de jeu en HTML/CSS/JavaScript. La dernière étape est de relier la classe Python avec le site web en utilisant Flask. À ce stade, le jeu est jouable sur le site web. Ce cycle a duré 2 à 3 semaines pour les 2 jeux que nous avons implémentés. Ensuite vient la phase où il faut déboguer et ajouter les fonctionnalités comme la possibilité de jouer contre l'IA ou améliorer l'interface graphique. En pratique, l'implémentation du jeu prend beaucoup moins de temps que de l'optimiser, le déboguer, et ajouter toutes sortes de petites fonctionnalités. Nous pouvons noter que grâce à l'architecture du projet, il est facile d'ajouter de nouveaux jeux : il suffit d'écrire une classe similaire à celle d'un autre jeu et d'ajouter une page web correspondante. C'est aussi le cas pour certaines fonctionnalités comme le MinMax : il suffit de recopier l'algorithme dans la nouvelle classe (notez que ce n'est pas du tout le cas pour les fonctions d'évaluation qui sont propres à chaque jeu). Nous avons construit le projet comme une plateforme pour les jeux combinatoires, pour y jouer et pour essayer les différents moteurs de jeux que nous pouvons implémenter.

6.3 Changements Majeurs

Les reunions hebdomadaires etaient l'occasion de nous mettre d'accord entre nous et avec notre encadrant sur les changements majeurs a apporter sur le projet pendant le developpement, en voici une liste non exaustive :

- Changement de la hiérarchie des classes : nous avons changé la hiérarchie des classes pour une homogénéisation du code et une meilleure compréhension.
- Changement de l'organisation des fichiers : nous avons changé l'organisation des fichiers pour mieux organiser le code et faciliter la maintenance. Cela a initialement été fait pour le backend, puis pour le frontend. Beaucoup de problèmes ont été rencontrés lors de ces changements, mais ils ont permis d'améliorer la qualité du code.
- Changement de la gestion des erreurs : nous avons changé la gestion des erreurs pour une meilleure gestion des exceptions et une meilleure gestion des erreurs. Cela a permis d'améliorer la robustesse du code. Cela a également renforcé la sécurité du code sans nuire à la performance ou au debuggage.

7 Bilan et Conclusions

Nous avons implémenté la plupart des fonctionnalités prévues dans le cahier des charges, à l'exception de quelques fonctionnalités mineures. Nous avons également ajouté des fonctionnalités supplémentaires qui n'étaient pas prévues dans le cahier des charges, mais qui ont été jugées nécessaires pour améliorer la qualité du projet.

En bref, nous avons implémenté les fonctionnalités suivantes :

- Possibilité pour les utilisateurs de jouer en JcJ au *Hex* et à l'*Awalé*
- Possibilité pour les utilisateurs de jouer au JcIA au *Hex* et à l'*Awalé*
- Possibilité pour les utilisateurs de regarder deux IA jouer au *Hex* et à l'*Awalé*
- Possibilité pour les utilisateurs de changer de thème de couleur pour le jeu du *Hex*
- Possibilité pour les utilisateurs de défaire un coup dans tous les modes de jeu (sauf IAvIA), pour le *Hex* et l'*Awalé*
- Possibilité pour les utilisateurs de rejouer une partie dans tous les modes de jeu, pour le *Hex* et l'*Awalé*
- Possibilité pour les utilisateurs de choisir la taille du plateau de jeu pour le *Hex*
- Possibilité pour les utilisateurs de choisir son camp (en JcIA) pour le *Hex* et l'*Awalé*
- Possibilité pour les développeurs de tester, déployer et maintenir facilement l'application
- Possibilité pour les développeurs de rajouter facilement de nouvelles fonctionnalités
- Possibilité pour les développeurs de rajouter facilement de nouveaux jeux
- Possibilité de changer le thème visuel pour le hex (en appuyant sur la touche H)

Si nous avions continué le projet plus longtemps, nous aurions implémenté les fonctionnalités suivantes :

- Possibilité de jouer en ligne contre un autre joueur
- Possibilité de choisir un niveau de difficulté pour l'IA
- Implémentation d'autres algorithmes de jeu comme *Monte-Carlo* par exemple
- Mise en place d'une base de données pour enregistrer des scores
- Tableau des "high scores"
- Implémentation d'autres jeux combinatoires abstraits
- Ajout de sons (musiques et effets sonores)

8 Bibliographie

Title	URLs
Hex	• https://www.pedagogie1d.ac-nantes.fr/medias/fichier/regles-jeu-de-hex_1465481131499-pdf?ID_FICHE=434770&INLINE=FALSE • http://www.cs.cornell.edu/~adith/docs/y_hex.pdf • https://en.wikipedia.org/wiki/Hex_(board_game) • https://gsurma.medium.com/hex-creating-intelligent\protect\penalty\z@-opponents-with-minimax-driven-ai\protect\penalty\z@-part-1-Îś-Îš-pruning-cc1df850e5bd • https://gsurma.medium.com/hex-creating-intelligent\protect\penalty\z@-adversaries-part-2-heuristics\protect\penalty\z@-dijkstras-algorithm-597e4dcacf93 • https://www.labri.fr/perso/renault/working/teaching/projets/2019-20-S6-C-Hex.php • https://www.lamsade.dauphine.fr/~cazenave/papers/hex-ria.pdf • https://hal.science/hal-02328750/document
Stratégie gagnante	• https://math.univ-lyon1.fr/irem/IMG/pdf/Hex.pdf • https://en.wikipedia.org/wiki/Hex_(board_game) • https://en.wikipedia.org/wiki/Zermelo's_theorem_(game_theory) • https://en.wikipedia.org/wiki/Game_complexity
Awalé	• https://fondem.ong/awale-regles-du-jeu/ • https://www.myriad-online.com/resources/docs/awale/francais/strategy.htm • https://en.wikipedia.org/wiki/Awale • https://medium.com/@mol02office/implÃľmentation-dune-ia-pour-l-awalÃľ-partie-1\protect\penalty\z@-b298328d5e14
Minimax	• https://en.wikipedia.org/wiki/Minimax • https://en.wikipedia.org/wiki/AlphaâĂŞbeta_pruning • https://fr.wikipedia.org/wiki/ÃĽlagage_alpha-bÃłta

9 Annexe

9.1 Cahier des charges

9.1.1 Objectifs du projet

Pour chaque jeu, nous devons implémenter les règles officielles afin d'assurer une cohérence entre tous les joueurs. De plus, nous voulons fournir une interface graphique agréable pour l'utilisateur. Cela comprend donc une page d'accueil intuitive ainsi que des plateaux de jeux faciles d'utilisation.

9.1.2 Description des jeux

- Hex :

  Le Hex est un jeu de stratégie à deux joueurs. Il se compose d'un plateau, de pions bleus et de pions rouges. Le plateau du jeu du Hex est composé de cases hexagonales formant un losange. La taille du plateau peut varier, mais est généralement de 11×11. Deux côtés opposés du losange sont bleus, les deux autres sont rouges.

  Le joueur bleu commence. Les joueurs jouent chacun à leur tour. À chaque tour, un joueur place un pion de sa couleur sur une case libre du plateau. Le premier joueur qui réussit à relier ses deux bords par un chemin de pions contigus de sa couleur a gagné. Il ne peut y avoir qu'un pion par case. Les pions posés le sont définitivement, ils ne peuvent être ni retirés, ni déplacés.

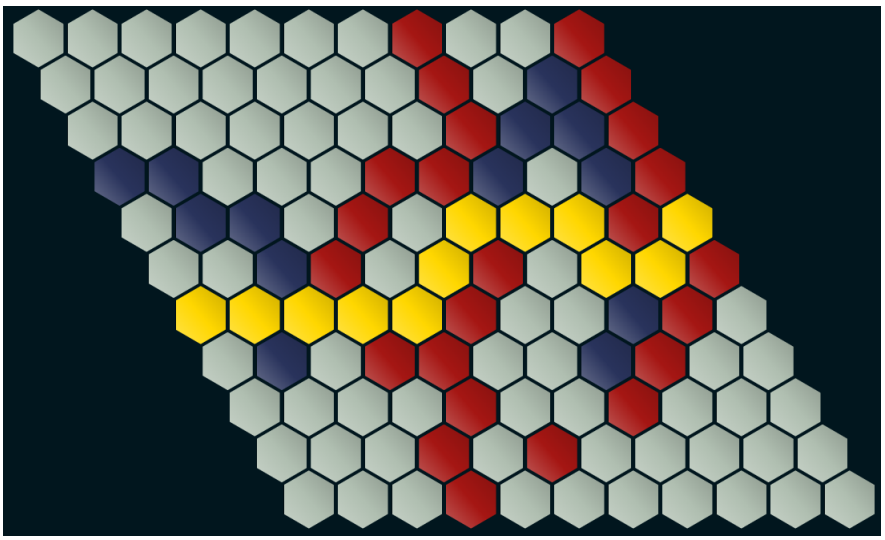

FIGURE 11 – Ici bleu a gagné, ses deux bords sont reliés.

- Awalé :

  L'Awalé est un jeu de stratégie à deux joueurs. Il se compose d'un plateau et de graines. Au début de la partie, 4 graines se situent dans chaque case du plateau. Les joueurs jouent à tour de rôle. À chaque tour, le joueur prend toutes les graines d'un des trous de son camp puis il les égrène une par une dans toutes les cases qui suivent sur sa rangée puis sur celle de son adversaire suivant le sens de rotation (une graine dans chaque trou après celui où il a récupéré les graines).

  Le joueur « capture » des graines lorsque la dernière case où il pose une graine est une case du camp adversaire et s'il contient 2 ou 3 graines en comptant la nouvelle (elle contenait 1 ou 2 graines avant). Le joueur prend alors les graines de cette case (2 ou 3), puis il prend également les graines de la case précédente si celle-ci répond aux mêmes conditions : être une case du camp adverse et contenir 2 ou 3 graines. Il continue ainsi à prendre les graines des cases antérieures tant que celles-ci répondent aux conditions.

La première façon qu'une partie s'achève est lorsqu'un joueur n'a plus de graines dans son camp alors que c'est à lui de jouer et que son adversaire n'est plus en mesure de lui en apporter une selon la règle de « l'obligation de nourrir l'adversaire ». Dans ce cas, son adversaire gagne toutes les graines restantes. C'est la fin par famine.
La deuxième façon qu'une partie s'achève est lorsqu'il reste trop peu de graines pour qu'aucune prise ne soit désormais possible (en pratique 2 ou 3). Chaque joueur récupère la ou les graines restantes de son camp. C'est la fin par indétermination.

FIGURE 12 – plateau du Awalé

Fonctionnalités de l'application

- Possibilité de jouer en joueur contre joueur sur les deux jeux.
- Possibilité de jouer en joueur contre IA avec la couleur de notre choix sur les deux jeux.
- Possibilité d'observer une partie entre deux IA sur les deux jeux.
- Possibilité d'annuler son dernier coup sur les deux jeux.
- Choisir la taille du plateau du Hex.
- Divers boutons pour naviguer à travers l'application.
- Possibilité de réinitialiser le plateau des deux jeux lorsqu'on le souhaite.

Interface utilisateur

L'application s'ouvrira sur une page d'accueil sur laquelle nous pourrons choisir le jeu auquel nous souhaitons jouer. Pour chaque jeu, nous pourrons trouver une page home nous permettant de sélectionner

le mode de jeu auquel nous voulons jouer. Nous retrouverons alors pour les deux jeux les modes JcJ,JcIA et IAvIA.

De nombreux boutons de couleur bleue seront mis en place afin de naviguer entre toutes les pages de l'application. D'autres boutons plus petits seront disponibles. Ils permettront pendant les diverses parties de réinitialiser le plateau de jeu ou encore de défaire notre dernier coup.

9.2 HexGame et stratégie gagnante

Au Hex, pour toutes les tailles de plateaux il existe une stratégie gagnante théorique pour le joueur qui commence. Cependant, celle-ci n'est pas connue pour la plupart des tailles de plateaux. En effet, celle-ci demande une connaissance totale de toutes les parties possibles, ce qui n'est pas calculable en temps réaliste.

9.2.1 Stratégies gagnantes et arbre du jeu

Modélisons le jeu du Hex à l'aide d'un arbre représentant toutes les parties possibles. L'arbre commence avec la position initiale, un plateau vide. De cette racine partent autant de branches qu'il n'y a de possibilités pour le premier coup du premier joueur. On réitère cette action pour chaque nœud jusqu'à tomber sur des positions gagnantes pour l'un des deux joueurs.

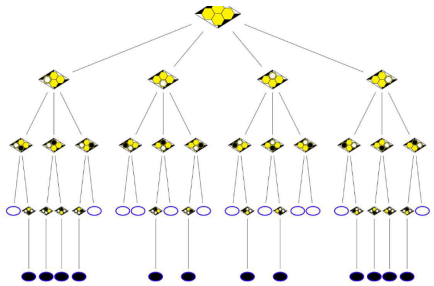

FIGURE 13 – Exemple pour un plateau 2×2

L'arbre va nous aider à nous convaincre qu'il y a bien une stratégie gagnante pour l'un des deux joueurs. Le principe consiste à colorier tous les nœuds de l'arbre en blanc ou en noir, chaque nœud colorié correspondant à une position à partir de laquelle le joueur de la couleur correspondante possède une stratégie gagnante. Nous commençons par colorier en blanc les feuilles représentant une fin de partie gagnée par les blancs, et en noir celles correspondant à la victoire du joueur noir. Le reste du coloriage se fait progressivement. À un moment donné, on veut attribuer une couleur à un nœud qui n'en a pas encore, mais dont toutes les branches descendantes mènent à des nœuds déjà coloriés. Supposons que ce nœud représente une position à partir de laquelle c'est aux blancs de jouer. Si au moins l'une des branches issues du nœud mène à un nœud blanc, alors on colorie le nœud en blanc. Dans le cas contraire, c'est-à-dire si toutes les branches mènent à des nœuds coloriés en noir, on le colorie en noir. On procède de façon symétrique si c'est aux noirs de jouer. De cette manière nous sommes assurés de remplir l'arbre de nos couleurs jusqu'à la racine. Ainsi, la racine possède une stratégie gagnante.

9.2.2 Remarques

- On voit que pour un plateau de dimension 2×2, l'arbre de tous les possibles possède déjà 24 feuilles et 52 branches. Il est calculé que pour un plateau de taille 11×11, le nombre de feuilles de l'arbre est approximativement 10^{98}, et le nombre de positions totales que peut prendre le jeu est de 2.4×10^{56}. Donc en pratique, utiliser cette technique pour trouver la stratégie gagnante n'est pas possible.
- Il est aussi possible d'appliquer ce genre de raisonnement à d'autres jeux combinatoires abstraits pour essayer de prouver l'existence d'une éventuelle stratégie gagnante.
- Enfin, on remarque que cette façon de faire ressemble à l'algorithme *MinMax* s'il avait une profondeur de recherche infinie.

9.2.3 Le premier joueur gagne à tous les coups

On a vu qu'il existe une stratégie gagnante, mais on ne sait pas quel joueur est le gagnant. Dans ce paragraphe nous allons prouver que le joueur 1, s'il joue parfaitement, gagne à tous les coups.

Puisque les matchs nuls sont impossibles, nous pouvons conclure que soit le premier, soit le deuxième joueur possède une stratégie gagnante. Supposons maintenant que le deuxième joueur ait une stratégie gagnante.

Le premier joueur adopte alors la stratégie suivante : il effectue un mouvement arbitraire. Ensuite, il joue en utilisant la supposée stratégie gagnante du deuxième joueur mentionnée ci-dessus. Si, en jouant cette stratégie, il doit jouer sur la case où un mouvement arbitraire a été fait, il effectue un autre mouvement arbitraire. De cette manière, il suit la stratégie gagnante tout en ayant toujours une pièce supplémentaire sur le plateau.

Cette pièce supplémentaire ne peut pas interférer avec l'imitation par le premier joueur de la stratégie gagnante. En effet, une pièce supplémentaire pour le joueur 1 n'est jamais un désavantage. Par conséquent, le premier joueur peut gagner.

Comme nous avons maintenant contredit notre hypothèse selon laquelle le deuxième joueur aurait une stratégie gagnante, nous concluons qu'il n'y a pas de stratégie gagnante pour le deuxième joueur. Par conséquent, il existe une stratégie gagnante pour le premier joueur.

9.3 Diagramme de Gantt

Comme indiqué précédemment, nous n'avons pas utilisé de diagramme de Gantt pour la gestion du projet, mais nous avons tout de même réalisé un planning prévisionnel des tâches à effectuer. Ce planning a été réalisé en début de projet, et a été mis à jour régulièrement pour refléter l'avancement du projet. Un exemple de diagramme de Gantt est présenté ci-dessous :

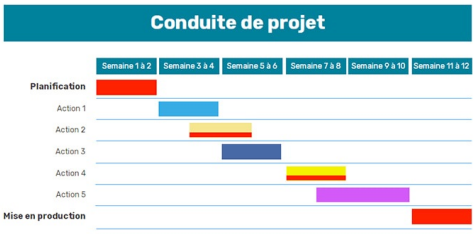

FIGURE 14 – Diagramme de Gantt