Rapport Avalam

BUT3-IA-Jeux - MATH 5.1 et DEV 5.5

IUT UPEC Sénart-Fontainebleau, site Fontainebleau
5 février 2026

Élèves :
Hugo Raban
Adrien Dick
Aurélien Amary
Patrick Felix-Vimalaratnam

Table des matières

1 Présentation du jeu Avalam

Avalam est un jeu de plateau abstrait à deux joueurs. Le plateau est constitué de cases dont certaines sont des “trous” inaccessibles et les autres accueillent des pions. Chaque joueur dispose au départ du même nombre de pions, disposés en tours de hauteur 1 réparties sur le plateau. Un coup consiste à déplacer une pile entière de pions (une tour) sur une case voisine déjà occupée, dans une des huit directions possibles (horizontal, vertical ou diagonal). On ne peut jamais poser de pion sur un trou inoccupé : les cases nulles restent définitivement vides. On ne peut pas non plus déplacer seulement une partie d’une tour : on déplace toujours la pile complète.

À chaque fusion, la nouvelle hauteur de la tour obtenue ne doit jamais dépasser cinq pions. Une tour de hauteur comprise entre un et cinq pions vaut un point pour le joueur dont la couleur est au sommet de la tour. La partie se poursuit tant qu’au moins un coup est possible ; dès qu’aucun déplacement légal n’existe, la partie est terminée. On compte alors le nombre de tours contrôlées par chaque joueur (c’est-à-dire le nombre de sommets de tours à sa couleur), sans tenir compte du nombre de pions à l’intérieur de chaque tour. Le joueur ayant le plus de tours à sa couleur remporte la partie, l’égalité donnant un match nul.

2 Présentation générale du projet

2.1 Contexte et objectifs

Le projet a pour objectif d’implémenter le jeu Avalam en suivant l’API fournie (GameAPI), puis de développer plusieurs joueurs artificiels capables de s’affronter automatiquement. L’enjeu est double : d’une part, modéliser fidèlement les règles d’Avalam et garantir la cohérence du moteur (vérification de la légalité des coups, application des mouvements, détection de fin de partie, calcul du résultat) ; d’autre part, concevoir des bots efficaces basés sur des algorithmes de recherche comme minimax avec élagage alpha-bêta et des fonctions d’évaluation adaptées à la structure du jeu (nombre et hauteur des tours, stabilité des positions, etc.).

L’architecture du projet est organisée autour de trois axes principaux. Le premier est le moteur de jeu, qui modélise le plateau, les tours, les coups et les règles d’Avalam via la classe AvalamBoard et ses auxiliaires. Le second est l’ensemble des bots, qui héritent de la classe abstraite AbstractGamePlayer et utilisent l’API IBoard pour explorer l’arbre des coups possibles, en appelant safeCopy, doPly et iterator. Le troisième est l’interface utilisateur, construite avec Swing, qui permet à un joueur humain de lancer des parties dans différents modes (joueur contre joueur, joueur contre bot, arène de bots) et de visualiser l’évolution du plateau.

2.2 Fonctionnalités réalisées et manquantes

Sur le plan fonctionnel, le moteur Avalam est complet. La classe AvalamBoard implémente l'interface IBoard via l'héritage de AbstractBoard, gère une grille 9×9 de tours et de trous, vérifie la légalité des coups, applique ces coups, détecte la fin de partie et calcule le résultat final en comptant les tours contrôlées par chaque joueur. Le jeu respecte les règles principales : impossibilité de jouer sur un trou vide, déplacement de toute la pile source vers une case voisine occupée, interdiction de dépasser la hauteur maximale de cinq pions, et obligation de jouer tant qu'un coup est disponible.

Plusieurs bots ont été développés. IdiotBot joue un coup purement aléatoire et sert de référence minimaliste. AlphaBetaBot implémente l'algorithme minimax avec élagage alpha-bêta et une fonction d'évaluation simple basée sur la différence du nombre de tours entre les deux joueurs. DivineBot reprend la même structure de recherche mais introduit une fonction d'évaluation plus sophistiquée qui pondère les tours en fonction de leur hauteur, ce qui améliore la qualité des décisions. Un mode arène (ArenaGame et ArenaWindow) permet de faire s'affronter automatiquement des bots, en suivant l'API AbstractGame, et l'interface graphique Avalam (AvalamWindow) offre plusieurs modes de jeu au lancement, y compris le mode "joueur vs bot divin" qui est pleinement fonctionnel.

Certaines fonctionnalités ont été abandonné. Par exemple, une approche par Monte Carlo pour la fonction d'évaluation, c'est-à-dire basée sur des simulations aléatoires de fins de partie, n'a pas été mise en œuvre, malgré une réflexion conceptuelle sur le sujet. Nous avons ajouté des tests (tests unitaires ciblant le moteur et les bots, ainsi que des tests de scénarios de parties), mais la couverture n'est pas encore exhaustive sur l'ensemble du code. Enfin, de nombreux tests manuels et des matchs en arène ont été effectués pour valider le comportement du moteur, des bots et de l'interface.

3 Architecture technique et logique du moteur Avalam

3.1 Vue d'ensemble de l'architecture

Le projet suit une architecture en couches séparant le moteur de jeu, les bots et l'interface graphique. Le diagramme de classes ci-dessous illustre cette organisation :

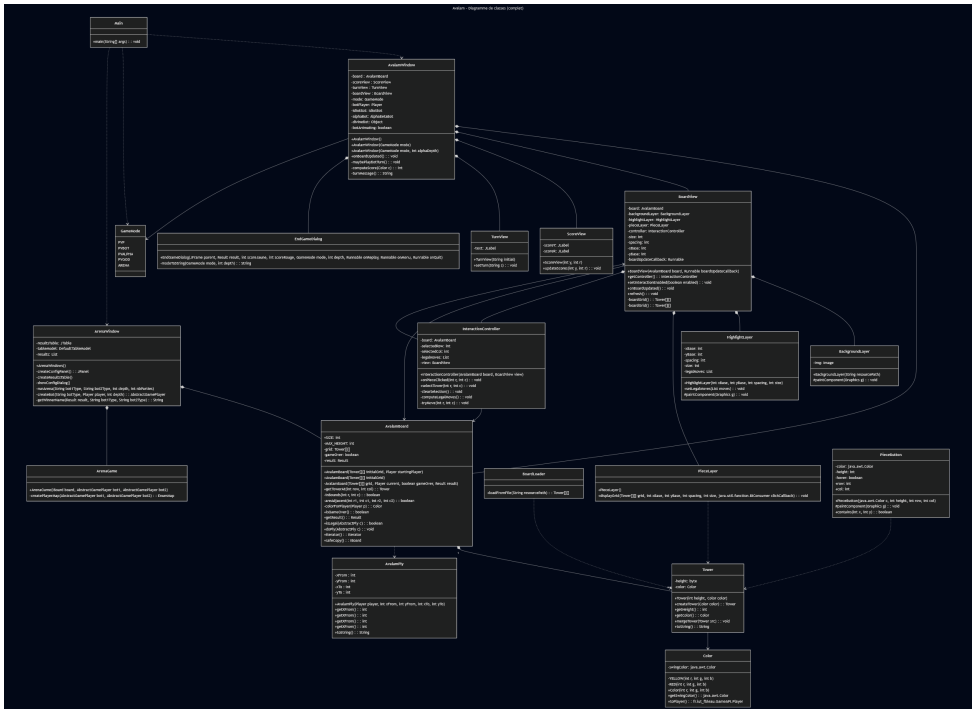

FIGURE 1 – Diagramme de classe sans l'API

Le diagramme montre trois grandes zones fonctionnelles. La partie supérieure gauche regroupe les abstractions de l'API (AbstractBoard, AbstractGame, AbstractGamePlayer, ainsi que les interfaces IBoard, IGame, IPly, IPlayer). Ces classes définissent le contrat que doivent respecter les implémentations concrètes. Au centre, on trouve les classes spécifiques à Avalam : AvalamBoard qui étend AbstractBoard et implémente IBoard, AvalamGame qui étend AbstractGame, ainsi que les classes utilitaires Cell, Coords et Move qui modélisent respectivement une case du plateau, des coordonnées et un coup. À droite, les trois bots (IdiotBot, AlphaBetaBot, DivineBot) héritent tous de AbstractGamePlayer et implémentent la méthode giveYourMove(IBoard), avec des stratégies de recherche différentes. En bas du diagramme, l'interface graphique est organisée autour de AvalamWindow et ArenaWindow, qui utilisent BoardView pour le rendu du plateau, et de contrôleurs (AbstractGameController, ArenaGameController) qui orchestrent l'interaction entre le moteur et l'interface.

Cette séparation permet une évolution indépendante des composants : on peut ajouter un nouveau bot sans modifier le moteur, ou changer l'interface sans toucher à la logique de jeu. Les associations montrent que AbstractGame utilise un IBoard et des AbstractGamePlayer, que les bots reçoivent un IBoard via giveYourMove, et que l'interface se connecte au moteur via les contrôleurs.

3.2 Représentation du plateau et des tours

Le moteur Avalam repose sur la classe AvalamBoard, qui étend AbstractBoard. Le plateau est modélisé par un tableau bidimensionnel Tower[][] grid de dimension 9×9, où chaque entrée représente une case du plateau. Une case peut contenir soit null, qui symbolise un trou sur lequel aucun pion ne peut être placé, soit une instance de la classe Tower. Une tour encapsule deux informations essentielles : la hauteur de la pile de pions, et la couleur du pion situé au sommet (par exemple jaune pour COLOR1 / PLAYER1, rouge pour COLOR2 / PLAYER2). Cette modélisation permet de retrouver rapidement le propriétaire d'une tour, ce qui est crucial pour la détermination du résultat final et pour les fonctions d'évaluation des bots.

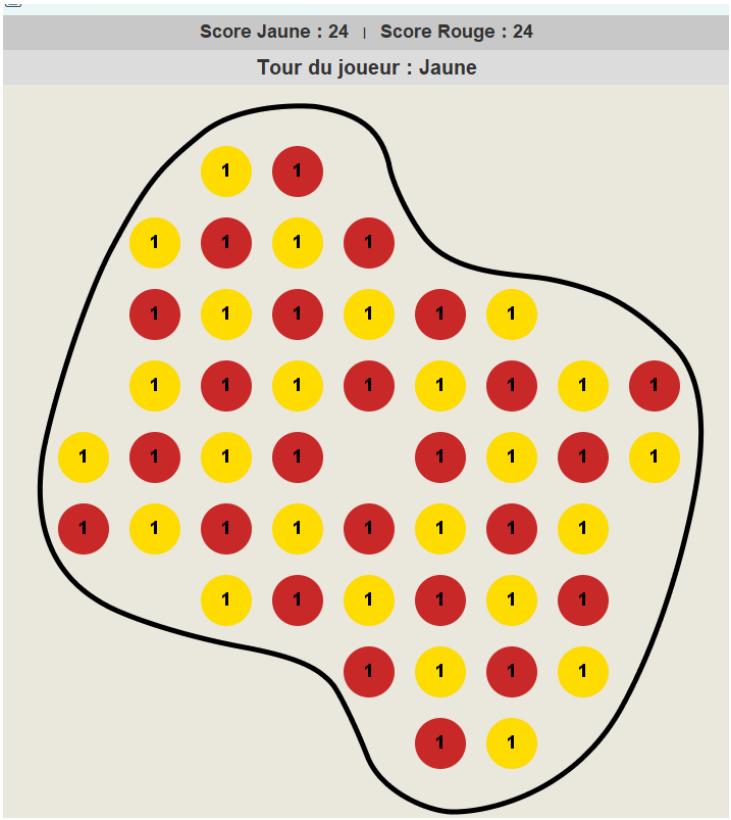

FIGURE 2 – Map d'une partie

Lors de la construction d'un AvalamBoard, la grille initiale est reçue en paramètre sous forme de matrice de tours et de null. Le constructeur recopie cette grille dans une nouvelle matrice interne afin de préserver l'encapsulation et d'éviter que des modifications extérieures n'altèrent l'état de la partie. La classe mémorise aussi le joueur courant via l'appel au constructeur de AbstractBoard, et utilise deux attributs gameOver et result pour mémoriser respectivement si la partie est terminée et quel en est le résultat, de manière à éviter des recalculs inutiles.

3.3 Vérification des coups : méthode isLegal

La validation d'un coup est centralisée dans la méthode isLegal(AbstractPly c). Le paramètre est typé de façon générique par l'API, il faut donc commencer par vérifier qu'il s'agit bien d'un coup propre à Avalam, via instanceof AvalamPly, puis effectuer un cast. On extrait ensuite les coordonnées de départ et d'arrivée du coup à l'aide de méthodes comme getXFrom, getYFrom, getXTo, getYTo. Une première vérification porte sur l'appartenance des coordonnées au plateau : une méthode utilitaire inBounds(int r, int c) renvoie vrai si les indices sont compris entre zéro et SIZE - 1. Si la source ou la destination sont hors bornes, le coup est jugé illégal.

Le moteur interdit ensuite les "coups nuls" en s'assurant que la source et la destination ne sont pas la même case. Il récupère les tours source et destination dans la grille : si la source est null, il n'y a aucune pile à déplacer ; si la destination est null, on tente de poser une tour sur un trou inoccupé, ce qui est interdit dans les règles du jeu. Dans ces cas, le coup est immédiatement rejeté. La méthode vérifie également que la tour source appartient bien au joueur courant. Pour cela, une méthode colorForPlayer(Player p) associe un joueur de l'API à une couleur interne, et on compare cette couleur à celle de la tour source.

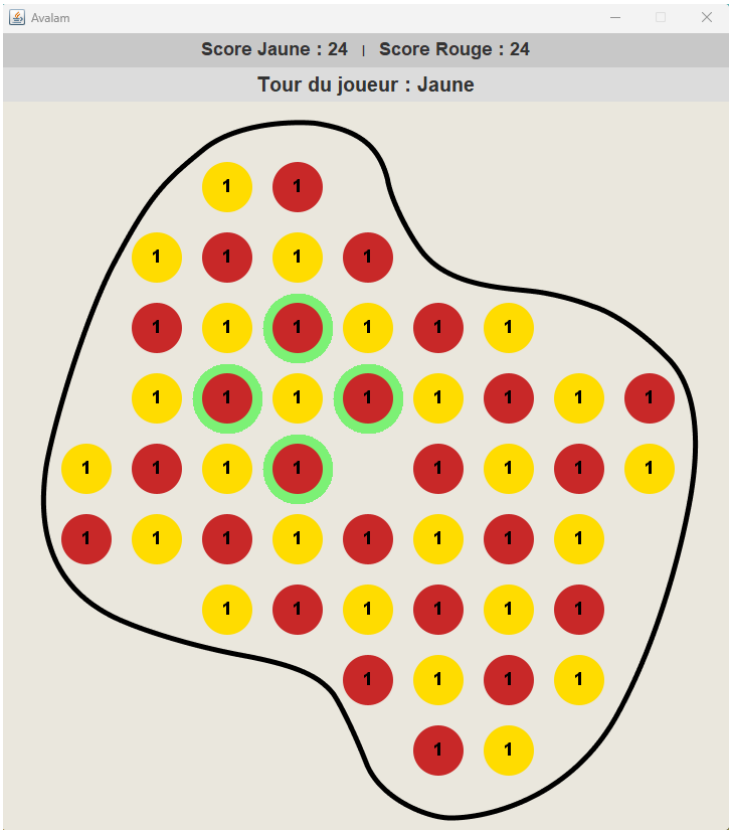

FIGURE 3 – Un coup possible pendant une partie

La notion de voisinage est gérée par la méthode areAdjacent(int r1, int c1, int r2, int c2). Celle-ci calcule les différences de lignes et de colonnes entre la source et la destination et vérifie que ces différences sont au plus égales à un en valeur absolue, tout en excluant le cas où elles seraient toutes deux nulles. Cela correspond à un voisinage en huit directions (horizontal, vertical et diagonal). Si la destination n'est pas une case voisine, le coup est illégal. Enfin, deux contraintes supplémentaires sont prises en compte : la couleur de la destination doit être différente de celle de la source (règle choisie dans notre projet pour fixer un comportement précis), et la somme des hauteurs des deux tours ne doit pas excéder la constante MAX_HEIGHT fixée à cinq. Si une seule de ces conditions échoue, isLegal renvoie faux, sinon le coup est jugé légal.

3.4 Application des coups : méthode doPly

La méthode doPly(AbstractPly c) applique effectivement un coup sur le plateau. Par sécurité, elle commence par rappeler isLegal : si le coup n'est pas reconnu comme valide, une exception est levée, ce qui empêche l'appelant de corrompre l'état du plateau. Une fois la légalité confirmée, le moteur extrait les coordonnées source et destination, puis les tours correspondantes. La fusion des tours s'effectue en appelant une méthode mergeTower sur la tour de destination, en lui passant la tour source. Cette méthode interne à Tower augmente la hauteur de la tour destination et met à jour la couleur du sommet conformément au mouvement (en pratique, c'est la pile source qui vient se placer au-dessus de la pile destination, ou inversement selon la convention choisie, mais dans tous les cas on connaît la nouvelle couleur dominante).

Après la fusion, la case source de la grille est mise à null, car elle devient un trou. Puis doPly appelle super.doPly(c) pour laisser la superclasse AbstractBoard assurer la gestion du changement de joueur courant et, éventuellement, de l'historique des coups si l'API le prévoit. Enfin, gameOver est remis à faux et result à null, ce qui oblige les prochains appels à isGameOver et getResult à recalculer ces valeurs à partir du nouvel état du plateau.

3.5 Détection de fin de partie et calcul du résultat

La détection de la fin de partie est effectuée par la méthode isGameOver(). Si gameOver est déjà à vrai, la méthode renvoie immédiatement vrai, ce qui évite un calcul coûteux. Dans le cas contraire, elle obtient un itérateur sur les coups possibles via iterator(). Si l'itérateur fournit au moins un coup, la partie n'est pas terminée et la méthode renvoie faux. Si au contraire aucun coup n'est disponible, la méthode fixe gameOver à vrai et renvoie vrai.

Le calcul du résultat est placé dans getResult(). Si la partie n'est pas terminée selon isGameOver(), la méthode renvoie null pour indiquer l'absence de résultat. Si un résultat mémorisé existe déjà (attribut result non nul), la méthode renvoie cette valeur. Sinon, elle parcourt l'ensemble du plateau, en comptant le nombre de tours dont la couleur du sommet est jaune et le nombre de tours dont la couleur du sommet est rouge. La comparaison de ces deux totaux permet de déterminer qui l'emporte. Conformément à l'API, Result.WIN, Result.LOSS ou Result.DRAW sont renvoyés du point de vue de PLAYER1, et cette valeur est mémorisée dans result pour ne pas être recalculée ultérieurement.

3.6 Génération des coups et copie sûre du plateau

La méthode iterator() de AvalamBoard génère l'ensemble des coups légaux pour le joueur courant. L'algorithme parcourt chaque case du plateau, et pour chacune, parcourt toutes les paires de déplacements possibles en ligne et en colonne dans l'intervalle [-1, 1], en excluant la paire (0, 0). Pour chaque combinaison, une instance d'AvalamPly est créée, avec comme joueur le joueur courant, et les coordonnées source et destination fixées par la double boucle. Le moteur appelle ensuite isLegal sur ce coup. Si isLegal renvoie vrai, le coup est ajouté à une liste de coups. À la fin du parcours de la grille, la méthode renvoie un itérateur sur cette liste. Ce mécanisme, bien que naïf, est suffisamment efficace pour la taille du plateau et simplifie la logique des bots, qui n'ont plus qu'à itérer sur ce flux de mouvements.

La méthode safeCopy() renvoie une copie sûre du plateau, indispensable pour les algorithmes de recherche. Elle crée une nouvelle matrice de tours, puis pour chaque case du plateau, si une tour est présente, elle instancie une nouvelle tour avec la même hauteur et la même couleur ; si la case est un trou, elle recopie null. À la fin, safeCopy instancie un nouvel AvalamBoard avec cette nouvelle grille et le même joueur courant. Cette copie est "profonde" au niveau des tours, ce qui garantit qu'un bot peut modifier librement la copie, par exemple en appelant doPly, sans interférer avec l'état réel géré par le moteur central.

4 Le meilleur bot : DivineBot et son fonctionnement

4.1 Positionnement de DivineBot parmi les autres bots

Dans notre projet, le bot le plus abouti et le plus performant est DivineBot. Il repose, comme AlphaBetaBot, sur l'algorithme minimax avec élagage alpha-bêta, mais il se distingue par une fonction d'évaluation qui prend en compte la hauteur des tours et la proximité des tours adverses, contrairement à AlphaBetaBot qui ne regarde que le nombre de tours contrôlées. Alors que IdiotBot se contente de jouer un coup aléatoire parmi les mouvements légaux. En pratique, dans le mode arène, c'est le bot qui obtient les meilleurs résultats : il surclasse nettement IdiotBot, et se montre généralement supérieur à AlphaBetaBot, surtout avec une profondeur de recherche suffisante.

4.2 Interaction avec l'API et structure générale

DivineBot étend la classe abstraite AbstractGamePlayer, ce qui l'oblige à implémenter la méthode giveYourMove(IBoard board). Cette méthode est le point d'entrée à chaque tour : le moteur de jeu lui fournit un plateau board qui est une copie sûre de l'état courant de la partie, via safeCopy(). Cela signifie que le bot peut librement simuler des coups sur cet objet sans craindre de modifier l'état réel de la partie géré par AbstractGame et AvalamBoard.

La structure interne de DivineBot repose sur quelques attributs clés :

— 'me', qui enregistre le joueur que contrôle le bot (PLAYER1 ou PLAYER2) ;
— 'maxDepth', qui fixe la profondeur maximale de recherche dans l'arbre de jeu ;
— un générateur Random, qui sert à départager plusieurs coups jugés équivalents par la fonction d'évaluation.

Dans giveYourMove, la première étape consiste à vérifier si la partie est déjà terminée (board.isGameOver()) ou s'il n'existe aucun coup légal (liste de mouvements vide). Dans ce cas, le bot renvoie null pour signaler qu'il ne peut pas jouer. Sinon, le bot appelle une méthode interne qui construit une liste de tous les coups possibles, en parcourant l'itérateur fourni par board.iterator(). Une fois cette liste obtenue, DivineBot va, pour chaque coup, simuler son application sur une copie du plateau (IBoard next = board.safeCopy() ; next.doPly(m) ;) et appeler la fonction récursive alphaBeta(next, maxDepth - 1, alpha, beta) pour estimer la qualité de la position résultante.

Le choix final du coup repose ensuite sur la valeur renvoyée par alphaBeta. Si c'est au tour du bot de jouer (le plateau indique que board.getCurrentPlayer() == me), celui-ci se comporte comme un joueur Max : il cherche à maximiser la valeur. Si c'est au tour de l'adversaire, il se comporte comme un joueur Min : il anticipe que l'adversaire essaiera de minimiser cette même valeur. DivineBot conserve l'ensemble des coups qui atteignent la meilleure valeur trouvée, puis en choisit un au hasard parmi ces meilleurs coups ex æquo. Ce tirage aléatoire permet d'éviter que le bot joue toujours les mêmes coups dans des positions symétriques, et introduit un minimum de diversité dans son style de jeu.

4.3 L'algorithme alpha-bêta dans DivineBot

La méthode alphaBeta(IBoard board, int depth, int alpha, int beta) est le cœur algorithmique de DivineBot. Elle reprend la structure classique du minimax avec élagage alpha-bêta. À chaque appel, elle commence par vérifier deux conditions d'arrêt :

- Si la partie est terminée (board.isGameOver()), la fonction appelle terminalValue(board). Celle-ci interroge board.getResult() pour savoir si la position est une victoire, une défaite ou un nul du point de vue de PLAYER1. En fonction du joueur contrôlé par le bot (me), la méthode convertit ce résultat en une valeur entière très positive (par exemple +10000) en cas de victoire du bot, très négative (10000) en cas de défaite, et nulle en cas de match nul. Ces valeurs extrêmes garantissent que les positions gagnantes ou perdantes sont prioritaires par rapport à toutes les évaluations heuristiques intermédiaires.

- Si la profondeur maximale est atteinte (depth == 0), la fonction renvoie directement la valeur renvoyée par evaluate(board), c'est-à-dire la fonction d'évaluation heuristique propre à DivineBot. À ce stade, la position n'est pas forcément terminale, mais on arrête la recherche pour limiter le temps de calcul.

- Si aucune de ces deux conditions n'est remplie, alphaBeta doit explorer les coups possibles. Elle commence par déterminer si c'est au tour du bot (board.getCurrentPlayer() == me) ou de l'adversaire. Dans le premier cas, la fonction se place en mode Max : elle initialise une valeur courante alpha (borne inférieure) et cherche à la maximiser. Dans le second cas, elle se place en mode Min : elle manipule principalement beta (borne supérieure) et cherche à la minimiser.

Dans chaque cas, l'algorithme parcourt la liste des coups possibles, obtenue comme dans giveYourMove en parcourant l'itérateur renvoyé par board.iterator(). Pour chaque coup, il crée une copie du plateau (safeCopy), applique le coup (doPly) et appelle récursivement alphaBeta avec une profondeur décrémentée. La valeur obtenue est comparée avec la meilleure valeur obtenue jusqu'ici, et les bornes alpha et beta sont mises à jour en conséquence. Si, à un moment donné, alpha devient supérieur ou égal à beta, la fonction effectue une coupure : elle sait que l'adversaire ne choisira jamais une suite de coups qui aboutit à une position aussi mauvaise pour lui, et il est donc inutile d'explorer la fin de la branche actuelle. Cette coupure permet de réduire considérablement le nombre de positions évaluées, surtout quand la profondeur est importante.

Le résultat final de alphaBeta est soit la meilleure valeur atteinte par le joueur Max, soit la meilleure valeur atteinte par le joueur Min, en tenant compte de ces coupures. C'est cette valeur qui alimente la décision de giveYourMove.

4.4 Fonction d'évaluation : exploiter la hauteur des tours

Les fonctions d'évaluation permettent de quantifier la qualité d'une position sans explorer toutes les suites possibles. Chaque bot utilise une approche différente, reflétant des niveaux de sophistication croissants.

IdiotBot ne possède pas de fonction d'évaluation. Sa méthode ‘giveYourMove()‘ sélectionne simplement un coup aléatoire parmi les coups légaux disponibles, sans aucune analyse de la position.

AlphaBetaBot utilise une évaluation simple : la méthode ‘evaluate()‘ compte le nombre de tours contrôlées par chaque joueur et retourne la différence (‘botTowers - oppTowers‘). Cette approche est rapide et reflète directement l'objectif du jeu, mais elle ne distingue pas les tours selon leur hauteur ou leur vulnérabilité. Elle peut ainsi conduire à des erreurs stratégiques, comme sacrifier une tour de hauteur 4 contre deux tours de hauteur 1, ce qui semble avantageux selon le comptage mais peut être désavantageux si ces tours sont facilement capturables.

DivineBot possède la fonction d'évaluation la plus élaborée. Elle attribue à chaque tour une valeur pondérée selon plusieurs critères. Les tours de hauteur 5 ou isolées (sans voisins) valent 1000 points car elles représentent des points garantis. Les tours vulnérables (pouvant être capturées au prochain coup) reçoivent une pénalité de -200 points. Pour les autres tours, une valorisation progressive est appliquée selon la hauteur : 400 points pour la hauteur 4, 150 pour la hauteur 3, 60 pour la hauteur 2, et 10 pour la hauteur 1. Cette évaluation s'appuie sur deux fonctions auxiliaires : ‘isIsolated()‘ pour détecter les tours isolées, et ‘isVulnerable()‘ pour identifier les tours menacées.

Cette évaluation multi-critères permet à DivineBot de prendre des décisions plus nuancées que AlphaBetaBot. Elle pourrait encore être améliorée en prenant en compte la position des tours sur le plateau ou la mobilité, mais elle représente déjà un net progrès par rapport à l'approche basique d'AlphaBetaBot.

4.5 Comparaison avec IdiotBot et AlphaBetaBot

Du point de vue de la logique, IdiotBot, AlphaBetaBot et DivineBot illustrent trois niveaux de sophistication croissante. IdiotBot s'appuie uniquement sur la génération de coups du moteur (board.iterator()) et sur un choix aléatoire, sans aucune évaluation ni recherche de profondeur. Il sert surtout de bot de test : si un bot plus avancé perd contre IdiotBot, c'est souvent qu'il y a un bug dans la logique.

AlphaBetaBot utilise déjà alpha-bêta, ce qui l'amène à simuler plusieurs coups d'avance et à choisir les lignes de jeu les plus favorables selon sa fonction d'évaluation. Toutefois, comme cette fonction ne regarde que le nombre de tours contrôlées, elle manque de finesse : le bot peut, par exemple, sacrifier une tour de hauteur 4 contre deux tours de hauteur 1, ce qui est parfois stratégiquement discutable dans Avalam.

DivineBot, lui, corrige ce défaut grâce à son évaluation qui valorise les tours de hauteur 5. Il voit immédiatement la différence entre une tour verrouillée et des tours encore modifiables, et cherche à maximiser un avantage structurel durable plutôt qu'un simple compteur de tours. Dans le mode arène, cela se traduit par des victoires fréquentes contre AlphaBetaBot, surtout dans les parties suffisamment longues pour que la gestion des tours de hauteur 5 soit décisive.

4.6 Pistes explorées et abandonnées : Monte Carlo et heuristiques plus fines

Lors de la conception de DivineBot, nous avons envisagé d'aller plus loin que cette heuristique en utilisant une approche de type Monte Carlo. L'idée était la suivante : pour une position donnée, lancer un certain nombre de simulations aléatoires jusqu'à la fin de la partie, en jouant des coups au hasard ou avec un bot simple comme IdiotBot, puis utiliser le taux de victoire du bot comme estimation de la qualité de la position. Ce type d'évaluation est en principe capable de capturer des effets stratégiques complexes, sans qu'il soit nécessaire de les modéliser explicitement.

Cependant, cette approche s'est heurtée à plusieurs limites pratiques. D'abord, chaque évaluation aurait exigé un nombre non négligeable de simulations pour être statistiquement significative, ce qui multiplie le coût en temps de calcul par le nombre de positions évaluées par alpha-bêta. Ensuite, intégrer Monte Carlo avec un algorithme comme alpha-bêta nécessitait de gérer finement le compromis entre profondeur de recherche et nombre de simulations, afin de ne pas dépasser des temps de réponse raisonnables sur une machine de l'IUT. Enfin, le comportement des simulations aléatoires dépend lui-même de la politique de jeu choisie : un adversaire "trop idiot" ou "trop aléatoire" donne des estimations biaisées de la difficulté réelle de la position.

Pour ces raisons, nous avons concentré nos efforts sur une heuristique déterministe mais bien adaptée à Avalam, en l'affinant autour de la hauteur des tours. D'autres heuristiques plus fines ont également été discutées, comme la prise en compte de la position des tours sur le plateau (par exemple en valorisant certaines cases plus stables), ou encore une valorisation progressive de toutes les hauteurs. Cependant, ces améliorations n'ont pas toutes été mises en œuvre faute de temps. La structure actuelle de evaluate dans DivineBot reste néanmoins suffisamment modulaire pour permettre, à l'avenir, d'ajouter de nouveaux critères pondérés et d'améliorer encore la qualité du bot.

L'idée d'une évaluation Monte Carlo était souhaitée, cependant, cette amélioration n'a pas été mise en œuvre pour faute de temps.

5 Interface graphique, mode arène

5.1 Interface Avalam et sélection des modes de jeu

L'interface graphique du jeu est bâtie grâce à Swing, autour de la classe AvalamWindow. Au lancement du programme, la classe Main invoque showModeSelection(), qui affiche un dialogue proposant différents modes : joueur contre joueur, joueur contre IdiotBot, joueur contre AlphaBetaBot, joueur contre DivineBot (le mode fonctionne, même si une ancienne mention "NON IMPLEMENTE" peut encore apparaître dans le menu) et mode arène.

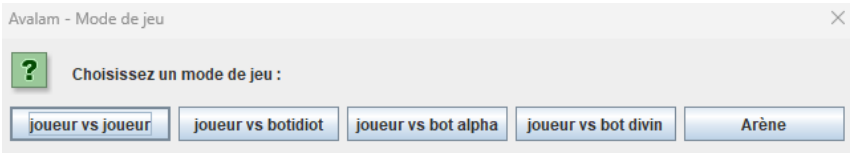

Figure 4 – Menu du choix du mod de la partie

Le choix de l'utilisateur détermine le type de fenêtre créée : AvalamWindow pour les modes impliquant un joueur humain, ArenaWindow pour les matches entièrement automatisés. Dans les modes alpha-bêta et divin, l'interface demande également à l'utilisateur de saisir une profondeur de recherche ; cette profondeur est ensuite passée au constructeur du bot correspondant, ce qui permet d'ajuster le compromis entre force de jeu et temps de calcul. Une fois la fenêtre construite, l'interface se charge d'afficher la grille, les tours et les informations de partie (joueur courant, scores, etc.) en interrogeant régulièrement le moteur pour connaître l'état du plateau. À la fin d'une partie, un écran de fin récapitule le résultat (victoire, défaite ou match nul) et propose de revenir au menu principal ou de relancer une nouvelle partie, ce qui facilite l'enchaînement des tests et des démonstrations.

Figure 5 – Menu de fin de la partie

5.2 Mode arène et orchestration des bots

Le mode arène est géré par la classe ArenaGame, qui étend AbstractGame. Son constructeur reçoit un plateau initial de type IBoard et deux instances de AbstractGamePlayer représentant les bots pour PLAYER1 et PLAYER2. Une méthode privée construit ensuite une EnumMap<Player, AbstractGamePlayer> associant chaque joueur à son bot. Grâce à l'héritage d'AbstractGame, l'arène peut automatiser le déroulement de la partie en alternant les appels à giveYourMove sur chaque bot puis à doPly sur le plateau, jusqu'à ce que isGameOver() retourne vrai. Ce mode est particulièrement utile pour comparer les comportements de différents bots, évaluer empiriquement leurs forces et repérer des faiblesses dans leurs fonctions d'évaluation. La fenêtre ArenaWindow offre une interface pour choisir les bots qui s'affrontent (Idiot, Alpha-Beta, Divin) et, le cas échéant, la profondeur de recherche associée.

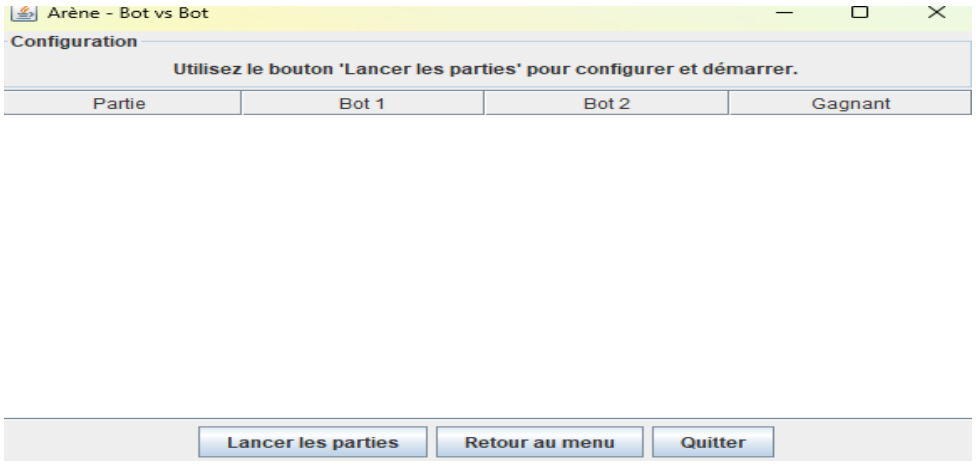

FIGURE 6 – Menu d'arrivée dans l'Arène

Elle sert principalement d'outil de test et de démonstration : en lançant une série de parties bots contre bots, on peut observer comment les différentes stratégies interagissent et si les résultats empiriques correspondent aux attentes théoriques (par exemple, DivineBot qui surclasse IdiotBot sur un grand nombre de parties).

FIGURE 7 – Menu de l'Arène après une simulation

6 Organisation du travail et gestion de projet

Pour l'organisation du travail, nous nous sommes appuyés principalement sur les fonctionnalités collaboratives offertes par notre plateforme de gestion de code (Git et son "Jira" intégré dans Grond). Nous avons structuré le développement autour de branches dédiées et de Merge Requests (MR), ce qui nous a permis de garder une branche principale relativement stable tout en développant de nouvelles fonctionnalités en parallèle. Chaque fonctionnalité importante (par exemple l'implémentation de AvalamBoard, l'ajout d'un nouveau bot, la création du mode arène ou l'intégration d'éléments de l'interface graphique) faisait l'objet d'une branche spécifique, qui n'était fusionnée qu'après revue et validation via une MR.

Le "Jira" intégré (système de tickets de Grond) nous a servi de support pour suivre les tâches et les bugs.

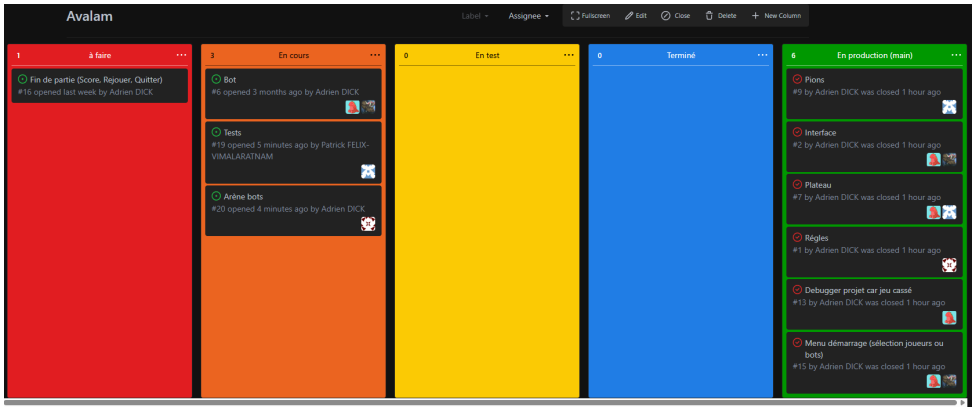

FIGURE 8 – Menu de l'interface des tickets

Nous créions des tickets pour les grandes fonctionnalités (moteur Avalam, bots, interface, javadoc, etc.) mais aussi pour des points plus précis comme la correction d'un comportement illégal dans isLegal, l'ajout d'un critère dans la fonction d'évaluation de DivineBot ou la correction d'un problème d'affichage dans ArenaWindow. Chaque ticket était associé à une branche et, le plus souvent, à une MR, ce qui assurait une bonne traçabilité entre le besoin fonctionnel, les modifications de code et leur intégration.

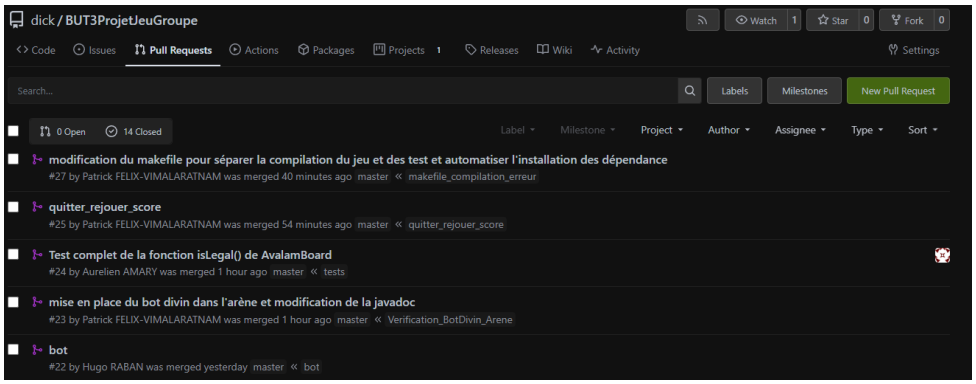

FIGURE 9 – Menu des différentes Merge Request faites lors du projet

Ce mode de travail en branches et MR nous a aidés à éviter les conflits de code trop importants et à maintenir une qualité minimale sur la branche principale. La revue de code lors des MR a également été l'occasion de discuter de certaines décisions techniques (par exemple le détail des règles dans isLegal, la profondeur par défaut des bots ou la structure de la fonction d'évaluation), ce qui a contribué à aligner l'équipe sur les choix d'architecture et à améliorer progressivement la lisibilité du projet.

7 Conclusion individuelle et collective

7.1 Conclusions Individuelles

Patrick Felix-Vimalaratnam :
J'ai trouvé ce projet intéressant car ce projet m'a permis de pouvoir utiliser un système similaire aux tickets Jira que je vois en entreprise et ceux lors d'un projet scolaire. J' ai également dû modifier ma façon de programmer car cette fois je ne pouvais pas le faire en partant sur ma piste comme dans les autres projets dû à l'existence de l'API.

Hugo Raban :
Le projet était fort intéressant, et cela à permis de créer un bot de ses propres mains. De plus, l'usage des tickets permet de mettre des priorités, et de pouvoir s'organiser dans le groupe, afin que tout le monde puisse participer.

Adrien Dick :
Le projet m'a permis de mettre en pratique ce que nous avons appris en cours. J'ai ainsi pu appliquer l'ensemble de ces connaissances. N'ayant jamais réalisé de projet avec une API tel quel auparavant, cette expérience a été un peu plus complexe, mais elle m'a permis de progresser et de m'améliorer grâce à cette découverte.

Aurélien Amary :
Ce projet m'a permis de me familiariser davantage avec les classes abstraites, leurs usages ainsi que l'intérêt qu'elles peuvent apporter dans un tel projet. Comme l'on souligné également d'autres membres, c'était la première fois que nous devions construire un programme modulaire (API) avec une base existante. Bien que ce ne fut pas forcément évident à comprendre au départ, nous nous en sommes bien sortis. Ce projet m'a aussi permis de monter en compétences sur la réalisation de tests avec la bibliothèque Junit. Un parallèle qui fait sens par rapport à Unittest (sous Python) que j'utilise au travail.
Dernièrement, il est vrai que j'étais plus en retrait sur ce projet que sur d'autres, cela m'a cependant apporté une vision différente du travail en équipe et de l'organisation. J'ai participé aux décisions, apporté ma contribution bien sûr mais là où j'ai plutôt l'habitude de m'investir émotionnellement dans un projet, je m'aperçois qu'il est peut-être préférable de prendre un peu de recul, faire davantage confiance, pour le bien du projet (et peut-être aussi de moi-même).

7.2 Conclusion collective

Nous avons atteint l'objectif principal du projet : proposer une implantation complète et fonctionnelle d'Avalam, avec un moteur conforme à l'API, plusieurs bots de niveaux de jeu différents et une interface utilisateur permettant de jouer et d'observer des parties. Nous avons également pris conscience que la conception d'une bonne fonction d'évaluation, même sur un jeu de taille modeste, est un problème non trivial qui demande de combiner intuition sur le jeu et expérimentation pratique. Bien que certaines pistes, comme une évaluation Monte Carlo, n'aient pas pu être menées à terme, le projet nous a permis de nous confronter à des problématiques réelles d'architecture logicielle, d'IA pour les jeux déterministes à deux joueurs et de gestion de projet en équipe.