Implémentation du jeu Hex via la Programmation Orientée Objet et stratégies de décision en C++

1st Jair Anderson Vasquez Torres
Ingénieur Degree Programme STIC
ENSTA Paris
Paris, France
jair-anderson.vasquez@ensta.fr

2nd Santiago Florido Gomez
Ingénieur Degree Programme STIC
ENSTA Paris
Paris, France
santiago.florido@ensta-paris.fr

***Abstract*—Ce projet implémente en C++ un système complet pour le jeu Hex, conçu principalement comme un exercice de Programmation Orientée Objet. La solution organise le domaine du jeu au moyen de classes qui encapsulent le plateau, l'état, les coordonnées et les règles, permettant la génération de coups légaux et la vérification de victoire au moyen d'un parcours BFS sur les connexions des pions. Sur cette base s'intègrent des stratégies de décision (p. ex., Negamax avec hachage et table de transposition) et une évaluation de positions découplée du moteur, qui peut être heuristique ou s'appuyer sur un modèle neuronal de valeur intégré à l'exécutable (exportable en TorchScript), sans lier la conception à une architecture spécifique. De plus, une interface graphique (GUI) en SFML a été développée pour faciliter l'interaction, la visualisation du plateau et les tests du comportement des agents.**

***Index Terms*—Hex game, C++, Object-Oriented Programming, Negamax, Transposition Table, TorchScript, SFML.**

I. Introduction

Dans son espèce, Hex est un jeu de stratégie abstraite pensé pour deux joueurs, c'est aussi un jeu d'information parfaite ce qui implique qu'à tout moment les joueurs connaissent toute l'information du plateau et il n'y a pas d'information cachée pour aucun des deux joueurs et sans hasard qui se joue sur un plateau avec des cellules hexagonales généralement en forme de losange mais dans la version modifiée réalisée avec des cellules de décalage intercalé. Chaque joueur a une paire de côtés opposés du plateau assignés comme objectif , chaque joueur doit à son tour correspondant placer une pièce de sa couleur dans une cellule vide à l'intérieur du plateau.

L'objectif principal du jeu consiste à ce que chaque joueur parvienne à la connexion d'une chaîne connectée de pièces de la même couleur avant que cette connexion soit réalisée par son adversaire le premier à réaliser la connexion gagne la partie , étant donnée la constitution du jeu celui-ci ne peut pas se terminer en match nul il doit toujours exister un joueur gagnant , dans le présent projet on propose un plateau de base de 7 x 7, qui peut être modifié par les joueurs pour avoir un plateau de plus grande dimension.

Dans la configuration proposée, on propose la possibilité de s'affronter comme utilisateur de l'application à un autre utilisateur qui se trouve sur le même dispositif , comme cela fonctionne de manière traditionnelle, le jeu , ou bien de s'affronter à un adversaire virtuel qui peut être un modèle heuristique qui a été défini en utilisant un moteur de règles de base qui expose des opérations et des coups légaux et de vérification de victoire sur lequel se monte un agent Negamax qui utilise une évaluation heuristique en ressemblant au fonctionnement d'un capteur de qualité des mouvements du joueur au tour dans le nœud dans lequel se trouve donnée l'évaluation ou un modèle impulsé par l'implémentation d'un modèle qui utilise la même logique de Negamax , mais qui au lieu d'utiliser une expression heuristique pour déterminer une valeur pour chaque mouvement possible utilise une MLP, qui a été obtenue au moyen de reinforcement learning avec des données de parties en direct entre le modèle et des versions antérieures ou l'heuristique.

On présentera dans ce projet des éléments de sa configuration de base en termes de structure et de fonctionnement, ainsi que la manière dont ont été appliqués les concepts de la programmation orientée objet pour l'obtention du produit final présenté.

II. Structure du projet

Le dépôt est organisé en dossiers principaux afin de séparer clairement le code C++, les scripts et les modules d'auto-jeu. Le schéma suivant résume l'arborescence (sans les dossiers de build).

```
HexProject/
|-- include/
|   |-- core/
|   |-- gnn/
|   `-- ui/
|-- src/
|   |-- core/
|   |-- gnn/
|   |-- ui/
|   `-- cli/
|-- selfplay/
|   |-- gnn/
|   `-- mlp/
`-- scripts/
    |-- models/
    `-- __pycache__/
```

HexProject/

Dossier	Contenu et rôle général
include/	en-têtes C++ (API publique par module)
src/	implémentation C++ (logique, GNN, UI, CLI)
selfplay/	auto-jeu (gnn/mlp) et génération de données
scripts/	scripts Python et modèles entraînés

TABLE I
RÉPARTITION DES DOSSIERS PRINCIPAUX.

```
|-- scripts/
|   |-- models/
|   |   |-- hex_value_ts_mp.pt
|   |   |-- hex_value.pt
|   |   |-- hex_value_ts.pt
|   |   `-- value_mlp_state.pt
|   |-- __pycache__/
|   `-- train_value_mlp_emulate_heuristic.py
|-- include/
|   |-- core/
|   |   |-- Board.hpp
|   |   |-- Cube.hpp
|   |   |-- GameState.hpp
|   |   |-- MoveStrategy.hpp
|   |   `-- Player.hpp
|   |-- gnn/
|   |   |-- FeatureExtractor.hpp
|   |   |-- GNNModel.hpp
|   |   `-- Graph.hpp
|   `-- ui/
|       |-- HexGameUI.hpp
|       |-- HexTile.hpp
|       `-- ImageViewer.hpp
|-- src/
|   |-- core/
|   |   |-- Board.cpp
|   |   |-- Cube.cpp
|   |   |-- GameState.cpp
|   |   |-- MoveStrategy.cpp
|   |   `-- Player.cpp
|   |-- gnn/
|   |   |-- FeatureExtractor.cpp
|   |   |-- GNNModel.cpp
|   |   `-- Graph.cpp
|   |-- ui/
|   |   |-- HexGameUI.cpp
|   |   |-- HexGameUI_updateVolumeIcon.cpp
|   |   |-- HexTile.cpp
|   |   |-- ImageViewer.cpp
|   |   `-- main.cpp
|   `-- cli/
|       `-- main.cpp
`-- selfplay/
    |-- gnn/
    |   |-- DataCollector.cpp
    |   |-- DataCollector.hpp
    |   |-- GameRunner.cpp
    |   |-- GameRunner.hpp
    |   |-- Serializer.cpp
    |   |-- Serializer.hpp
    |   `-- maingnn.cpp
    |-- mlp/
    |   |-- ReplayBuffer.cpp
    |   |-- ReplayBuffer.hpp
    |   |-- RLTrainer.cpp
    |   |-- RLTrainer.hpp
    |   |-- ValueMLP.cpp
    |   `-- ValueMLP.hpp
    `-- main.cpp
```

A. Core

Cette section regroupe les classes du noyau qui encapsulent l'etat du jeu, les joueurs et les strategies de decision.

Classe specialisations	Derivees directes / Description	
Board (classe)		Représentation et manipulation du plateau de jeu.
GameState (classe)		État courant du jeu et détection du gagnant.
Cube (classe)		Coordonnées cubiques pour les positions hexagonales.
Player (classe)	HumanPlayer, AIPlayer, HybridPlayer	Joueur de base et identité.
IMoveStrategy (classe)	RandomStrategy, MonteCarloStrategy, NegamaxStrategy	Interface de sélection de coups.
NegamaxStrategy (classe)	NegamaxHeuristicStrategy, NegamaxGnnStrategy	Stratégie Negamax avec évaluation et recherche.
Zobrist (classe)		Hashing du plateau pour tables de transposition.

TABLE II
PANORAMA DES CLASSES MODULE CORE.

Définition et implémentation de la classe board , qui est chargée de la manipulation et de la représentation du plateau dans le jeu Hex, encapsulant les conditions de taille du plateau, l'état d'occupation interne dans une matrice N*N , ainsi que également elle encapsule les opérations de base d'impression de l'état du plateau dans le terminal et le placement de pièces à l'intérieur du plateau dans l'état dans lequel il est stocké , elle se trouve dans Board.hpp-Board.cpp.

On implémente aussi la classe cube qui encapsule une coordonnée d'un espace tridimensionnel discret qui est utilisé pour modéliser les positions d'un espace tridimensionnel discret x, y , z, parce que malgré le fait que l'espace du plateau et sa représentation graphique soient bidimensionnels le fait que les cellules soient de forme hexagonale , permet l'usage d'une troisième coordonnée qui rend compte de la relation d'intercalation de lignes qui est déterminante quand on analyse le voisinage entre cellules , on incorpore aussi la surcharge de l'opérateur de somme pour des objets de cette classe et on intègre une méthode pour obtenir un identifiant unique en une valeur long pour chaque position à l'intérieur du plateau., dans Cube.hpp-Cube.cpp

Les trois types de joueurs que peut avoir le jeu son implémentes, la conception est construite de manière que le joueur soit constitué d'une identité du joueur et de sa capacité de

construction de mouvements , on définit la logique de sélection pour trois types de joueurs human player, qui demande à l'utilisateur l'entrée , ai player qui délègue la décision à une stratégie de mouvement spécifique et enfin le hybrid player qui peut se comporter comme un humain ou comme un ai player ceci dans Player.hpp-Player.cpp

On incorpore le noyau d'intelligence artificielle du projet, on implémente les stratégies de mouvement , random, Monte Carlo, et Negamax, et les deux manières de déterminer la fonction d'évaluation quand l'algorithme est Negamax, au moyen de la MLP ou au moyen de l'heuristique. On construit une unique représentation du plateau comme graphe , et on utilise un cache statique par taille n pour que le graphe ne soit construit qu'une seule fois par taille. On implémente la méthode pour construire la structure de plateau à partir de sa représentation linéaire, qui est utilisée pour construire des copies d'un plateau et pouvoir tester des coups., on incorpore la fonction pour vérifier la victoire instantanée , en créant un nœud un gamestate fils pour chaque coup légal et en vérifiant que dans aucun de ces cas le joueur ne soit gagnant.

On définit en traitant le plateau comme un graphe la fonction boundary distance pour identifier la distance minimale dont un joueur dans HEX a besoin pour connecter ses deux bords en utilisant la définition actuelle du plateau. L'algorithme boundaryDistance estime à quel point un joueur est proche de gagner à Hex en calculant le coût minimal pour connecter ses deux bords objectif. Il modélise le plateau comme un graphe d'adjacences et réalise une recherche de type 0–1 BFS depuis toutes les cellules du bord initial du joueur : traverser une cellule propre a un coût 0 (elle contribue déjà à la connexion), traverser une cellule vide coûte 1 (équivaut à une fiche qu'il faudrait encore placer) et les cellules de l'adversaire sont considérées comme bloquées. En s'étendant par voisins, il maintient le moindre coût accumulé jusqu'à chaque cellule et s'arrête lorsqu'il atteint n'importe quelle cellule du bord opposé, en renvoyant ce coût comme une approximation du nombre minimal de coups nécessaires pour compléter la connexion.

On implémente aussi des fonctions pour mesurer la quantité de mouvements libres par voisinage de chaque joueur, on utilise le bridge score pour augmenter l'importance de la cellule en fonction du nombre de voisins qu'elle a du même joueur, on définit un centre numérique et on donne des points à chaque cellule en fonction de la distance qu'elle a vers ce centre finalement on empaquette toute cette information sous forme de structure pour pouvoir l'utiliser dans l'évaluation de l'heuristique du Negamax. On incorpore aussi le hashing du plateau au moyen d'une table de transposition d'un objet de la classe Zobrist en initialisant deux clés aléatoires, par cellule une paire pour le joueur 1 et une autre pour le joueur 2 , on fait le hash par cellule avec le compute hash et on incorpore la mise à jour du hash en appliquant ou en défaisant des mouvements,

On implémente la classe Negamax, on définit aussi des classes dérivées de stratégies Negamax spécialisées qui sont NegamaxHeuristicStrategy qui force à utiliser seulement l'heuristique et NegamaxGNNStrategy qui configure l'usage du modèle et mélange avec cache, Negamax en général ce qu'il fait est de tester une victoire immédiate si ce n'est pas le cas ce qu'il fait est d'exécuter iterative deepening et il renvoie le meilleur mouvement, Negamax va de profondeur 1 jusqu'à la profondeur maximale par paramètre tant que la limite de temps n'a pas été dépassée Les résultats de profondeurs petites aident aux profondeurs grandes parce que : le meilleur mouvement trouvé avant sert comme "candidat principal", et cela améliore l'ordre d'exploration (spécialement avec table de transposition / killers / history), ce qui fait que alpha–beta élague plus. On utilise aussi une fenêtre limitée pour la recherche de alpha et beta, tant que le temps le permet avec l'usage de flags on obtient l'exploration de fenêtre complète en cas de fail high ou fail low par la sélection de la beta Dans ce Negamax, les killer moves sont gardés en mémoire à l'intérieur de l'objet NegamaxStrategy dans la structure killers, qui stocke deux mouvements par profondeur (killers[d][0] et killers[d][1]). Chaque fois que, durant la recherche, un mouvement m provoque une coupe alpha–beta (quand on vérifie alpha >= beta), ce mouvement est enregistré comme "killer" à la profondeur correspondante, parce qu'il a été assez fort pour écarter le reste des alternatives dans ce nœud. Dans des évaluations ultérieures à la même profondeur, l'algorithme consulte ces valeurs et leur donne une priorité élevée dans l'ordonnancement des coups, en les essayant d'abord pour augmenter la probabilité de nouvelles coupes et accélérer la recherche, sans altérer le résultat final de Negamax. On utilise le parallélisme pour répartir entre plusieurs threads l'évaluation des mouvements du tour actuel, il partage un alpha global pour élaguer plus, il synchronise le meilleur mouvement avec un mutex, et à la fin il sauvegarde le résultat dans la table de transposition pour le réutiliser dans des recherches futures. On parallélise uniquement le niveau racine de la recherche : plusieurs threads évaluent en parallèle différents coups initiaux (moves), en partageant un alpha global pour augmenter les coupes et en synchronisant le meilleur bestMove/bestScore avec un mutex. Ceci se trouve dans les fichiers MoveStrategy.hpp et MoveStrategy.cpp

<<<<< HEAD On implémente la logique de l'état de jeu ou GameState, qui est la classe pour définir la manière dont on représente, stocke et manipule l'état du jeu , incluant sa représentation linéale plane , en coordonnées cubiques, on incorpore les méthodes pour identifier s'il y a un gagnant et qui est le gagnant au moyen de techniques d'exploration par voisins dans les fichiers GameState.hpp et GameState.cpp
======= On implémente la logique de l'état de jeu ou GameState, qui est la classe pour définir la manière dont on représente, stocke et manipule l'état du jeu , incluant sa représentation linéale plane , en coordonnées cubiques, on incorpore les méthodes pour identifier s'il y a un gagnant et qui est le gagnant au moyen de techniques d'exploration par voisins dans les fichiers GameState.hpp et GameState.cpp
>>>>> main

B. Mlp-gnn

Dans cette section on inclut les éléments liés à la représentation en graphe du plateau et en plus à l'implémentation du calcul du score de mouvement au moyen de la stratégie d'IA qui implique une MLP ou celle qui implique un réseau neuronal de graphes. Les classes principales du module gnn sont résumées ci-dessous.

Classe specialisations	Derivees directes / Description
Graph (structure)	Graphe du plateau (adjacences, features, super-nœuds).
NodeFeatures (structure)	Vecteur de features par nœud (pions, bords, distances, joueur).
FeatureBatch (structure)	Tenseurs aplatis prêts pour l'évaluation GNN/MLP.
FeatureExtractor (classe)	Construit le graphe et produit les batches (cache par taille).
GNNModel (classe)	Wrapper TorchScript pour évaluer un modèle GNN/MLP.

TABLE III
PANORAMA DES CLASSES DU MODULE GNN.

On implémente la classe graphe et on utilise la définition cubique pour l'identification des voisins de chaque nœud Mettre à jour les features dynamiques qui dépendent bien de la partie actuelle:p1 = 1 si la cellule a une pièce du joueur 1 p2 = 1 si elle a une pièce du joueur 2 empty = 1 si elle est vide, cela se trouve dans les fichiers Graph.hpp et Graph.cpp

On transforme un état du plateau en un batch de données l'état du jeu , en construisant un graphe pour représenter l'état du jeu on convertit le graphe en un feature batch, puis après avoir aplati les caractéristiques du nœud et les arêtes du graphe. dans les fichiers FeaureStractoo hpp et Feature-Extracto.cpp, finalement dans ce dossier on implémente un wrapper C++ sur le modèle de PyTorch qui a la GNN ou sur la MLP dans une classe GNNNModel en TorchScript, le forward a des arguments; ici ils comptent combien d'inputs réels il reçoit (sans le self implicite). «««< HEAD Puis on l'utilise pour :expectsEdgeIndex(): si le forward attend au moins 2 inputs, on assume qu'il a besoin de edge_index. ce qui permet de différencier quel modèle est chargé dans MoveStrategy dans la méthode de l'évaluation, on définit aussi les méthodes de requête et d'évaluation du modèle dans le cas où il s'agit d'une GNN en prenant un batch de nœuds par features et d'arêtes, en construisant un tenseur, on incorpore aussi l'évaluation pour la MLP', qui ce qu'elle fait est de copier les features dans le tenseur et fait l'estimation. qui se trouve dans les fichiers GNNModle.cpp et GNNNModel.hpp ======= Puis on l'utilise pour :expectsEdgeIndex(): si le forward attend au moins 2 inputs, on assume qu'il a besoin de edge_index. ce qui permet de différencier quel modèle est chargé dans MoveStrategy dans la méthode de l'évaluation, on définit aussi les méthodes de requête et d'évaluation du modèle dans le cas où il s'agit d'une GNN en prenant un batch de nœuds par features et d'arêtes, en construisant un tenseur, on incorpore aussi l'évaluation pour la MLP', qui ce qu'elle fait est de copier les features dans le tenseur et fait l'estimation. qui se trouve dans les fichiers GNNModel.cpp et GNNNModel.hpp »»»> main

C. Self-play

On inclut dans cette section ce qui concerne l'entraînement avec reinforcement learning du modèle pour l'estimation du score qui est employé dans le Negamax du modèle heuristique en utilisant la MLP. RLTrainer, dont le rôle (du point de vue de la POO) est d'encapsuler tout le cycle d'entraînement par auto-jeu : configurer le modèle, générer des parties, transformer les parties en données, entraîner le réseau et sauvegarder/exporter les résultats. Dans le constructeur il applique la validation des invariants de configuration (cfg_), initialise les dépendances (modèle ValueMLP, optimizer, replay buffer) et sélectionne le dispositif (CPU/CUDA). Il maintient aussi deux instances du modèle : model_ (entraînement) et evalModel_ (évaluation), séparées pour la stabilité.

Classe / structure	Description
RLTrainer (classe)	Orchestration de l'auto-jeu, de l'entraînement et de la persistance (checkpoints/exports).
RLConfig (structure)	Paramètres de configuration de l'entraînement et de l'auto-jeu.
ReplayBuffer (classe)	Buffer circulaire d'échantillons avec échantillonnage aléatoire uniforme.
ReplaySample (structure)	Exemple d'entraînement (features + target).
ValueMLP (classe)	Alias/wrapper du module ValueMLPImpl pour l'usage LibTorch.
ValueMLPImpl (classe)	Réseau MLP de valeur; hérite de `torch::nn::Module`.
DataCollector (classe)	Collecte et agrège les états de parties auto-jouées.
Sample (structure)	Échantillon de jeu (plateau, joueur, résultat, coups restants).
GameRunner (classe)	Lance une partie entre deux stratégies et enregistre les états.
Serializer (classe)	Sérialise les samples en JSON Lines.

TABLE IV
PANORAMA DES CLASSES ET STRUCTURES DU MODULE SELFPLAY.

La partie d'auto-jeu utilise le polymorphisme via l'interface IMoveStrategy : l'entraîneur n'a pas besoin de savoir s'il joue un NegamaxStrategy ou un NegamaxHeuristicStrategy ; il appelle seulement select(...). Dans playOneGame on joue une partie complète, et à chaque tour on sauvegarde un EpisodeState (features + joueur au tour) dans un episode. Ensuite addEpisodeToBuffer parcourt l'épisode et le convertit en échantillons d'entraînement (ReplaySample) avec une cible basée sur le gagnant final, et les insère dans un replay buffer, en séparant clairement la responsabilité de "générer l'expérience" et de "la stocker". Finalement, trainUpdates réalise les étapes d'apprentissage : il échantillonne des batches du replay buffer, construit des tenseurs x (features) et y (targets), fait le forward, calcule la perte, rétro-propage et met à jour les poids avec AdamW, en appliquant du clipping pour la stabilité. Après l'entraînement, il synchronise evalModel_ en copiant les paramètres et les buffers (avec NoGradGuard) pour que l'évaluation soit cohérente, et optionnellement crée des

snapshots “frozen” pour la diversité des adversaires. Le fichier inclut aussi des responsabilités de persistance (checkpoints) et de sérialisation (export TorchScript + smoke test), complétant le pipeline dans une seule classe cohésive.

On implémente aussi une structure qui permet de stocker des données d’entraînement et d’extraire des échantillons aléatoires avec la classe replay buffer, on inclut aussi une fonction pour définir le réseau neuronal de valeur et l’exporter en TorchScript. Dans cette conception, mean, std et valueScale sont stockés comme buffers parce qu’ils font partie de l’état du modèle nécessaire pour faire l’inférence correctement (normaliser les entrées et mettre à l’échelle la sortie), mais ils ne doivent pas être optimisés comme s’ils étaient des poids ; en les enregistrant comme buffers, LibTorch garantit automatiquement que ces tenseurs se déplacent vers le même dispositif que le modèle quand on fait to(cpu/cuda), qu’ils soient inclus dans les checkpoints lors de la sauvegarde/chargement, qu’ils soient copiés quand on synchronise model_ avec evalModel_, et qu’ils soient exportés vers TorchScript, en évitant des erreurs typiques comme une normalisation perdue après chargement ou des échecs dus à des tenseurs sur des dispositifs différents.

III. Programmation Orientée Objet (POO)

A. Classes principales

Le projet organise ses classes par modules. Dans le noyau, on utilise Board, GameState, Cube, Player (HumanPlayer, AIPlayer, HybridPlayer), IMoveStrategy (con RandomStrategy, MonteCarloStrategy, NegamaxStrategy y sus variantes) y Zobrist. Dans le module gnn apparaissent Graph, NodeFeatures, FeatureBatch, FeatureExtractor y GNNModel. Dans le module selfplay on utilise RLTrainer, RLConfig, ReplayBuffer, ValueMLP, DataCollector, GameRunner y Serializer.

B. Héritage

L’héritage sert à abstraire des comportements communs. Des exemples directs sont HumanPlayer : public Player, AIPlayer : public Player et HybridPlayer : public Player dans le module core. De même, RandomStrategy : public IMoveStrategy, MonteCarloStrategy : public IMoveStrategy et NegamaxStrategy : public IMoveStrategy encapsulent différentes stratégies de décision, tandis que NegamaxHeuristicStrategy et NegamaxGnnStrategy héritent de NegamaxStrategy. Dans selfplay, ValueMLPImpl hérite de torch::nn::Module pour intégrer le modèle avec LibTorch.

C. Constructeurs

Par défaut: Board board; utilise le constructeur par défaut Board(int n = 7) dans src/cli/main.cpp. Avec paramètres: NegamaxStrategy p1(depthP1, cfg_.timeLimitMs, "", false, false, cfg_.alpha, false); dans selfplay/mlp/RLTrainer.cpp. Par copie: Board childBoard(base); dans src/core/MoveStrategy.cpp, qui invoque Board(const Board& other).

D. Templates

Le code emploie quelques templates utilitaires pour generaliser des operations selon le type d’entree. Les traits de specialisation seront detaillees plus tard; ici on liste seulement les templates. FixedArray<T, Size>sert de conteneur statique pour un nombre fixe d’elements (ex. directions). InRange<T>valide des bornes pour des indices entiers. evalValueModel<T>fournit un point d’entree unique pour evaluer le modele avec une entree MLP (features) ou GNN (batch).

E. Contraintes

Le seul usage explicite de contraintes sur templates est dans InRange, qui utilise std::enable_if pour n’accepter que des types entiers. Cette restriction evite des conversions implicites et garde l’intention claire pour la validation d’indices. Le reste repose plutot sur des specialisations (par exemple ModelInputTraits) plutot que sur des contraintes formelles.

F. Itérateurs

Le code utilise des itérateurs explicites principalement dans GameState: LinearBoard et ToCubeCoordinates parcourent la matrice du plateau, tandis que GetAvailableMoves itère sur la version linéaire. Dans Winner, des itérateurs servent à initialiser les files de BFS depuis les bords. Un autre usage apparait dans Graph, où l’on parcourt un unordered_map avec l’itérateur retourne par find pour relier les voisins. Cette approche garde un parcours séquentiel clair sans modifier la logique du jeu.

G. Opérateurs

Les opérateurs surchargés apparaissent dans plusieurs modules. Dans Cube, l’opérateur + sert à combiner des coordonnées et il est utilisé dans GameState et Graph pour explorer les voisins. Dans GameState, le type FixedArray définit operator[] pour accéder à la table de directions. Dans Player, HybridPlayer définit operator= pour gerer l’affectation en conservant l’identité et le mode (humain/IA). Dans GNNModel, CacheKey définit operator== et un functor operator() pour permettre la mise en cache des modules TorchScript dans un unordered_map.

H. Exceptions

Les exceptions presentes sont principalement des controles d’entree et d’environnement. Board et GameState jettent invalid_argument si la taille du plateau est non valide, et Board jette out_of_range pour un indice hors limites. Player et la CLI jettent runtime_error si la lecture d’entree utilisateur echoue. GNNModel jette runtime_error si le fichier du modele est absent, si le chargement TorchScript echoue, si la signature forward n’est pas supportee (1 ou 2 entrees), ou si l’evaluation est incoherente (modele MLP utilise avec edge_index, modele GNN sans edge_index, taille des features ou des aretes invalide). RLTrainer leve runtime_error quand CUDA est demande mais indisponible. Serializer jette runtime_error en cas d’echec d’ouverture ou d’ecriture JSONL.

Exceptions utiles a envisager: signaler explicitement un modele inexistant lorsque l'IA GNN est requise (au lieu d'un retour silencieux), lever une erreur si la taille lineaire du plateau n'est pas un carre parfait lors des conversions, et valider l'identifiant du joueur (1/2) dans Board::place. Pour le chargement de modeles, on peut aussi lever des erreurs plus precises: device CUDA demande mais indisponible a l'execution (dans l'interface), fichier corrompu ou incomplet, et incompatibilite de version LibTorch entre l'export et l'inference.

I. Gestion de memoire et pointeurs intelligents

La gestion de memoire s'appuie sur des pointeurs intelligents pour eviter les fuites et clarifier l'ownership. unique_ptr est utilise pour injecter les strategies dans AIPlayer et HybridPlayer, et pour gerer des strategies temporaires dans l'entrainement selfplay (p2Model/p2Heur). RLTrainer stocke aussi l'optimiseur avec un unique_ptr. shared_ptr est employe dans GNNModel pour le PIMPL et le cache de modules TorchScript, avec weak_ptr pour eviter des cycles et reutiliser un modele charge. Ces choix permettent une liberation automatique des ressources et un partage controle des objets lourds (modeles, strategies).

J. Parallélisme

Le parallele est utilise a deux endroits. Dans NegamaxStrategy, la recherche parallele au niveau racine divise l'evaluation des coups entre plusieurs threads, avec un alpha partage et un mutex pour proteger le meilleur score (src/core/MoveStrategy.cpp). Dans l'entrainement selfplay, RLTrainer lance plusieurs threads pour generer des parties en parallele, tandis que les mises a jour d'apprentissage restent sur le thread principal; les files et etats partages sont proteges par mutex et atomics (selfplay/mlp/RLTrainer.cpp).

IV. Installation

A. Prérequis

a) CMake:

```
cmake --version
```

b) Compilateur C++17:

```
g++ --version
# ou: clang++ --version
```

c) LibTorch (TorchScript):

```
ls $HOME/libtorch/share/cmake/Torch
# ou definir Torch_DIR/CMAKE_PREFIX_PATH
```

d) SFML 2.5:

```
pkg-config --modversion sfml-all
# ou: sudo apt-get install libsfml-dev
```

e) CUDA Toolkit (optionnel):

```
nvcc --version
```

f) Python 3 + poetry (optionnel):

```
python3 --version
poetry --version
```

B. Compilation et execution du jeu

```
cmake -S . -B build
cmake --build build
cd build
./hex_ui
```

C. Compilation et execution du selfplay

```
cmake -S selfplay -B selfplay/build
cmake --build selfplay/build
./selfplay/build/selfplay \
  <games> <minDepth> \
  <maxDepth> <outputPath> \
  <minPairs> <maxPairs> \
  <timeLimitMs>
# ou entrainement/export:
./selfplay/build/selfplay \
  --selfplay-train \
  --export-ts \
  --train-games 200 \
  --min-depth 10 \
  --max-depth 20 \
  --batch-size 256 \
  --updates-per-game 1 \
  --device cuda
```