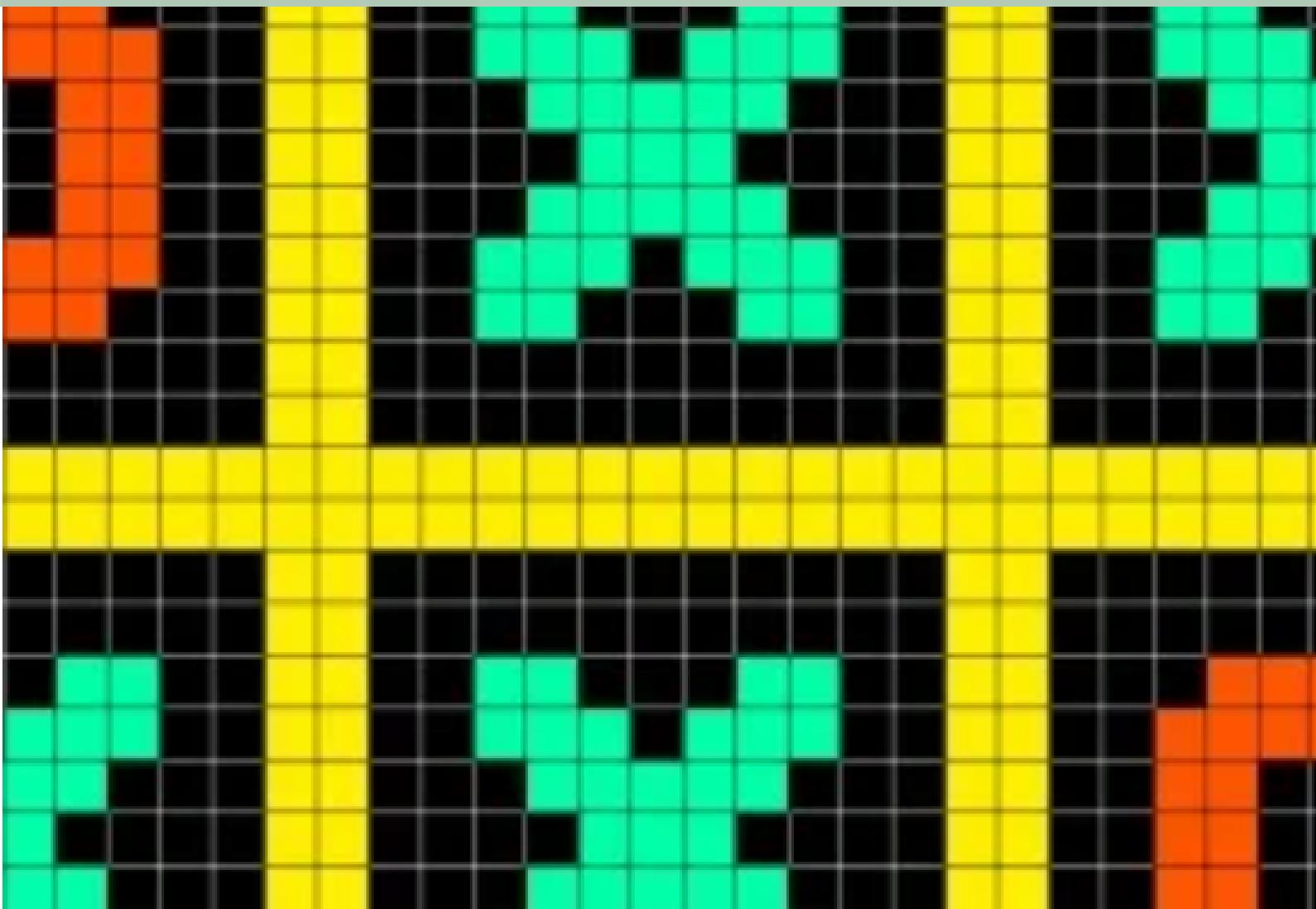


Power Rangers vs Morbac

Un projet Codingame





HAI606I : Projet de Programmation 2
L3 - Informatique
Faculté des Sciences de Montpellier

Cocheton Alexis
Collenot Heiarii
Poitelea Mihai
Ruiz Nicolas
Serva Benjamin
Tisserant Tom

Mai 2022 - Montpellier

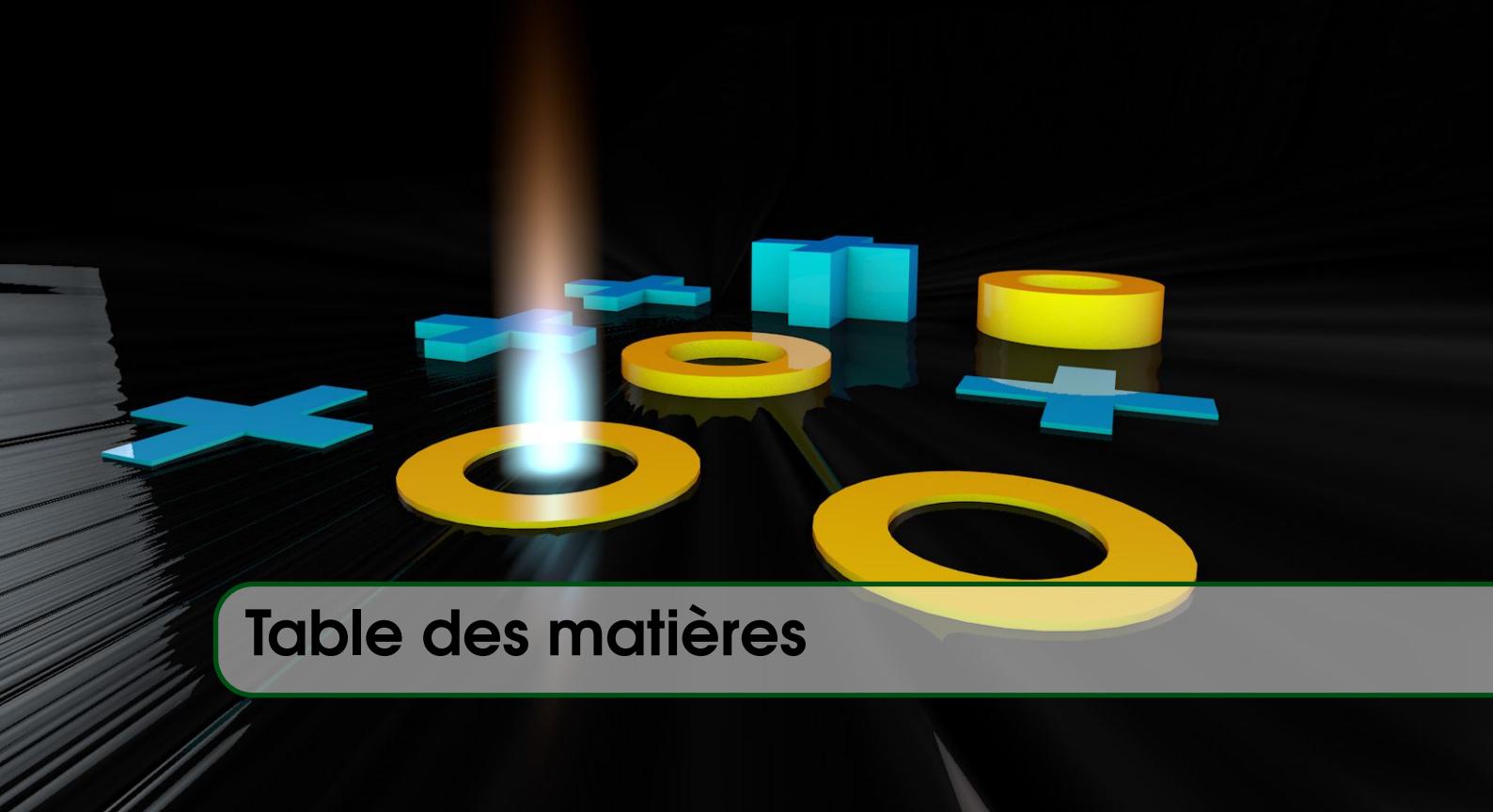


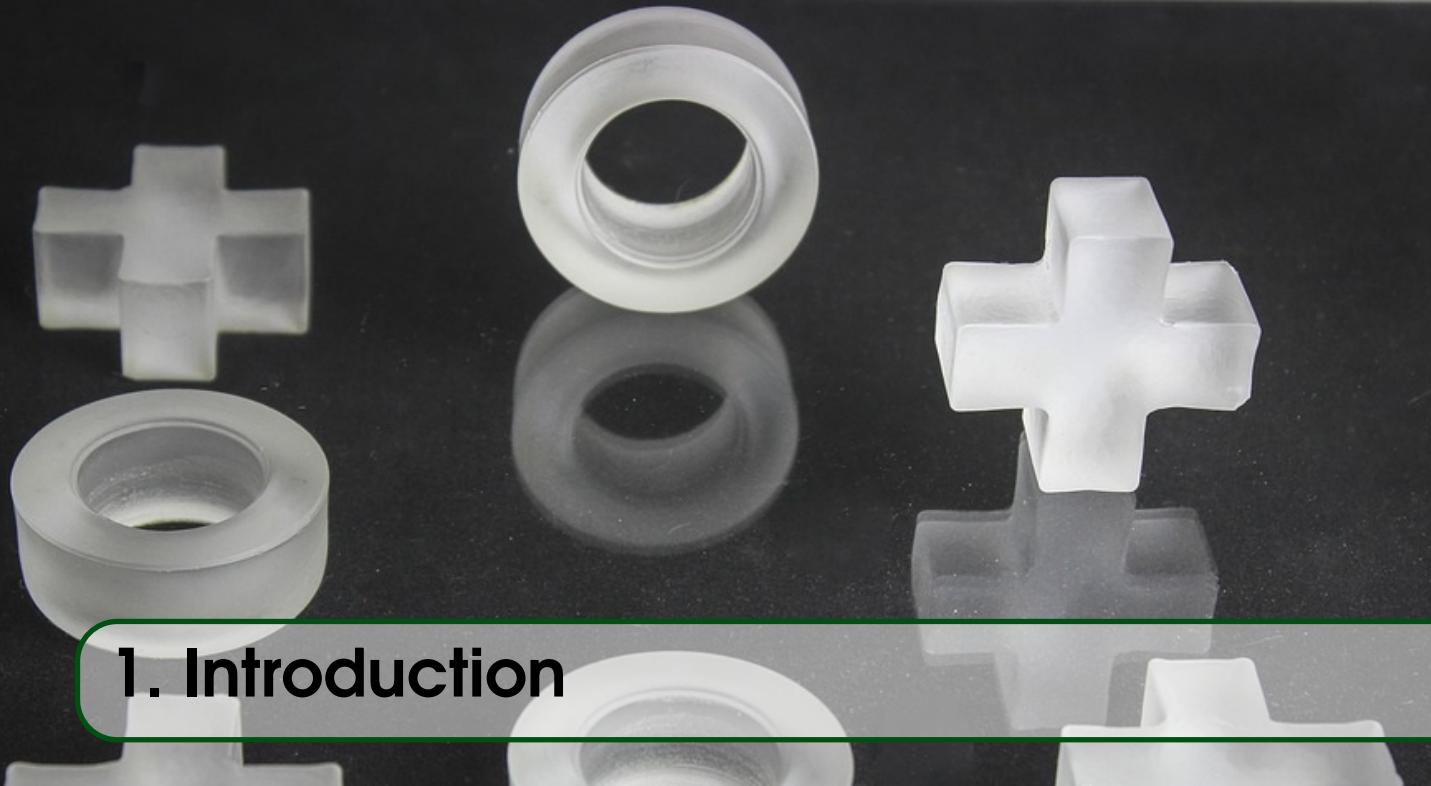
Table des matières

I	Autour du projet
1	Introduction
1.1	Contexte
1.2	Nos motivations
1.3	Différentes approches
1.4	Cahier des charges
2	Gestion de projet
2.1	Organisation
2.2	Diagramme de Gantt
3	Technologies
3.1	C++
3.2	Outils
3.3	Codingame
II	Développement logiciel
4	Conception et modélisation
4.1	Minimax
4.2	Monte-Carlo Tree Search

4.3	UML	19
4.4	Généricité	22
4.5	Smart pointers	23
5	Implémentation des algorithmes	24
5.1	Minimax	24
5.1.1	Structures de données et statistiques	24
5.1.2	Déroulement	25
5.1.3	Performance et complexité	29
5.1.4	Progression et comparaisons	31
5.2	Monte-Carlo	32
5.2.1	Structures de données et statistiques	32
5.2.2	Déroulement	32
5.2.3	Performance et complexité	35
5.2.4	Progression et comparaisons	35
5.3	Débogage	36
III	Conclusion	
6	Bilan	40
IV	Annexes	
Index		48

Autour du projet

1	Introduction	6
1.1	Contexte	
1.2	Nos motivations	
1.3	Differentes approches	
1.4	Cahier des charges	
2	Gestion de projet	9
2.1	Organisation	
2.2	Diagramme de Gantt	
3	Technologies	12
3.1	C++	
3.2	Outils	
3.3	Codingame	



1. Introduction

1.1 Contexte

Le projet voit le jour sous l'initiative de Marie-Laure Mugnier, Guillaume Pérution-Kihli et Florent Tornil qui proposent, à l'occasion de l'enseignement Projet de Programmation 2 HAI606I, le sujet 12 intitulé : “Projet de programmation d'un bot jouant au méta-morpion”.

Le méta-morpion (ou *ultimate tic-tac-toe*) est une variante du célèbre jeu du Morpion, la différence avec celui-ci réside principalement dans la composition de la grille de neuf cases. Cette grille est en fait une « méta-grille » dont les cases contiennent chacune une grille classique de Morpion. Ce qui nous fait neuf grilles de Morpion, soit quatre-vingt une cases au total. Afin d'apposer son symbole (croix ou rond) dans une des neuf cases de la « méta-grille », il faut gagner la sous-grille à la manière d'un morpion classique. Ainsi pour gagner la partie, il faut aligner trois symboles dans la grille principale, donc aligner trois morpions gagnés (verticalement, horizontalement ou en diagonale). La partie se termine lorsqu'un des joueur gagne ou qu'il n'y a plus aucun coup de disponible.

Pour accompagner cette évolution du morpion, certaines règles sont ajoutées :

- Le coup suivant dépend du coup précédent, le joueur devant jouer dans la sous-grille de la méta-grille dont la position correspond à celle où son adversaire a joué dans une des sous-grilles au tour précédent. Par exemple, si un joueur joue en bas à droite d'une sous-grille, l'adversaire devra jouer sur la sous-grille de morpion en bas à droite de la méta-grille.
- Exception à la règle précédente : si un joueur doit jouer dans une sous-grille déjà complète, ce dernier pourra alors choisir de jouer où il veut parmi les cases libres ,

c'est également le cas au premier tour.

- Lorsque un morpion est gagné il est considéré comme appartenant au vainqueur.
- Une fois le morpion gagné, on ne peut plus jouer dedans.

Une autre version de ce jeu existe (pas sur Codingame) où il est possible de continuer de jouer dans un morpion gagné. Cependant, en 2020 est exposée une stratégie qui permet au premier joueur de s'assurer la victoire. Cette version n'a donc pas été traitée lors de notre projet.

Le morpion est un problème très facile à résoudre par des algorithmes de parcours en

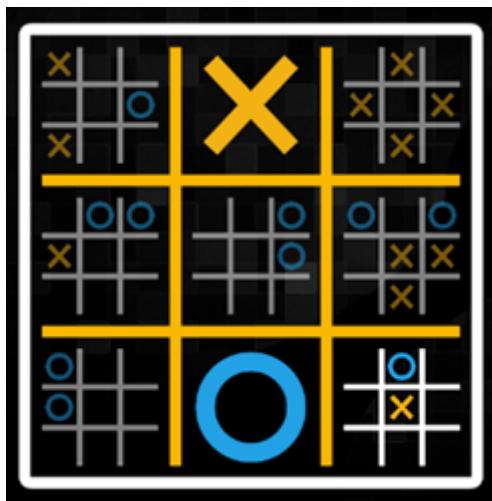


FIGURE 1.1 – Grille type du jeu méta-morpion

profondeur. À contrario, le méta-morpion n'est pas résoluble par une approche brute-force. Les problématiques centrales abordées sont la conception d'une intelligence artificielle capable de jouer à ce jeu tout en utilisant les algorithmes Monte-Carlo Tree Search et mini-max. Deux approches permettant de créer un programme devant monter dans le classement de la plate-forme Codingame pour y combattre des bots de plus en plus forts.

1.2 Nos motivations

Notre groupe de six s'est formé autour d'un même engouement pour ce projet et principalement pour la liberté de réflexion qu'offre la conception d'une IA, domaine montant aux multiples champs de réflexions.

Le méta-morpion est un jeu stratégique qui, bien que simple à comprendre, est plutôt difficile à maîtriser de par les nombreux éléments de réflexion possibles, tels que l'anticipation, garder une vue d'ensemble, ne pas se focaliser sur de petites victoires etc. C'est une problématique peu explorée et n'ayant aucune solution parfaite. Le projet, en plus de son aspect ludique, offre une grande maniabilité sur la conception et la configuration des IA ainsi que tout ce qui s'articule autour de son implémentation. Pour nous fixer sur le choix

de ce projet nous avions remarqué que le langage recommandé par les référents, à savoir le C++, nous permettait de commencer avec de bonnes bases tout en ayant une grande marge de progression. Sur une note plus globale, ce projet motive des intérêts professionnels certains. Chaque membre du groupe était également motivé par des intérêts différents tels qu'améliorer leur niveau en C++, ou s'essayer à la gestion d'une petite équipe.

1.3 Différentes approches

Le métamorpion partage des caractéristiques communes de jeu avec les échecs, les dames, le go... C'est pourquoi nous lui appliqueront les mêmes stratégies qui leur sont habituellement appliquées, à savoir Monte-Carlo Tree Search (MCTS) et minimax. Avec la première méthode, le bot joue en fonction de la plus grande probabilité de gagner. Il est donc très performant pour gagner pour jouer contre des adversaires humains, mais calculer ces probabilités demande un long temps de calcul.

Tandis qu'avec la seconde approche, le bot choisit de minimiser sa perte maximale, cherchant plutôt l'égalité que la victoire ce qui le rend très difficile à battre, mais victorieux moins souvent. De plus, il n'existe pas de fonction d'évaluation simple, donc face à un adversaire humain, l'IA est plus souvent en difficulté que ne le serait MCTS.

Les deux approches bénéficieront de plus amples explications dans la suite de ce rapport.

1.4 Cahier des charges

Comme évoqué antérieurement, l'élaboration d'une IA assurant la victoire au métamorpion n'étant pas résolu, la progression n'avait de fin que la date de rendu ; et nous avions convenu d'aller aussi loin possible que le projet nous aurait porté, avec toutefois des contraintes à respecter et des objectifs qui évoluaient au fil des livrables.

Certains points étaient annoncés dès le départ : nous devions coder une IA capable de jouer sur le site condigame, l'IA créée devait être un minimax ou un Monte-Carlo. Le C++ serait utilisé pour l'implémentation de cette IA. Ensuite, il a été décidé d'implémenter en utilisant des classes dans un souci de propreté et de lisibilité du code. À cela s'ajoute l'emploi de la généricité, et de techniques optimisées de gestions de mémoire telles que les structures dynamiques vector ou les pointeurs intelligents (smart pointers). Pour ce qui est de la gestion, il était également impératif de répartir le travail équitablement.

Il y avait donc un premier objectif d'étude et conception, puis d'intégration. Enfin pour se donner un objectif clair, et pouvoir quantifier la progression, nous nous sommes fixé de battre, au moins une fois, le bot de l'encadrant (voir annexe) 6.6, classé 110ème de la ligue or, et d'implémenter une IA qui rassemble minimax et MCTS.



PROJECT MANAGEMENT

2. Gestion de projet

2.1 Organisation

Au départ le projet consistait en l'étude et implémentation de surface du minimax, puis de s'attarder principalement sur le Monte-Carlo Tree Search qui pourrait éventuellement être amélioré avec les heuristiques du minimax. Néanmoins le projet a été conçu par les référents pour un groupe de quatre. Au bout de deux semaines, nous avons donc décidé de séparer le groupe et le projet en deux. Le premier groupe de trois s'occuperait du MCTS pendant que l'autre groupe ferait le minimax mais plus approfondie que prévu. Cela correspondait d'ailleurs à notre motivation première d'élaboration d'heuristiques pour l'IA, plutôt que de laisser faire le hasard. Et avec une réunion potentielle des deux projets.

Pour pouvoir communiquer entre étudiants et avec les référents nous avions plusieurs méthodes : En premier lieu, les encadrants communiquaient avec le chef de groupe par mail mais il a très vite été choisi de communiquer grâce à un serveur Discord pour qu'une communication globale puisse s'installer, permettre des échanges plus fréquents avec les encadrants, et pouvoir leur poser plus simplement des questions techniques ou d'implémentation.

De plus, des réunions plénière bimensuelles avaient lieu au LIRMM (Laboratoire d'Informatique, de Robotique et de Micro-électronique de Montpellier). Un compte-rendu en était extrait afin de guider les réunions en internes qui suivaient peu après. Ces dernières étaient hebdomadaires, avec pour objectif d'échanger sur l'avancement de chacun et d'attribuer des tâches.

Enfin pour une communication plus fluide, et organiser des séances de codage en binôme/trinôme nous organisions des séances de travail dans des salles informatique de l'université.

Nous avions mis un management de type agile mais il a été nécessaire de conserver un semblant de hiérarchie pour ne pas stagner dans la prise de décision et impulser les initiatives. Pour diriger l'équipe et diviser les tâches, deux personnes furent assignés aux rôles de chef de groupe et manager et une personne fut nommée comme chef développeur. Ces rôles avaient comme tâches la communication, l'organisation, la gestion des outils entre autres.

Ces attributions ne sont pas faites au hasard, une enquête (figure 6.11) a été diligentée par le chef d'équipe auprès de ses membres dès la création du groupe, afin de connaître les objectifs et motivations personnelles de chacun aux travers de leurs besoins, leurs limites, leurs attentes, les habitudes de travail, leur disponibilités etc. Nous avons ensuite cristallisé ces décisions en nous prêtant au jeu d'incarner le système hiérarchique des héros de la franchise des Power Rangers (système de couleurs), unis pour défaire le Robot (bot) Méta-morpion. Cela avait bien-sûr de multiples objectifs tels que favoriser l'entente, et l'adhésion au groupe et le sentiment d'appartenance pour booster les résultats de l'équipe.

Pour diriger le travail et gérer notre temps, nous avons créé un diagramme de Gantt interactif sur le site Bitrix24 (figure 6.9) ce qui a permis à tout les membres de le consulter et de le modifier à leur convenance puisque les managers l'actualisait régulièrement.

Nous avons suivi un plan classique de création de logiciel. Dans un premier temps nous avons fait des rappels de C++ et d'UML, puis effectué des recherches sur la plate-forme Codingame ainsi que sur les algorithmes minimax et MCTS. Les recherches étaient ensuite exposées et expliquées au cours de réunions au sein du LIRMM.

Ensuite vint la phase de modélisation et conception à l'aide de l'UML et de pseudo code. Nous nous sommes mis d'accord sur ce que nous allions coder, et comment, en précisant les besoins et en actualisant le cahier des charges. Enfin, vint la scission en deux groupes pour la phase d'implémentation (coder, déboguer, optimiser) des deux IA, et ce, jusqu'à la date de rendu de projet.

2.2 Diagramme de Gantt

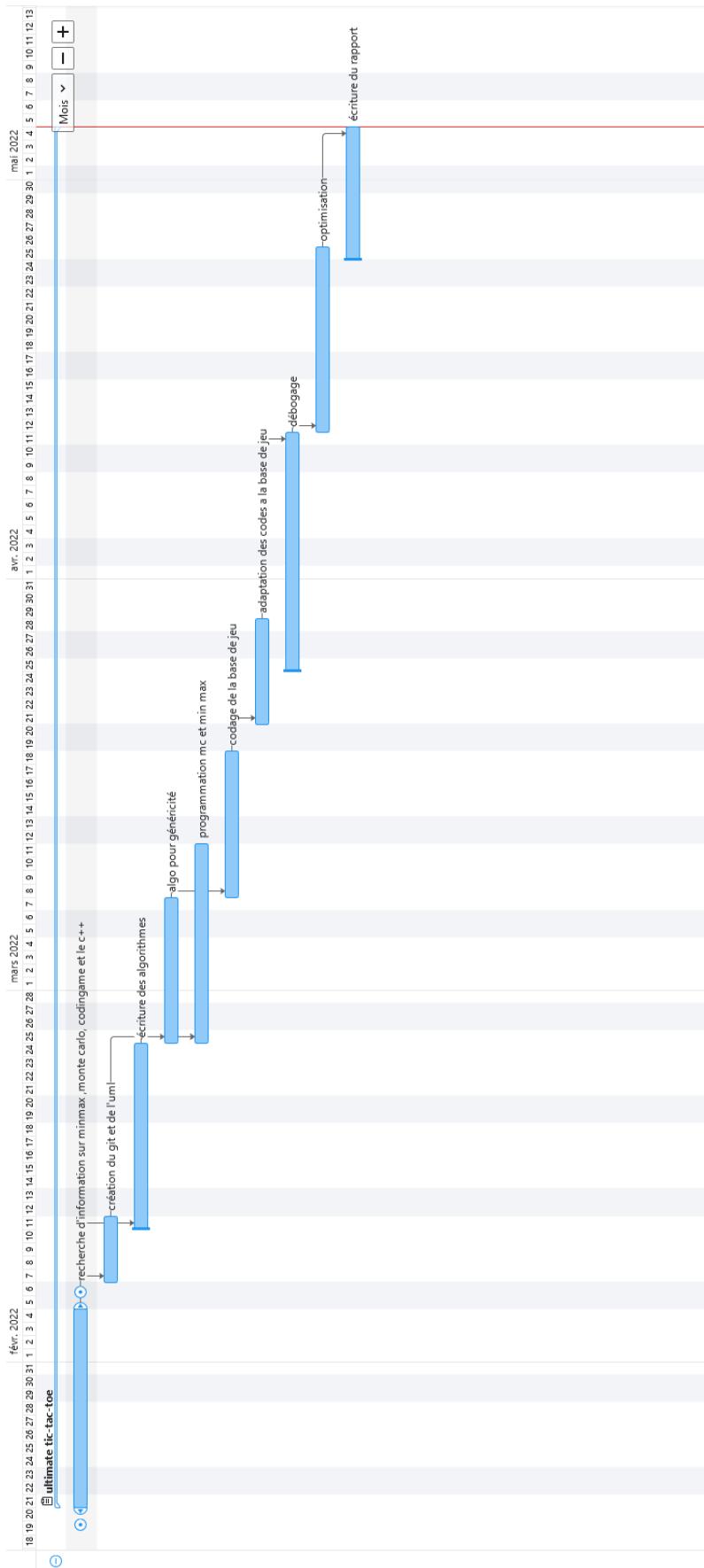


FIGURE 2.1 – Diagramme de Gantt

3. Technologies

3.1 C++

Ce n'est pas sans raison que C++ constitue un choix imposé dans le cadre du projet. C++ est un langage compilé, donc à priori plus rapide à l'exécution qu'un langage interprété comme Python. Il offre une précision pour l'optimisation avancée tout en faisant de la programmation orientée objet. En effet Java "tout référence" ne nous laisse pas choisir en détail si l'on souhaite passer en copie, par référence, utiliser un certain type de pointeur... C++ offre aussi un gestion des structures de données et des types que Python nous cache, et qui ralentit fortement l'exécution. Or, la vitesse est primordiale pour ce projet. Le fait que ce soit orienté objet est un point très important, car il nous permet de mettre en pratique les notions vues en cours de conception et modélisation UML et celui de modélisation et programmation objet avancée, les objectifs du projet.

Enfin, il permet aussi la généricité, demandée dans le cahier des charges. Il dispose aussi de nombreuses librairies et fonctionnalités, car continuellement mis à jour, et donc d'une bonne documentation. En effet, C++ est bien classé dans les langages les plus usités et demandés dans le milieu professionnel, c'est pourquoi ce projet constitue une bonne opportunité d'amélioration, car c'est un langage que nous avons tous déjà abordé au cours notre scolarité.

3.2 Outils

Voici une liste des différents outils qui ont permis un bon déroulement du projet :

Communication et Management :

1) *Discord* :

La communication avec les encadrants, s'est essentiellement faite sur la plate-forme Discord, une application originellement déployée pour promouvoir la communication entre joueurs de MMORPG.

Aujourd'hui Discord fait office de réseau social, et permet de gérer des groupes de discussion, par thèmes, par permissions (rôles), et sert accessoirement de dépôts et stockage pour des fichiers non volumineux. Il y est possible d'échanger par messages, par vidéos, de partager son écran en temps réel, d'interpeller un membre (ping) etc. Ses options d'interfaces, l'édition de message, la possibilité de présenter du code similaire à un IDE, les nombreux bots installables, en font la plate-forme de choix pour de nombreux projets, notamment mais pas exclusivement de programmation.

On a donc scindé le serveur en deux catégories, une pour les encadrants (pour poser des questions spécifiques au code ou organiser le projet), et l'autre pour les membres (minimax, Monte-Carlo, UML, information...) (voir annexe).6.10

2) *Wizeboard* :

Witeboard est un outil en ligne, qui permet de faire des dessins. Utile pour représenter des cas de jeu particuliers dans le métamorphe ou bien pour travailler sur l'arbre des possibilités etc. On l'a utilisé tout au long de nos réunions pour exposer différents problèmes rencontrés au cours du projet et pour mieux se projeter avant de travailler dans le code.

3) *Bitrix24* :

C'est une application de management qui permet d'affilier des tâches à chacun avec des dates butoirs ainsi qu'une priorité. Ça nous a permis de garder une trace de tous ce qui a été fait et par qui. Bitrix24 partage de nombreux points communs avec Discord, mais le premier étant déjà maîtrisé des membres, ce dernier se cantonne à la partie manageriale.

Collaboratif :

4) *Overleaf* :

Outil en ligne qui permet d'écrire le rapport en LaTeX. Il nous permet de travailler à plusieurs et surtout en même temps sur le rapport.

5) *Git* :

On a créé un espace git pour partager le code avec tout le monde, et pouvoir y accéder de n'importe quel ordinateur (très utile pour la transition de chez nous à la faculté) et de garder différentes versions de celui-ci. Dans le git il y a une branche principale avec différents fichiers tels que le rapport, les umls, images... ; On y trouve aussi deux branches secondaires, pour minimax et pour Monte-Carlo et dans lesquelles se trouve les différents codes.

6) *Lucidchart* :

Outil en ligne qui permet de créer des UML, et surtout à plusieurs en même temps.

3.3 Codingame

C'est un site qui propose des défis de programmation, notamment sous forme de combats de bots, dans des contextes (des jeux). Cette plate-forme française se veut un vitrine pour ses inscrits, par exemple elle favorise la recherche d'emploi. De nombreuses entreprises n'hésitent pas à proposer un défi Codingame. On peut voir que c'est une fonctionnalité phare, présente à tout moment sur l'interface (en haut à droite "trouver un job").

L’interface de l’IDE se présente comme sur l’image qui suit. Une partie avec le code à droite, une fenêtre graphique de l’avancement de la partie lancée, qui permet de visionner coup-par-coup le jeu, et enfin une sortie standard (*cout*) des coups joués et une sortie pour les erreurs (*cerr*). Avec trois types d’erreurs : erreur de compilation du code, *timeout* et *eliminated : invalid action*. Il existe une erreur par rapport à la taille de stockage (768 MB) utilisée, mais nous n’y avons jamais été confronté. Le *timeout* quant à lui se produit lorsqu’aucune réponse valide n’est reçue avant le temps imparti, qui est d’une seconde au tout premier tour de chaque opposant, et de 100 millisecondes pour les tours suivants. La moindre erreur déclare bien sûr perdant celui qui l’a commise.

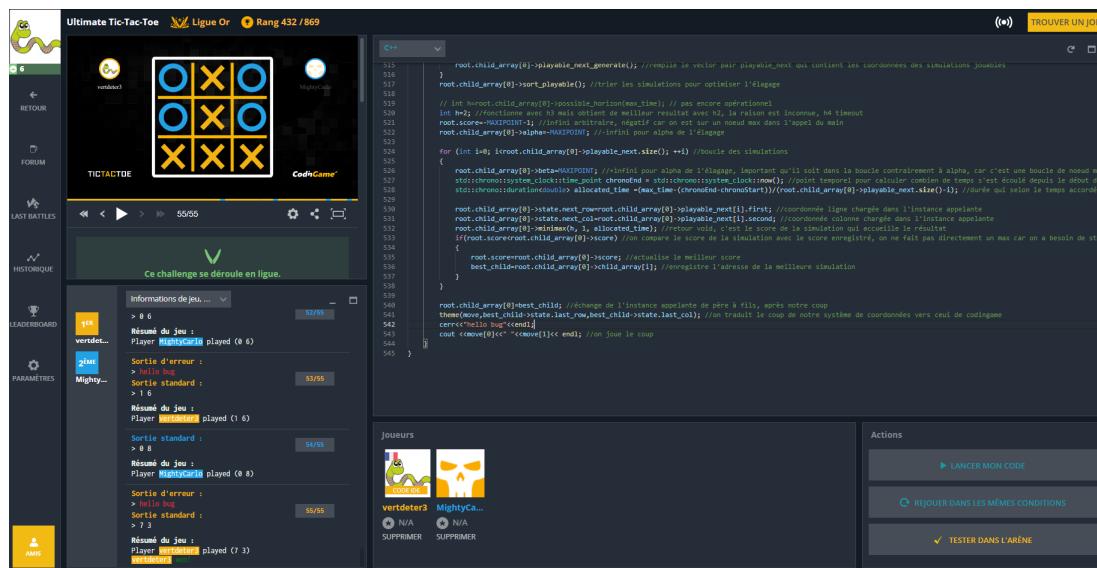


FIGURE 3.1 – Interface IDE Codingame

Codingame nous laisse la possibilité de traiter ce problème dans 27 langages différents. Et chaque langage possède des versions, options et librairies à prendre en compte, par exemple `lpthread` pour C++. Il y a plusieurs ligues qui vont de la ligue bois à la ligue légende en passant par bronze, argent et or. En sachant que la ligue bois est un simple morpion, ce n'est qu'à partir de la ligue bronze qu'on doit résoudre le problème du métamorpion.

Une fois le code écrit il suffit de choisir un adversaire, généralement le boss de ligue mais on peut s'affronter nous même ou tout adversaire appartenant à la même ligue. Ensuite nous lançons un test avec le bouton "lancer mon code". Si celui-ci est compilable, on peut le soumettre à l'arène, une succession de deux-cent vingt combats contre des adversaires de notre ligue. À l'issu de ces combats, un score est obtenu, et si celui-ci est plus élevé que celui du boss, nous sommes promus vers la ligue suivante. Enfin, chacun des combats ayant eu lieu est visionnable, et si l'un nous intéresse, on peut le charger dans l'IDE pour reproduire ou simplement combattre un adversaire précis.

L'une des fonctionnalités les plus intéressantes de Codingame est de 'rejouer dans les mêmes conditions', et cela s'avère très utile car l'adversaire va refaire les mêmes coups (même graine aléatoire) donc on peut voir si les changements apportés ont été efficaces ou bien néfastes. Cependant, cette fonctionnalité n'est plus aussi efficace arrivée en ligue or. Les bots de la ligue or sont le plus souvent basés sur MCTS, et comme celui-ci travaille à base du temps restant, même si la seed est identique, il se peut que le temps ne le soit pas, et donc la simulation finit par être altérée.

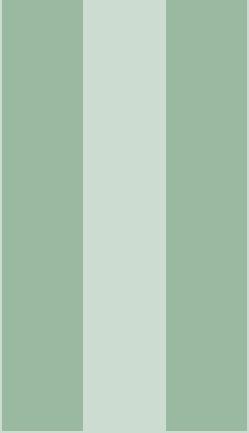
On reçoit plusieurs variables de la part de Codingame, et on doit obligatoirement les lire, dans le cas contraire le jeu se désynchronise et nous ne pourrions plus récupérer correctement les coups joués par l'adversaire. Ne pas correctement interagir avec Codingame va donc forcément entraîner la défaite.

On a donc plusieurs variables d'entrées qui sont :

- Le dernier coup de l'adversaire, séparé en deux variables qui sont respectivement 'opponent_row' et 'opponent_col', si c'est le début de partie et que l'adversaire joue en second alors celles-ci seront égales à -1.
- valid_action_count, c'est le nombre de coups jouables.
- Les coordonnées de tous les coups que l'on va pouvoir jouer (une ligne et une colonne), avec une boucle itérant valid_action_count afin de toutes les parcourir.

Au total on a $2+1+(2*nb)$ entrées, factorisé en $2(nb+1)+1$ entrées qui sont fournies par Codingame.

Notre code doit lui fournir en sortie standard deux variables qui sont la ligne et la colonne (variables séparées par un espace) du coup que l'on veut jouer. Bien entendu ce coup doit correspondre aux règles du jeu à l'instant T. Par exemple, si l'on joue dans une case déjà prise on perd la partie mais également si on joue autre part que dans une case autorisée.



Développement logiciel

4 Conception et modélisation 17

- 4.1 Minimax
- 4.2 Monte-Carlo Tree Search
- 4.3 UML
- 4.4 Généricité
- 4.5 Smart pointers

5 Implémentation des algorithmes 24

- 5.1 Minimax
- 5.2 Monte-Carlo
- 5.3 Débogage



4. Conception et modélisation

Comme évoqué au début du rapport, les deux types d'IA recommandées par les encadrants sont le minimax et la recherche arborescente Monte-Carlo ; tous deux utilisés dans le cadre de la prise de décision de jeux en un contre un. Nous allons d'abord procéder à leur conception (étude), puis modélisation (UML). Enfin nous nous attarderons sur la conception des points suivants du cahier des charges : généréricité et smart pointers.

4.1 Minimax

L'algorithme minimax s'appuie sur le domaine mathématique de la théorie des jeux, pour les jeux à somme nulle (tout ce qu'un joueur gagne l'autre le perd obligatoirement) et à information complète (les joueurs ont toutes les informations sur ce que joue leur adversaire).

Le bot va alors simuler tous les coups possibles en un nombre limité de tours. Ce nombre de tour est noté h pour horizon. Par exemple pour le métamorpion, avec un h de 2 il testera toutes les cases sur lesquelles on peut jouer puis toutes les possibilités qu'aura l'adversaire et de nouveau tout nos coups possibles (zéro indicé). Une fois la hauteur (horizon) voulue atteinte, l'algorithme va assigner des valeurs à l'état du plateau (évaluation statique).

Le coup choisi sera alors celui qui minimise les pertes du joueur, en supposant que les deux joueurs jouent toujours parfaitement, à savoir, le joueur cherche à maximiser la pondération de l'état du jeu, tandis que l'adversaire cherche à la minimiser (ce qui revient à la maximiser de son point de vue). Pour ce faire et puisque le point de vue ne change pas, l'algorithme alterne, en fonction de la hauteur, entre prendre le maximum et le minimum de tous les fils d'un nœud, voir figure ci-dessous.

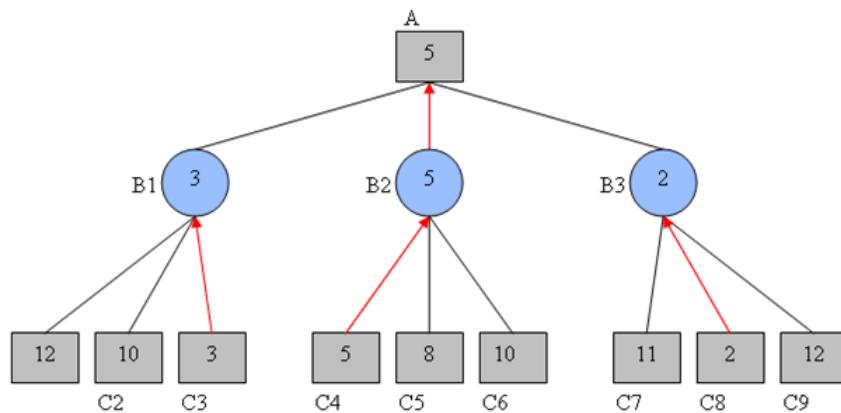


FIGURE 4.1 – Arborescence minimax
Nœuds min (cercles) / Nœuds max (rectangles)

À cela peut s'ajouter l'élagage (ou pruning) alpha/bêta. Cela consiste à arrêter préma-turément le parcours en largeur (et donc en profondeur des nœuds élagués) si la condition $\alpha \geq \beta$ est remplie. Où α est le plus grand minimum espéré, et β le plus petit maximum espéré, pour le nœud maximum (le joueur qui cherche quel coup jouer). Plus simplement, ses paramètres servent à comparer les valeurs entre des générations successives, et la condition peut vouloir dire deux choses. Si le frère cadet B1 d'un nœud max A1, obtient sur un de ses enfants nœud min B2 un score inférieur à β il va l'élaguer. Le raisonnement est le suivant. Si le joueur choisi de jouer B1 alors il sait qu'au prochain coup son adversaire va jouer B2 qui le désavantagera plus grandement que le pire score qu'obtient le nœud A1. Ainsi, il va refuser catégoriquement de jouer B1, et donc il n'est pas la peine de continuer l'exploration des fils de B1, les frères de B2. Le raisonnement est inversé lorsque c'est au tour du nœud min de calculer, et d'élaguer.

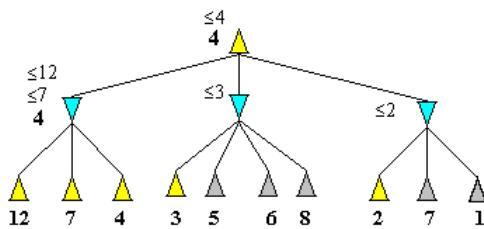


FIGURE 4.2 – Élagage de l'arborescence
Nœuds max (jaunes) / Nœuds min (bleus)

L'élagage ne modifie en rien le résultat que produirait un parcours exhaustif. Il permet de se contenter d'un parcours partiel pour économiser des ressources. Un élagage est particulièrement efficace lorsqu'il est couplé à des fonctions de tri des coups à calculer, pour faire en sorte d'économiser un maximum de ressource. Il en résulte que l'horizon calculable peut être doublé grâce à ce gain de ressource (temps).

Un élagage se base sur une évaluation, donc est dépendant de la qualité des heuristiques mises en place. Si l'adversaire ne dispose pas des mêmes heuristiques, il peut arriver qu'il joue un coup imprévu, considéré comme non parfait (car nous avantageant selon nos critères, au lieu de chercher à nous pénaliser). En effet l'évaluation est subjective, et dépendra de l'implementation de chaque programmeurs. Ainsi, on peut se retrouver face à un coup qui n'aura pas été calculé dans l'arbre. Dans ce cas de figure, au niveau d'une implémentation arborescente, il faudra prévoir la création de ce nœud manquant, et cela suppose aussi de perdre toute connaissance de l'horizon qui été connu d'avance, une remise à zéro (reset).

4.2 Monte-Carlo Tree Search

Monte Carlo Tree Search (MCTS) est une technique de recherche dans le domaine de l'intelligence artificielle bien connue. Il s'agit d'un algorithme de recherche probabiliste et heuristique qui combine les implementations classiques de la recherche arborescente aux principes d'apprentissage automatique de l'apprentissage par renforcement.

Dans la recherche arborescente, il est toujours possible que la meilleure action actuelle ne soit en fait pas l'action la plus optimale. Dans de tels cas, l'algorithme MCTS devient utile car il continue d'évaluer d'autres alternatives périodiquement pendant la phase d'apprentissage en les exécutant, au lieu de la stratégie optimale perçue actuelle.

Une explication simple de MCTS serait de dire qu'un nœud va parcourir au hasard certaines simulations, jusqu'à les terminer, et ensuite retenir qui est le vainqueur. Des statistiques sont établies à partir de nombreux scores qui auront été remontés (back/rétro propagation), et le coup à jouer sera celui qui aura la plus grande chance de victoire par rapport au nombre de parties calculées. D'autres techniques existent, plutôt que de choisir au hasard, on peut privilier l'étude des meilleurs coups, ou alterner entre nœuds à bonne probabilité, et nœuds peu explorés (UCT).

Plus de détails dans le fonctionnement de cet algorithme seront disponibles dans la partie algorithmique dédiée au MCTS.

4.3 UML

La modélisation est l'un des points les plus importants dans le développement d'un projet. Une bonne modélisation permet à tous les membres du groupe de bien visualiser son architecture et éviter les imprévus. En premier lieu, nous avons fait le choix de séparer notre projet en deux parties, une consacrée à la partie Monte-Carlo et une autre consacrée au minimax.

Nous voulions les fusionner en une seule et même partie (voir Schéma UML de la fusion des deux programmes). Malheureusement bien que nous ayons tenté de modéliser au mieux le projet. Certaines difficultés nous ont obligés à dévier de notre modélisation initiale, sans quoi, il aurait été plus difficile de rendre fonctionnel le programme. Nous nous sommes donc retrouvés avec deux programmes possédant des architectures différentes.

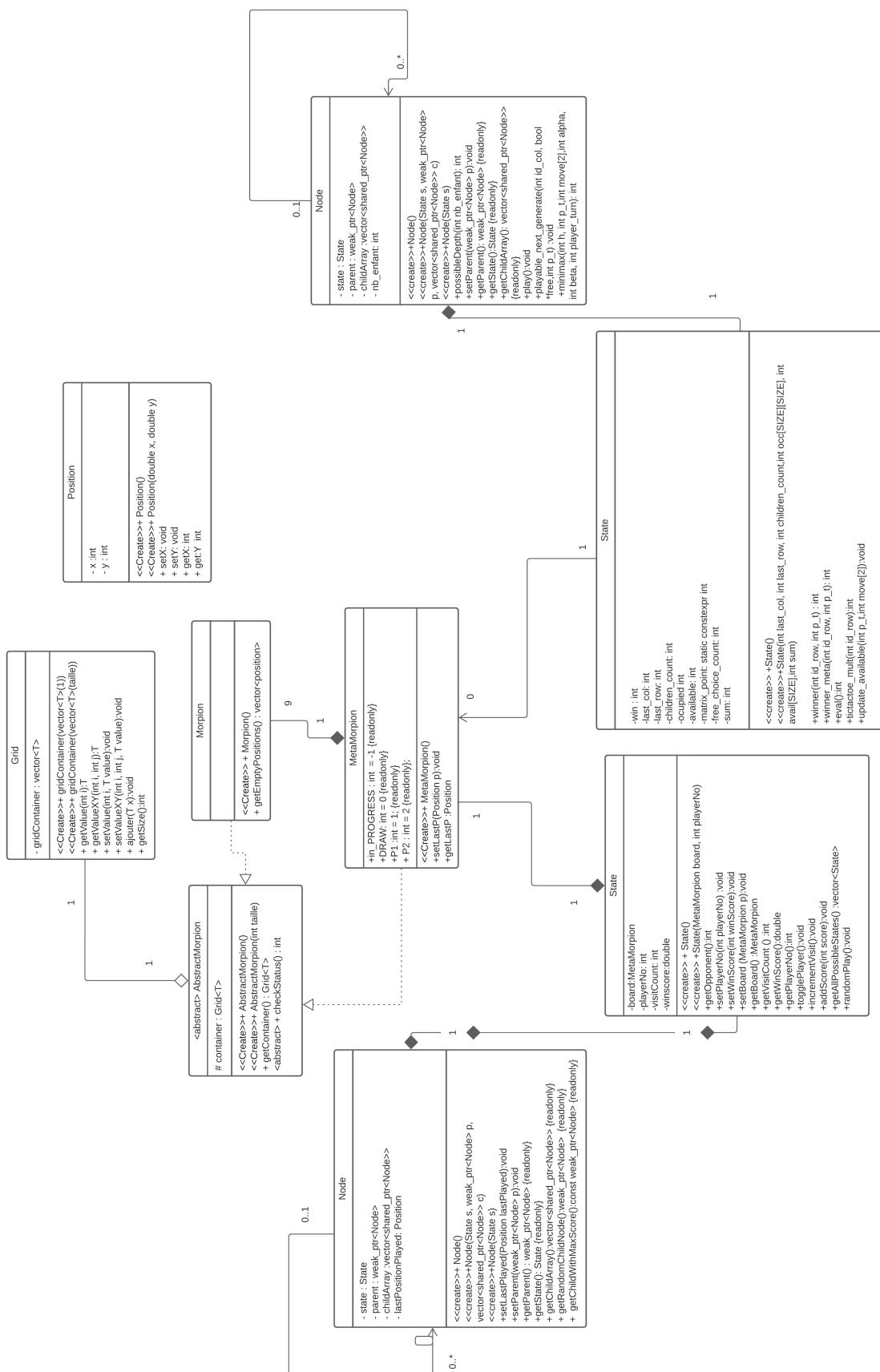


FIGURE 4.3 – Schéma UML de la fusion des deux programmes

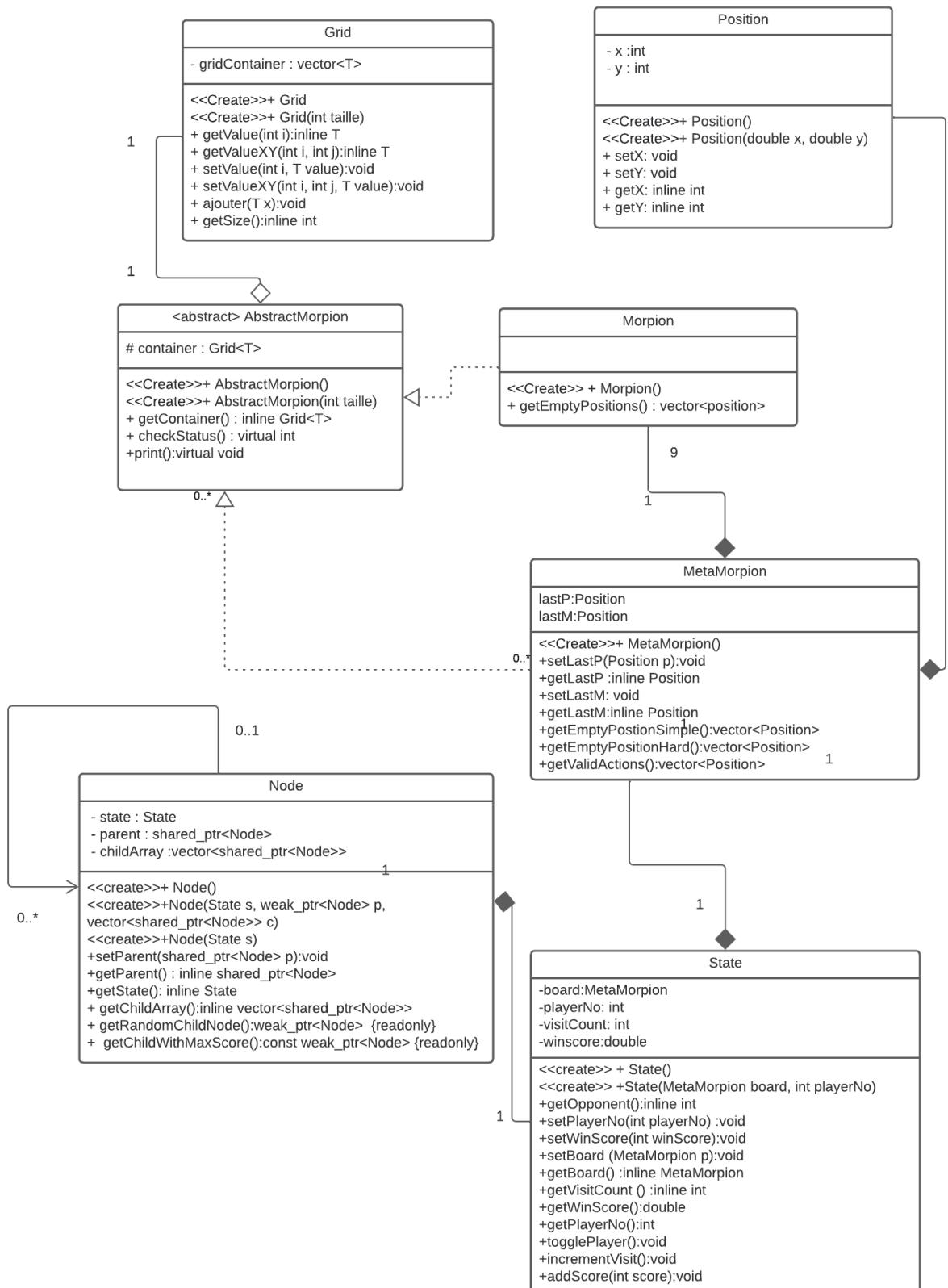


FIGURE 4.4 – Schéma Uml MCTS

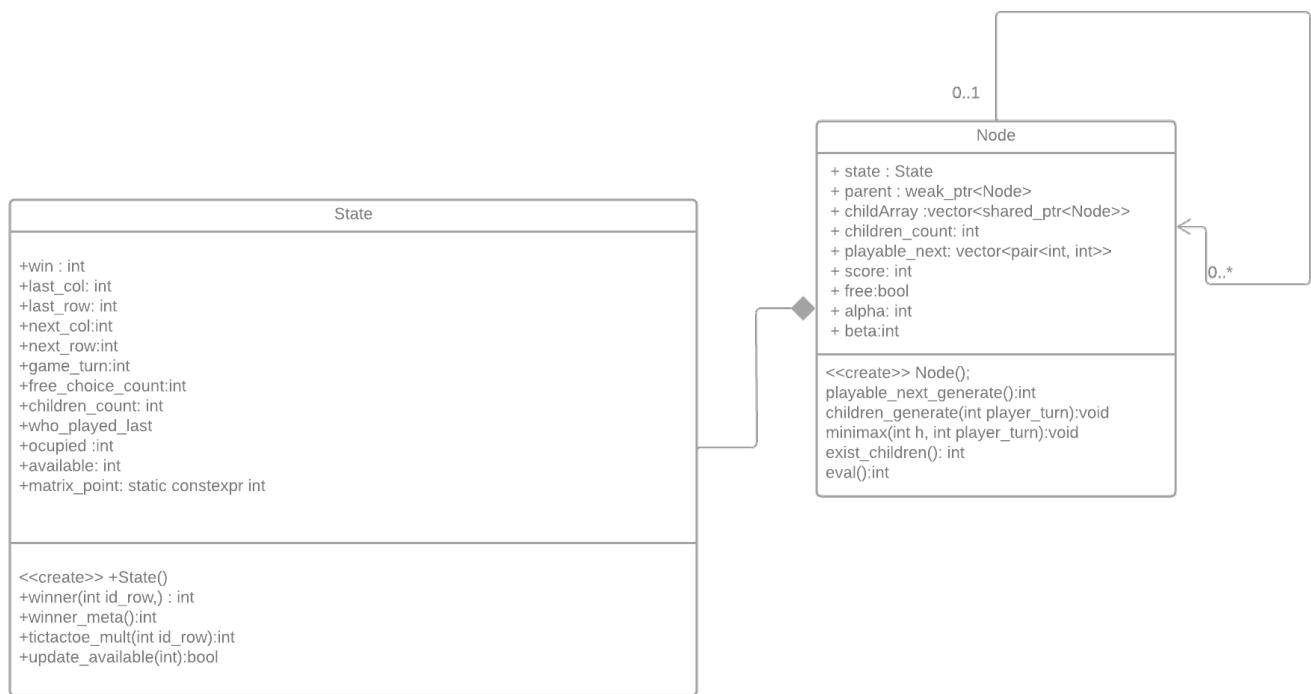


FIGURE 4.5 – Schéma Uml Minimax

4.4 Généricité

Des le début du projet, sur les conseils de nos encadrants, il nous a été recommandé d'utiliser la généricité. Que ce soit pour réutiliser nos algorithmes pour des problèmes différents se basant sur des structures de données similaires, pouvoir les faire travailler ensemble, ou encore changer la façon de traiter les données facilement, il nous était utile de faire un partie générique aux deux IA. Cette partie commune contiendrait notamment le plateau de jeu, et son conteneur, c'est pourquoi comme vu sur l'UML, on peut noter la présence de classes template et/ou abstraites comme "Grid" ou "AbstractMorpion".

Pour rentrer plus en détails, créer une classe template "Grid" permet de centraliser des méthodes créées par nos soins, agissant en l'occurrence sur un vector, et ceci peu importe le type des données choisies dans le template. La classe "AbstractMorpion" centralise quant à elle les méthodes en commun entre ses classes filles : "MetaMorpion" et "Morpion", en effet, bien qu'elles soient légèrement différentes, elles partagent beaucoup de méthodes en commun et aussi une variable membre qui est le conteneur de type Grid.

En bref, la généricité, bien que nous n'ayons pas réussi à l'appliquer parfaitement sur les deux algorithmes (besoins de gains de temps pour la plate-forme, etc...), est un concept fondamental et très intéressant que nous avons assimilé et compris, et que nous appliquerons certainement dans de futurs projets.

4.5 Smart pointers

Les pointeurs intelligents nous ont été proposés pour gérer efficacement la mémoire. Rappelons que l'espace de stockage limite accordé est de 768 MB, et les simulations se comptant en milliard. Donc si nous ne procédons pas à une gestion même minime de celle-ci, nous pourrions faire planter l'instruction en cours.

Au départ, deux types de pointeurs intelligents sont conçus, les unique, que nous n'utilisons pas et ne détaillerons donc pas, et les shared. Les pointeurs intelligents, ont d'intelligents qu'ils gèrent eux-mêmes leur suppression lorsque cela est nécessaire. La nécessité est en fait un compteur de références pour les shared pointers. Cette suppression peut être vue comme un delete/free automatique pour des pointeurs dynamiques de C/C++. Les templates surchargent les opérateurs de pointeurs du C++ et permettent donc de les utiliser quasiment de la même façon (nullptr au lieu de NULL...). Ils ont aussi d'intelligent, la possibilité d'empêcher la fuite mémoire grâce à une initialisation via le mot clé make_shared, qui soulèverait une exception en cas de mauvaises utilisations et qui a de meilleures performances.

Lorsqu'on déclare un pointeur et l'affecte à une variable correspondante, son compteur de références vaut un. Pour chaque variable se retrouvant liée à lui le compteur incrémenté d'une unité, et décrémenté si la variable se le voit désaffecté. C'est lorsque le compteur tombe à zéro que la suppression automatique opère. Ceci est très utile pour les arbres, lorsqu'on choisit un embranchement, les autres branches sont supprimées sans interventions particulières.

Le souci c'est que ce concept ne nous permet pas à lui seul de manoeuvrer comme nous le souhaiterions dans un arbre. En effet, on souhaite souvent regarder son prédecesseur/père, mais le simple fait de regarder constitue une référence, et nous ne voulons pas que le compteur de références augmente, cela créerait un cycle (référence circulaire) entre père et fils, et aucune suppression ne pourrait opérer. Entrent en jeu les weak pointers, ce sont en tout points des shared pointers à ceci près qu'ils ont la particularité de ne pas incrémenter le compteur de référence. Ainsi on pourra déclarer un weak pointer pred/parent pour utiliser des systèmes de rétro-propagation dans l'arbre.

5. Implémentation des algorithmes

Le développement logiciel se poursuit avec la troisième phase d'implémentation, ou écriture du code. Nous compartimenterons en deux sections de description du code, une pour chaque IA, et enfin une section commune de débogage, qui reprend des thématiques partagées de résolution des bugs sur l'interface de Codingame. Il est conseillé de suivre en parallèle le code concerné, pour apporter de plus amples précisions aux commentaires incrustés. Nous ferons donc une explication séquentielle. Nous débuterons par une description des structures de données propres à chaque code, ainsi que des statistiques associées. Le détail des structures, les signatures de méthodes/fonction, retours, effets de bord, ainsi que les attributs de classes sont disponibles en annexes6.1.

5.1 Minimax

5.1.1 Structures de données et statistiques

State et Node sont les deux seules classes ayant survécues au modèle commun, au début pensées comme une seule classe, mais un peu trop grosse, et avec des attributs qui n'avaient pas besoin d'être instanciés aux mêmes moments, il a été préférable de scinder en deux entités.

Ainsi, par exemple le socle de départ, le nœud root, qui n'utilise que quelques attributs, ne se voit pas affliger un état de jeu particulier.

Il n'a pas été utile de créer d'accesseurs, ni de constructeurs paramétrés, ni de surcharges. En effet Codingame est en soi une interface, l'encapsulation fournie par le paradigme objet est obsolète, voire ralentirait l'enchaînement des simulations.

Statistiques :

Personnel assigné : 3 membres

Module : 1 seul fichier comprenant tout le code, imposé par l'ide.

Lignes : 550-950 (selon les versions implémentées)

Classes : 2

Attributs : 18

Méthodes : 11

Fonctions : 4

5.1.2 Déroulement

Main

Avant le while :

La boucle while représente l'enchaînement des tours. Codingame ne lance qu'une seule fois notre main, ainsi ce qui se trouve avant le while est la partie d'initialisation.

Un nœud (Node) root est créé afin de servir de socle à un enfant unique (child0), l'objet courant qui appelle la méthode minimax. On profite du vector de shared pointer child_array déjà déclaré dans la structure de la classe, en se limitant à la création d'un seul élément, donc d'indice 0. L'objet courant n'est pas directement créé car celui-ci change au fur et à mesure de la partie, et il ne faut pas créer de conflits de références, comme par exemple supprimer les enfants de l'objet courant lorsqu'on transforme un de ces enfants en l'objet courant (voir smart pointer). Ce nœud sert aussi à stocker les valeurs renvoyées par son unique enfant (score), et pouvoir ainsi faire une comparaison au fil de l'itération de la boucle autour de minimax.

Un shared pointeur best_child sert de réceptacle à l'adresse de la meilleure simulation, et sera chargé comme objet courant au moment de jouer en fin de partie.

Row, col et nb_child sont des variables qui servent au fonctionnement de Codingame, pour éviter une désynchronisation dans l'ordre des entrées fournies. Cependant nous avons opté pour une génération propre à notre algorithme pour générer les données nécessaires.

Certaines implémentations contiennent des variables telles que first_turn, avec pour but par exemple de donner un comportement spécifique lors du tout premier tour, ou bien la combinaison d'avoir la main au premier tour, pour orienter un coup, gagner en temps, et donc en horizon.

Dans le while :

Le temps peut être sujet à certaines variations, alors il est préférable de ne pas définir de valeurs strictement égales à celles imposées par Codingame (1 sec et 0.1sec), c'est pourquoi celles-ci sont 0.98 et 0.095 secs, pour absorber une certaine marge d'erreur.

Le tout premier time_point pour le chrono se situe après le premier cin, car la boucle while continue en arrière-plan attendant que l'adversaire joue pour fournir l'entrée, le cin étant bloquant, le départ chrono de notre tour se situe après celui-ci.

Afin de faciliter les calculs pour l'évaluation statique, à base de modulos, il fallait convertir les données de Codingame en notre propre système de coordonnées, et comme l'évaluation est une méthode appelée très souvent et coûteuse, il a fallu factoriser la traduction des coordonnées dès le départ. Version et thème sont donc un couple de méthodes qui traduisent dans un sens et dans l'autre.

Il faut ensuite gérer la situation en fonction de qui a la main. Pour ce faire, on utilise le fait que si l'adversaire n'a jamais joué, ses derniers coups récupérés sont -1 et -1 , respectivement pour ligne et colonne, toujours dans cet ordre. Si ce n'est pas lui qui joue, on passe cette partie, sinon il faut impérativement actualiser l'objet courant par celui que l'adversaire a choisi. Pour cela on dispose de la méthode exist_children qui va chercher l'indice de l'enfant correspondant. Si celui-ci n'existe pas, et c'est logique au tout premier tour, on va charger les coups qu'il a choisi dans les attributs next_row et next_col qui sont à la base de la création des enfants via la méthode children_generate.

Une fois l'enfant créé avec l'indice 0 s'il n'existe pas, ou d'indice retourné par exist_children, on le transforme en objet courant (descendre d'un étage dans l'arborescence).

Ici nous anticipons la création des simulations en regardant s'il s'agit du premier tour, et si nous avons la main. Si c'est le cas, nous outrepassons (bypass) la création des 81 simulations qui devrait normalement arriver, et en choisissons directement une que nous avons sélectionnée (selon nos heuristiques). Cela a pour but donner à notre bot une marge d'avance sur l'horizon. Il pourra disposer pleinement de la seconde du premier tour, pour parcourir un horizon trois fois plus grand.

Si nous nous trouvons aux tours suivant ou que nous n'avons pas la main au premier tour, il faut donc ensuite savoir qu'elles vont être les prochaines simulations. Normalement fournies par Codingame, nous utilisons plutôt un vector pair de coordonnées afin de ne garder strictement que les coordonnées qui nous intéressent car au fil des récursions nous en élagueront certaines. Ce vector est un attribut de la classe nœud, playable_next. Cette méthode n'est lancée que si celui-ci est vide, pour éviter d'une part de la lancer dans les récursions, et ensuite dans le main, et pour éviter que les coups élagués ne soient ensuite créés dans le main en écrasant la sélection pré-établie.

Toujours sur l'élagage, pour rendre le pruning alpha_beta plus performant, il faut ordonné les simulations qui vont être étudiées. Au départ nous avions mis en place un tri alterné en fonction du type de nœud, mais il s'avère qu'ordonner à chaque fois en premier les coups que notre heuristique qualifie de plus mauvais, va provoquer l'élagage le plus effectif. Le tri travaille de pair avec l'évaluation, qui le confirme ou l'infirme.

Une variable h pour horizon, dont la valeur est arbitrairement ajustée en fonction

des différentes implementations. Elle peut être substituée par une durée (duration) de la librairie chrono, être utilisée en combinaison du dit chrono, ou bien calculée à partir du temps restant et d'autres facteurs comme le nombre de simulations restantes. C'est la condition d'arrêt principale de la récursion du minimax. Deep Blue avait un horizon supérieur à 12, tandis qu'un débutant a un horizon d'environ 4, et un maître de 8.

Notre IA a son meilleur potentiel avec h valant 2. C'est peu, et cela semble même contre intuitif qu'un h de 3 ne soit pas meilleur. Mais l'explication réside peut-être dans le fait que nous affrontons une majorité de bot MCTS, qui ne disposent pas des mêmes heuristiques. De fait, plus nous regardons loin, plus nous consacrons de ressources vers une simulation qui ne sera pas du tout envisagé par l'adversaire. On peut aussi envisager que nous déstabilisons l'adversaire en jouant plus rapidement que prévu. En effet il est possible, bien que sûrement marginal, d'utiliser le temps de notre adversaire pour calculer des simulations en arrière plan, via des threads, forks etc. En écourtant notre phase de réflexions, nous écourtions la leur, les rendant donc plus mauvais.

Enfin dernière étape avant de rentrer dans la boucle qui parcourt les simulations, rétablir les infinis correspondant au score et à alpha. Pour bête nous y procédon dans la boucle, car cette boucle à la particularité d'être un nœud max.

Boucle autour de minimax

Si l'on restreint l'étude à la boucle en elle-même, il ne s'agit que de marquer le temps avec un autre time_point (pour les implémentations où nous l'utiliserions), charger les coups extraits de playable_next dans les attributs next_row et next_col de l'objet courant, lancer minimax sur l'objet courant actualisé, comparer le score (moins infini au premier tour de boucle) avec celui de la simulation (qui se trouve dans l'attribut score du fils créé dans la simulation) et enfin d'actualiser le score de l'objet courant et enregistrer l'adresse du fils de cette meilleure simulation (dans best_child) si bien sur le score est supérieur à celui stocké. La meilleure car l'objet courant est un nœud max, cherchant le plus gros score, d'où la comparaison avec moins l'infini au départ.

Fin de tour

Terminons l'étude du main avant de nous attarder sur minimax.

À la sortie de la boucle, toutes les simulations qui nous intéressaient ont été étudiées, on peut donc procéder à la descente dans l'arborescence, le transfert de l'objet courant (père) vers son fils. En arrière-plan, cela réduit le nombre de références pointeur du père à zéro et va à son tour réduire de un vers zéro toutes les références de ses fils et donc les supprimer, sauf celui qui nous intéresse, conservé (sauvé) par la référence de best_child.

Nous appliquons la méthode thème pour traduire de nos coordonnées vers celles de Codingame, avant bien sur de procéder à l'instruction de la sortie standard, signifiant la fin de tour et le retour au début du while.

Méthode minimax

Pré cas d'arrêt

Contrairement au schéma général d'une fonction récursive, la première action de minimax n'est pas de vérifier le cas d'arrêt mais de créer le fils portant sur la simulation étudiée, c'est très important car celui-ci sert à transmettre vers le main l'évaluation de la simulation. C'est aussi une façon de cristalliser la simulation, car les implémentations précédentes chargeaient un coup, l'évaluaient et le défaisaient, et c'était très coûteux en sauvegarde/copie d'états, et surtout source de nombreux bugs.

On reprend donc les mêmes éléments que dans le main mais adaptés pour la récursion. On cherche si la simulation existe déjà avec `exist_children`. Rappelons que minimax est lancé sur le père, et non pas sur le fils qui va être créé juste après. Cela permet d'avoir une visibilité sur toute la fratrie et chercher l'indice de la simulation concernée et de comparer les frères entre eux. Si celui-ci n'existe pas (indice -1) alors celui-ci est créé. Avec `playable_next_generate` on calcule les simulations du fils nouvellement créé. Enfin on tri se vector de simulation. On pourrait appeler tri dans `playable_next_generate`, mais comme c'est une implementation tardive et que `playable_next_generate` est déjà très grosse, et avec effets de bord, laissons les séparées. On pourrait décider plus tard d'effectuer des tris différenciés, si on trouve de meilleures heuristique, en fonction du type de noeud.

Le cas d'arrêt

Il y a deux voire trois conditions d'arrêt selon l'implementation. Le premier étant si le fils nouvellement créé n'a pas de descendance, soit parce que l'arbre touche à sa fin (fin de partie), soit parce que soumis à un élagage. Le deuxième et troisième sont des cas d'arrêt de l'horizon, soit via `h` soit via le temps. Les récursions successives de minimax décrémentent d'un l'horizon `h`, et lorsque celui-ci est nul, on procède à l'évaluation. Les quelques lignes de gestion d'alpha/beta dans ce bloc ne sont là que pour gérer le cas particulier d'un lancement de minimax dans le main avec `h=0`. En effet nous pourrions décider de ne pas octroyer de visibilité, ou bien si la méthode `possible_horizon()` renvoie 0. C'est une méthode inconsistante, selon nos implémentations, qui se charge de calculer en fonction du temps et du nombre de simulations, quel serait l'horizon `h` adéquat.

L'évaluation statique

On commence par vérifier que le jeu n'est pas terminé, via l'attribut `win`, dans le cas contraire on retourne tout simplement l'infini signé par l'identité du gagnant (-1 pour l'adversaire, sinon 1).

Voici quelques-unes des heuristiques implémentées :

- Posséder un coin a plus de valeur que le centre, qui a plus de valeur que les bords, par contre quand il s'agit de posséder un morpion, celui du centre est le plus avantageux, suivi des coins, et enfin les bords. Pour mettre cela en place, une matrice statique de noeud est utilisée pour comptabiliser une à une les pondérations. Le morpion choisi

est aussi représenté dans d'autres calculs comme étant le coefficient (coef).

- Posséder/Céder un morpion.
- Aligner des coups pour de possibles victoires. On peut aligner deux croix sur une ligne, colonne ou diagonale, et si la dernière case est libre alors cela représente une possibilité de s'approprier un morpion. De même avec les morpions, si deux sont alignés cela représente la possibilité de gagner le jeu, ou de le perdre.
- Combien de fois l'adversaire à l'opportunité de choisir dans tout le plateau de jeu et non pas un morpion particulier.

Certaines heuristiques ne sont intéressantes qu'en début de partie, c'est pourquoi certaines implémentations comprennent un attribut `game_turn` pour lancer un type de calcul en fonction de l'avancement de la partie. Par exemple pondérer les cases nous intéresse fortement au début du jeu pour occuper les coins/centres alors qu'en milieu/fin de partie on privilégie les alignements.

La récursion

Comme pour le main, il faut réinitialiser un score infini selon le type de nœud (caractérisé par `player_turn`), puis utiliser une boucle qui parcourt les simulations possibles (dans `playable_next` du fils de l'instance appelante). On charge les coups de la même façon depuis le vector de coordonnées, on transmet les valeurs alpha et bêta pour que l'élagage ait lieu, et on lance la récursion sur le fils.

Selon le noeud, on voudra actualiser alpha ou bêta, comptabiliser le nombre de fois que l'adversaire a eu le choix libre dans toute la grille (heuristiques de eval) dans le cas du coup adverse, et mettre à jour le score du père pour le comparer avec les prochaines itérations, enfin si la condition d'élagage survint on arrête prématurément la boucle.

5.1.3 Performance et complexité

Le main de l'algorithme minimax n'appelle que des méthodes/fonctions constantes et linéaires, avec une taille d'entrée le plus souvent inférieure à 9. Seul le tri peut se retrouver quadratique en cas de réallocation de l'espace, mais c'est une implémentation tardive qui n'a pas servi à atteindre les dernières places atteintes, donc nous n'en tenons pas compte. L'affichage n'est pas pris en compte non plus, certes en pratique dans l'IDE il consomme la ressource temps octroyée à chaque tour, mais nous les désactivons avant un passage dans l'arène.

Reste donc la méthode minimax, qui dépend de l'horizon, du temps restant, et du nombre de simulations possibles. Seuls, aucun de ces paramètres n'est en mesure de poser un problème de performance, mais combinés, le nombre de cas à étudier devient très vite impossible à calculer en temps humain.

Un algorithme minimax lambda se contente de calculer à partir d'un état, le coup qui dans h coups avantagera le plus le joueur. Le minimax que nous avons mis en place, ne va pas juste simuler, et passer à la simulation suivante, il va réellement stocker l'arborescence. L'idée est que parcourir un nœud dont on connaît déjà un certain horizon, va faire bénéficier d'une avance, qui va permettre d'avancer encore un peu plus jusqu'à atteindre la fin de partie.

Prenons un h qui vaut 2, l'algorithme créera le nœud courant $h=0$, le nœud fils $h=1$ et le nœud petit-fils $h=2$. Après avoir joué et l'adversaire aussi, l'arbre possédera donc déjà le petit-fils, il aura bien donc un coup créée d'avance.

Grâce à l'arborescence, et donc la conservation des nœuds, on peut choisir d'augmenter l'horizon si on a déjà connaissance d'une simulation, et ainsi gagner petit à petit beaucoup d'avance. C'est pourquoi même un petit horizon $h=2$ est capable de se hisser parmi les plus robustes, car en gardant en mémoire toute la largeur de l'arbre il est sûr de ne jamais tomber sur une situation qui lui est inconnue. Ce n'est pas le cas si on stop la récursion à cause du temps restant. En effet l'algorithme peut être stoppé prématurément, et toute la largeur (le nombre de simulations d'un nœud) peut ne pas être parcourue. Si par malheur c'est un coup non parcouru que l'adversaire choisi, alors toute la visibilité sera perdue (reset de l'arbre, et donc visibilité réduite à zéro).

Cependant cette implémentation est extrêmement difficile à mettre en place. Premièrement, il faut éviter que le nombre de simulation s'emballe. En effet, plus on va en profondeur, plus vite l'ordre de grandeur du nombre de simulation frôle les factorielles. On ne peut pas simplement dire "si fils connu, alors profondeur plus un". Il faut contrôler avec le temps, qu'on va avoir assez de temps pour remonter la récursion. Autre souci, il faut absolument que tout les enfants d'un même niveau de profondeur ait le même horizon d'évaluation, sans quoi on comparerait deux nœuds qui auraient des descendances de degrés différents, et la comparaison n'aurait plus aucune valeur (nombre de cases pondérées différents, ou comparer un nœud min avec un nœud max). C'est pourquoi nous avons entamé l'implémentation du chrono de cette façon dans notre code, avec au dénominateur le nombre de simulations à étudier, pour partager équitablement le temps dans une fratrie.

En définitive, h ne serait plus un nombre arbitraire que le programmeur fixerait, mais serait entièrement calculé à partir du temps restant divisé par le nombre de simulation. Ce que proposerait de faire la méthode possible_horizon. Mais c'est une implémentation que nous n'avons pas pu mettre correctement en place.

Il en résultat que l'élagage, bien que parfaitement opérant, n'est pas utile. Avec un horizon constant, non contrôlé par le temps, il est inutile d'en gagner avec l'élagage. De plus tant que nous n'aurons pas élucidé pourquoi un h valant 2 est plus performant qu'un h valant 3 face à des bots en majorité MCTS, il n'est pas souhaitable de laisser l'aléa décider de la valeur de l'horizon.

Maintenant, si nous poursuivons le raisonnement précédent. Avec un élagage efficace (double l'horizon), et un horizon ajusté par rapport au temps ; nous pourrions obtenir

un calculer et conserver quatre à six coup d'avance par tours. Ajouté à cela le premier coup qui fait disposer d'une seconde ; au bout de quelques tours seulement nous connâtrions sur l'ensemble des coups possibles, un horizon plutôt conséquent. Contrairement à un MCTS qui dispose d'une largeur très minime, nous pourrions profiter de ces coups d'avances pour terminer le jeux sur les situations les plus favorables, à la façon d'un MCTS.

Rappelons toutefois que l'élagage qui permet de doubler l'horizon peut aussi le réduire à néant, si les heuristiques divergent, et que l'adversaire nous avantage selon nos critères. C'est pour cela qu'il est souhaitable après quelques coups, de se débarrasser de l'évaluation, pour passer en mode MCTS. C'est comme ça qu'était prévu notre algorithme combiné.

5.1.4 Progression et comparaisons

Les balbutiements

La première implémentation du minimax s'est faite sur celle du morpion, sans classe. Après le passage en ligue bronze (méta-morpion) nous avons de suite été confronté au changement de coordonnées. Il a fallu créer des structures capables de simuler, et prévoir des mécanismes pour rétablir le plateau en phase de pré-simulation, pour lancer la suivante. C'était soit ça, soit créer autant de structures que de simulations. Du fait de l'absence de classe, la transmission des valeurs se faisait via les paramètres, en référence, et pour atteindre les fonctions les plus profondes depuis les étages supérieurs, nos fonctions avaient une dizaine d'arguments, et servaient de passeurs.

Éventuellement, une fois les concepts de minimax pris en main, l'interface du site, et le C++ revisité, nous avons recommencé l'implementation de zéro, pour intégrer les classes, et le modèle commun conçu alors. Cette première version minimax avait réussi à atteindre la 2nde places de la ligue bronze.

Confirmation

Les implémentations qui suivirent n'eurent pas beaucoup plus de succès, il a fallu beaucoup de temps pour se rendre compte que les valeurs étaient mal transmises entre les différents niveaux de récursions, où encore que le retour de fonction arrêtait prématurément la boucle sans tester toutes les simulations. De plus nous avons implanté à plusieurs reprises l'élagage (pruning) alpha bêta, mais celui-ci compliquait considérablement la tâche de débogage. Enfin nous avons opté pour notre propre génération de simulations, en dépit des entrées déjà fournies par Codingame, dès le premier tour, dans le main, afin de supprimer l'aléa et faciliter le débug. Il a aussi fallu se séparer momentanément de la fonctionnalité d'évaluation, afin de renvoyer un résultat toujours identique, sans variation, et parfois il a fallu plutôt utiliser un random.

Ce n'est que quand toutes ces fluctuations étaient écartées/contrôlées, que nous avons pu nous focaliser sur le noyau, et le corriger, notamment pour ce qui est de la transmission des valeurs et les erreurs que provoquaient les simulations sur les données de départ. Une fois le noyau opérationnel, et surtout après avoir modifié la méthode minimax pour qu'une

simulation étudiée soit une simulation créée, et ainsi ne plus avoir à gérer un grand nombre de rétablissement de données sauvegardées, le bot a pu se hisser en ligue argent. Enfin après avoir rétabli l'évaluation statique, et quelques erreurs supplémentaires sur la transmission, et la génération des simulations, le bot a bondi en ligue or.

Après son entrée en ligue or, nous avons rétabli l'élagage, et tenté d'implémenté le temps à l'équation pour rendre ce dernier utile. Nous avons opéré à la façon de tests d'intégrations, en reprenant le code qui nous avait permis de passer en ligue or, et y ajoutant une fonctionnalité après l'autre, et les corrigeant. Le bot n'était pas meilleur, mais il n'était au moins pas plus mauvais. On y ajoutait des fonctionnalités (tri, élagage, possible_horizon...) pour préparer l'étape suivante, et peut-être ensuite une fusion avec MCTS.

5.2 Monte-Carlo

5.2.1 Structures de données et statistiques

— Les différentes classes :

Pour intégrer un modèle commun et permettre une réutilisation de notre code pour le développement d'autres algorithmes (précisé plus haut notamment avec la générativité), nous avons dû implémenter un système de classes. Ainsi notre algorithme s'articule autour de plusieurs classes et structures de données nécessaires à son fonctionnement et dont il est totalement dépendant. L'UML vu précédemment décrit l'interaction et le lien entre ces classes. Elles possèdent chacune leurs attributs et méthodes respectifs. Certaines sont des classes abstraites, et/ou des classes template.

— Ensembles de fonctions pour le MCTS :

Là est le cœur de l'algorithme, nous avons une fonction principale MonteCarloTree-Search, orchestrant le déroulé précis et appelant les différentes fonctions associées à chaque phase de l'algorithme. Ces dernières effectuent ce pour quoi elles existent, remplissent leur rôle avant de renvoyer des valeurs

Statistiques :

Personnel assigné : 3 membres

Module : 1 seul fichier comprenant tout le code, imposé par l'ide.

Lignes : 970-998 (selon les versions implémentées)

Classes : 7

Attributs : 13

Méthodes : 43 (en comptant les accesseurs/mutateurs)

Fonctions : 6

5.2.2 Déroulement

Comme expliqué plus haut, le Monte-Carlo Tree Search (abrégé MCTS) est un algorithme de recherche heuristique, donc un algorithme qui fournit rapidement une solution

possible, sans pour autant que cette solution soit la meilleure ni la plus optimisée.

Le principe derrière le MCTS est une recherche dans un arbre contenant des nœuds qui représentent chacun une configuration possible du jeu ou de l'application dans laquelle est utilisé le MCTS, avec la racine (nœud en haut de l'arbre, n'ayant pas de nœud(s) parent(s)) comme configuration initiale, et les feuilles (nœuds tout en bas de l'arbre, n'ayant pas de nœud fils) comme configuration finale ou comme configuration n'ayant pas été explorée jusqu'au bout.

Cet algorithme se décompose en 4 phases principales :

— La phase de Sélection :(voir premier arbre sur le schéma ci-après)

Durant cette phase, l'objectif sera de sélectionner un nœud feuille prometteur. Pour cela on va descendre progressivement dans l'arbre en prenant soin de choisir à chaque fois, le « meilleur » nœud fils du nœud sur lequel on est actuellement.

Pour déterminer qui est le « meilleur » nœud fils, on applique une formule nommée UCT (Upper Confidence Bound applied to Trees) qui va, en prenant en compte, leur chance respective d'aboutir à une victoire, et leur nombre respectif de fois où on les a visités, en sélectionner un. On répétera l'opération en appliquant à nouveau l'UCT sur ses fils, et ainsi de suite jusqu'à arriver en bas de l'arbre.

Cette formule permet de mettre en valeur les nœuds victorieux mais aussi les nœuds n'ayant pas été beaucoup visités, pour ne pas laisser de côté un nœud qui pourrait potentiellement par la suite, devenir victorieux.

— La phase d'Expansion :(voir deuxième arbre sur le schéma ci-après)

Une fois un nœud feuille choisi, nous allons dans cette phase l'étendre en construisant toutes les nœuds fils/configurations possibles à partir de celui-ci. Dans l'exemple du métamorpion, on va construire, un nœud fils par coups possibles à partir de la grille actuelle.

— La phase de Simulation : (voir troisième arbre sur le schéma ci-après)

Ensuite, on choisit un de ces nœuds fils au hasard (ou selon la méthode voulue), et on va simuler l'application de façon aléatoire jusqu'à son terme. C'est-à-dire pour un jeu, exécuter la partie jusqu'à ce qu'un des joueurs gagne ou qu'il y ait un match nul.

— La phase de BackPropagation : (voir quatrième arbre sur le schéma ci-après)

Une fois cette simulation terminée, on remonte dans l'arbre jusqu'à la racine en actualisant les scores de chaque nœud : on incrémente le nombre de visite, et si la simulation nous indique une victoire, on incrémente également le nombre de victoire.

Enfin, on renvoie le nœud ayant le meilleur score parmi les fils de la racine, car c'est

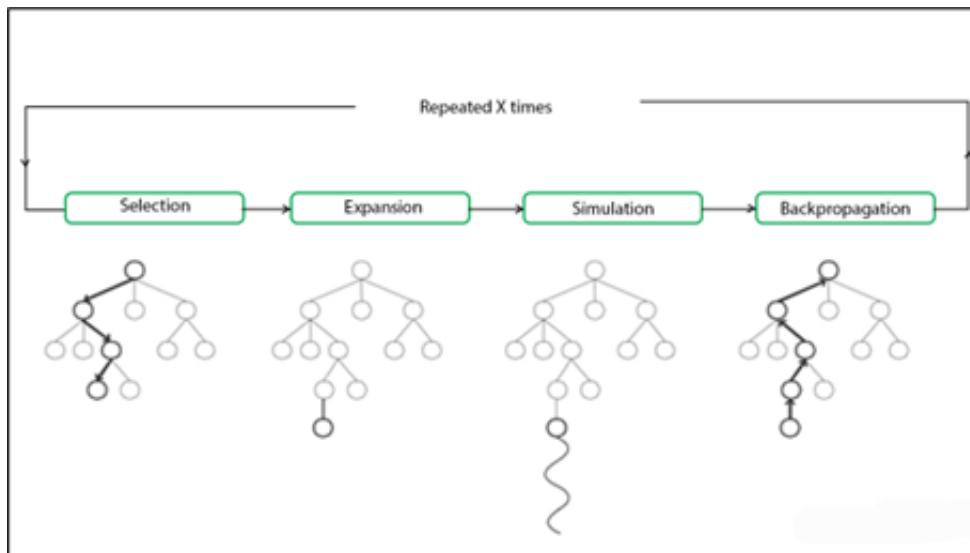


FIGURE 5.1 – [Schéma récapitulatif de représentation du fonctionnement du Monte-Carlo Tree Search]

« techniquement» la configuration qui a le plus de chance, pour le coup suivant, de nous amener vers une victoire.

Le MCTS est un algorithme efficace, qui permet de toujours faire valoir un compromis entre les choix qui ont l'air prometteurs (grande chance de victoire) et les choix qui ont été peu explorés, desquels peu de simulations ont été effectuées.

Cependant, pour que le MCTS donne un résultat exploitable, il faut en exécuter un grand nombre, ainsi , on aura un arbre assez grand et des coefficients (nbr. victoires/nbr. visites) plus précis sur chaque noeud.

Notre programme se déroule comme suit :

Il s'articule autour de la boucle de jeu, boucle qui se lance à chaque fois que c'est à notre tour de jouer. En dehors de cette boucle, nous n'avons que quelques déclarations de variables, et notamment du noeud racine initial pour notre arbre. A l'intérieur de cette boucle, on commence par gérer la réception des entrées, c'est-à-dire la réception des coordonnées du dernier coup de l'adversaire. On s'occupe ensuite de vérifier si l'arbre existant (1) contient déjà un noeud avec comme dernier coup, le coup de l'adversaire. S'il existe (2), on déclare ce dernier comme la nouvelle racine actuelle. S'il n'existe pas, on le génère et l'arbre existant est détruit car invalide.

1. On conserve l'arbre généré par le MCTS entre chaque tour, pour gagner du temps et éviter de tout refaire à chaque fois.
2. Ce qui est quasiment toujours le cas car on a déjà générée l'entièreté de la partie de l'arbre qui nous intéresse lors des tours précédents.

On lance ensuite, dans une boucle, l'algorithme MCTS sur la nouvelle racine, et on boucle tant qu'on est encore en dessous de la limite de temps par tour imposée par la plate-forme. Après cette boucle, on sélectionne le noeud fils ayant le meilleur score parmi ceux de la racine, et on renvoi les coordonnées associées. Celui-ci correspondra au coup censé être le meilleur coup possible calculé par notre algorithme à ce moment-là du jeu.

5.2.3 Performance et complexité

Nous pouvons remarquer que notre MCTS va être de moins en moins coûteux au fur et à mesure que la partie avance et que nous réduisons le nombre de cases restantes.

Cela peut être expliqué par le fait qu'une des fonctions les plus coûteuses dans notre algorithme est `getValidActions`, qui va chercher les prochains coups possibles en fonction du dernier coup. Pour le premier coup, on utilisera une double boucle de $9 * 9$ itérations pour les 81 coups qui sera plutôt coûteuse.

Sinon on aura deux cas : soit le morpion dans lequel on doit jouer n'est pas fini auquel cas les coups possibles sont les cases restantes dans ce morpion, ou alors ce morpion est terminé et les prochains coups possibles sont toutes les cases du méta-morpion vides. Cette dernière situation est la plus coûteuse mais coûte de moins en moins cher au fur et à mesure de l'avancement de la partie car il y aura moins de cases à ajouter.

Cela rend donc l'expansion et surtout la simulation très coûteuse car ces parties doivent chercher les prochains états possibles en utilisant `getValidActions`. La simulation doit en plus faire cela pour simuler chaque coup jusqu'à arriver à la fin de la partie. La simulation coûtera donc moins cher au fil de l'avancement de la partie puisqu'il y aura de moins en moins de coups à simuler. C'est pour cela qu'il a été suggéré de fusionner minimax pour orienter en début de partie de façon moins coûteuse, et de terminer avec MCTS.

La sélection est moins coûteuse, elle nous fait descendre en bas de l'arbre du MCTS, on utilise une double boucle, pour sélectionner le meilleur des fils de chaque noeud jusqu'à arriver à une feuille.

La rétro(back) propagation est d'autant moins coûteuse puisque l'on doit juste sélectionner le père de notre noeud jusqu'à atteindre la racine, donc seulement une boucle.

5.2.4 Progression et comparaisons

Nous avions convenu à nos premières réunions et après avoir effectué des recherches que MCTS paraissait plus prometteur que minimax pour le méta-morpion. Il s'avère que minimax obtient de meilleurs résultats en pratique et un meilleur classement dans Codingame.

Notre algorithme MCTS a atteint la 2e position de la ligue de Bronze. On voit que notre algorithme arrive à aligner des coups et à gagner des morpions et même des parties, même si des fois il ne joue pas des coups qui pourraient faire apporter la victoire.

5.3 Débogage

La plate-forme Codingame n'est pas la plus adapté pour debuger nos bot. Elle impose de fournir le code en un seul module de plusieurs centaines, voire un millier de lignes, ainsi que des des contraintes de temps pour les tours de chaque joueur. Lorsqu'une erreur est signalé, on obtient très peu d'information (au nombre de trois). L'affichage étant décompté de notre temps, et limité à un maximum de caractères dans la sortie erreur, nous devions être très astucieux pour dénicher les erreurs, afficher juste la bonne dose d'informations, et au bon moment. C'était l'activité la plus chronophage, et de par l'environnement, la moins satisfaisante car nous effacions le plus souvent nos display de débogage. Pour une implémentation qui auraient eu lieu hors codingame nous aurions plutôt écrits des tests unitaires, eux réutilisables.

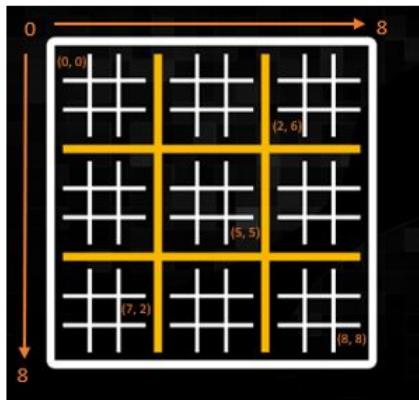
Il nous est arrivé de sortir de l'interface Codingame car le débogage devenait trop ardu. Nous devions ainsi créer à la façon de tests unitaires, des jeux de données à donner en entrée du programmes, de la même façon que Codingame. Ensuite, nous insérions des sorties standard (cout) que nous redirigions dans un fichier grâce à la magie des chevrons » du terminal linux. La sortie erreur, elle, n'est pas redirigeable, elle apparaît automatiquement dans le terminal. Nous pouvions ainsi dissocier des affichages pour une plus grande précision de débogage. Le hic étant que ce code est complètement inutilisable en l'état pour codingame, du fait de cout, qui doit être unique.

Lorsqu'on est à stade avancé du projet et que beaucoup de fonctionnalités ont été ajoutées, il nous est arrivé de perdre des places dans le classement alors qu'on aurait dû en gagner avec ces améliorations. Il faut donc revenir au code de base en ajoutant chaque fonctionnalité une par une et en vérifiant à chaque fois qu'elle soit performante, grâce à la fonctionnalité 'rejouer dans les mêmes conditions' on peut s'assurer que ce qu'on vient d'ajouter améliore ou pas le code (test d'intégration). Cela n'est vrai que pour les ligues inférieures à or. Pour utiliser cette fonctionnalité pour déboguer le minimax en or, il fallait le faire s'affronter lui-même pour que le temps n'intervienne pas, et donc que les simulations soient identiques.

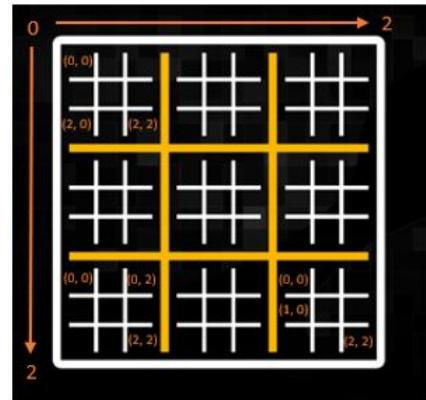
Le but est de faire le plus de simulations par tours, ceci nous mena donc à essayer de faire le code le plus rapide et efficace possible. Nos efforts ont permis de passer de quelques dizaines de MCTS par tour, à plusieurs centaines et même plusieurs milliers en fin de partie. Grâce notamment à une meilleure gestion de la boucle while MCTS, conditionnée par un chrono vérifiant de ne pas dépasser le temps limite. Pour minimax, l'optimisation se répercute sur un horizon plus grand.

Codingame impose un format pour les entrées et sorties, en effet, la plate-forme voit la grille du méta-morpion comme une seule et même grille de 81 cases, et note les coordonnées des cases comme suit : ($x = [0, 8]$, $y = [0, 8]$) (voir illustration ci-dessous, à gauche). Or, notre structure de classes pour MCTS étant basée sur une « méta » grille de 3 par 3, contenant dans chacune de ses cases, une grille de Morpion classique, elle-même de 3 par 3 (voir illustration ci-dessous, à droite). Tandis que pour minimax la lecture se fait avec une matrice à deux dimensions, la première numérotant le morpion, la seconde sa case. Il y avait donc un problème de concordance des coordonnées. Ceci nous enjoint à convertir à

plusieurs reprises dans le code, les coordonnées, que ce soit dans un sens ou dans l'autre.



[Structuration des coordonnées de la grille par [Coding Games](#)]



[Structuration des coordonnées de la grille et des sous-grilles par notre algorithme (grille principale 3x3 contenant des grilles de 3x3)]



Pour la fonction gérant la phase de simulation (rappel : l'objectif de cette phase est de simuler à partir d'un nœud choisi, le reste de la partie aléatoirement jusqu'à son terme, et ensuite de faire disparaître ces coups car ils ne sont que temporaires) ; Notre première tactique a été de générer un nœud pour chaque coup (fils du nœud du coup précédent) jusqu'à la fin de la partie. Et une fois le statut de fin récupéré, nous détruisions ces nœuds.

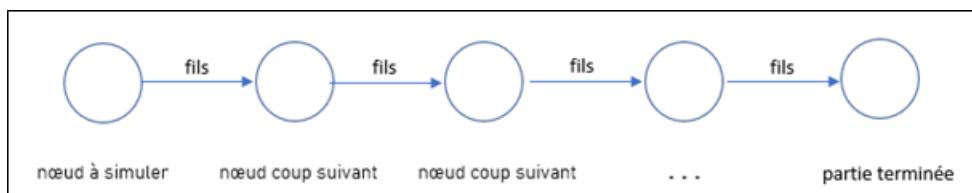


FIGURE 5.2 – Boucle MCTS principale

Cependant cette méthode prenait beaucoup trop de temps, nous sommes donc passé à une autre façon de faire : dans le nœud ciblé par la simulation, nous générerions aléatoirement le reste de la partie tout en mémorisant les coups simulés dans une liste, pour pouvoir ensuite les « annuler » (undo) et ainsi revenir à l'état antérieur du nœud avant simulation.

Mais encore une fois, nous devions trouver une technique plus rapide, nous avons opté finalement, pour la méthode la plus simple mais qui nous avait échappée, celle de prendre une copie de la grille à simuler, de faire nos modifications dessus, et une fois le résultat

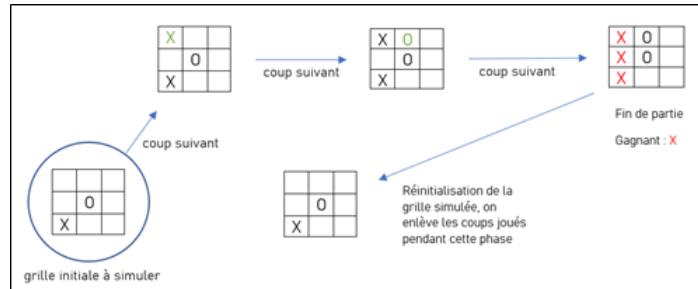


FIGURE 5.3 – Schéma explicatif de notre deuxième approche du problème

acquis, détruire entièrement cette copie. Ceci évite de gaspiller des ressources dans la gestion des coups à annuler.

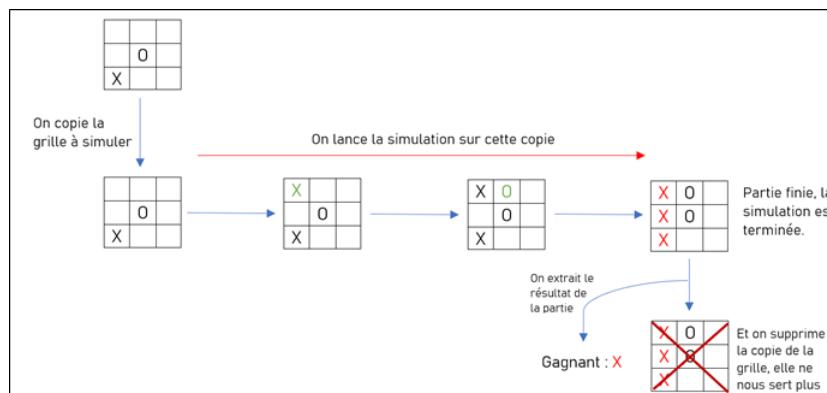
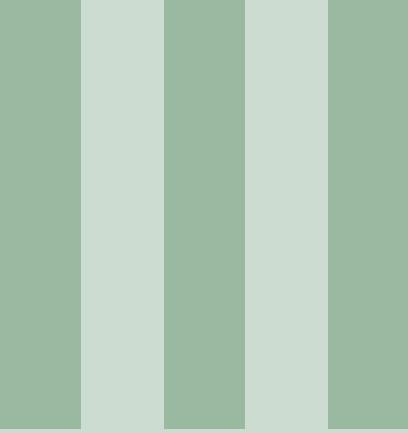


FIGURE 5.4 – Schéma explicatif de notre deuxième approche du problème

Les fonctions servant à renvoyer les coups possibles et valides ont été revues, en effet, nous les avons raccourcies, le coût des opérations sur celles-ci a donc été réduit.



Conclusion

6. Bilan

Difficultés

Au fur et à mesure de l'avancement de notre projet nous nous sommes heurtés à quelques difficultés. Premièrement, la nécessité de nous réapproprier le langage C++, notamment pour la Programmation Orientée Objet Avancée ou encore le concept de pointeurs intelligents qui se sont révélés être très importants dans notre projet pour un gain de mémoire. Étant habitué à utiliser le langage de programmation Java, passer au C++ qui est un langage de plus bas niveau et donc par définition moins facile à appréhender, n'a pas été une mince affaire. Mais nul doute que cela nous bénéficiera grandement dans nos futurs projets.

Deuxièmement, la conception. Aucun de nous n'a l'expérience pour concevoir à l'avance la totalité d'un projet de cette ampleur, ni prévoir les soucis d'implémentation qui amèneront à des modifications des structures de données, ou des changements en profondeur des noyaux de nos algorithmes. Il a donc fallu plus souvent ajuster l'UML par rapport au code que le contraire. L'expérience vient en pratiquant.

L'avancement de notre projet fut aussi ralenti par pas mal de bug sur le noyau qu'il était difficile de dénicher, de par les possibilités qu'offre l'IDE, c'est pourquoi nous avons migré sur un terminal Linux plus d'une fois.

Enfin en terme de difficultés logistiques, elles furent nombreuses. Nos uniques créneaux libres pour nous réunir à six et travailler ensemble, étaient systématiquement réquisitionnés par des cours déplacés. De plus nous ne disposions pas d'un environnement adéquat en libre accès. Il a donc fallu pour les managers, redoubler d'effort et de sollicitations pour faire fonctionner le projet, par exemple transformer le manque de matériel à disposition en l'occasion idéale pour pratiquer le codage en binômes. De plus, Codingame a organisé

un tournoi au cours d'avril, il était donc très compliqué de tester son code dans l'arène, puisque les serveurs étaient très sollicités.

Fonctionnalités

Revenons sur les points du cahier des charges. Les pointeurs intelligents ont été compris et utilisés, bien sûr il s'agit d'une notion bien plus complexe que ce que nous présentons dans le projet, mais nous n'avions pas l'utilité de la plénitude du concept, que sont les unique pointeurs par exemple. Pour la généricité, les classes ont été conçues et implémentées pour l'algorithme MCTS. Le code utilise des classes, et le C++. Enfin, les deux modèles algorithmiques ont été employés pour répondre au problème énoncé, avec des succès différents. Enfin, pour ce qui est de battre le bot du référent, c'est arrivé (voir annexe)6.6, tout comme battre le boss de ligue (voir annexe)6.7. Mais ce sont des issues, rare, et il faut lancer plusieurs fois la partie pour espérer tomber dessus, dû à la différence de classement (350ème (nous) VS 110ème (référent) / 1er (boss)).

Perspectives

Afin d'améliorer notre projet, nous avons plusieurs voies à explorer. En effet déjà nous pouvons continuer d'optimiser notre code afin d'appliquer plus de fois notre algorithme dans le laps de temps donné à chaque tour. Nous pourrions aussi combiner notre algorithme MCTS et celui minimax, notamment en utilisant minimax dans la partie simulation du MCTS qui pour l'instant est aléatoire, ce qui rendrait notre algo d'autant plus efficace. Ou tout simplement commencer avec minimax et terminer en MCTS. Nous pourrions aussi utiliser la programmation multithreads pour paralléliser les calculs. Pour les deux groupes, l'implémentation du noyau qui gère les simulations a demandé beaucoup de temps et d'effort, et n'en a laissé que trop peu pour une étude poussée des heuristiques d'une évaluation statique ultra-performante.

Enfin, nous avions suggéré que nous pourrions aussi créer nos propres règles de métamorpion, par exemple faire s'affronter des bots qui cherchent la défaite, des grilles de dimensions non canoniques, et même des couches superposables pour jouer par dessus une case déjà occupée.

IV

Annexes



Annexes

Structures de données et statistiques - minimax

Attributs

State state;	Instance de l'état du jeu (simulation), appartenant à un noeud particulier(fils).
vector<shared_ptr<Node>> child_array;	Vector (structure dynamique) accueillant des shared pointers. Celui de root n'accueille qu'un seul élément (child0) qui lie l'instance courante du main. Les vectors des instances appelant minimax sont remplis directement dans minimax au fur et à mesure des appels successifs à children_generate.
vector<pair<int,int>> playable_next;	Vector de coordonnées, remplis par playable_next_generate, la méthode qui calcule les simulations prochaines de l'instance appelante.
int score = 0;	Accueille le score de l'évaluation statique, et sert à communiquer entre l'intérieur de la récursion et le niveau supérieur.
bool free = false;	Indique si le noeud à la possibilité de choisir un coup dans tout le métamorpion, sert ensuite à incrémenter l'attribut free_choice_count de son père.
int alpha;	Attribut qui va servir à transmettre d'une simulation à une autre, sans conflit avec des références/copies, les valeurs nécessaires à la comparaison aboutissant à l'élagage.
int beta;	Idem.

FIGURE 6.1 – Les attributs de Node

Méthodes et fonctions

int win=0;	Indique l'état de la partie, 0 pour aucun gagnant, 1 si le joueur gagne, -1 si l'adversaire gagne, 2 si ex aequo.
int last_col=-1;	Indique quelle a été la dernière colonne jouée. Est quasi-constant, le mot clé const ne figure pas car le -1 sert à savoir si c'est le premier tour de l'adversaire. Mais ce n'est pas un attribut qui change comme peut l'être next_row/col.
int last_row=-1;	Idem mais pour la ligne.
int next_col=-1;	Variable qui va accueillir la coordonnée colonne de la prochaine simulation de l'instance appelant minimax. On y charge une valeur qui servira à la création du fils.
int next_row=-1;	Idem pour la coordonnée ligne
int game_turn;	Indique à quel tour de jeu le nœud se trouve, afin de lancer des heuristiques différentes.
int who_played_last=0;	Semblable à la valeur player_turn de minimax, mais évite le déphasage dû à la récursion, en effet l'évaluation du père se passe dans le fils, et l'identité du tour s'en retrouve décalée.
int free_choice_count=0;	Compte le nombre de fois qu'un nœud fournit à ses fils la possibilité de jouer où bon leur semble sur l'ensemble du métamorpion.
int available[SIZE]={};	Tableau de morpion, sert à indiquer si un morpion est libre 0, gagné par le joueur 1, gagné par l'adversaire -1, ou bien si aucune case n'est disponible mais aucun gagnant 2.
int occupied[SIZE][SIZE]={};	Matrice constituant le tableau de jeu, on y stocke des 0 si la case est libre, 1 si occupée par le joueur, et -1 si occupée par l'adversaire.
static constexpr int matrix_point[SIZE][SIZE];	Instancie une matrice statique qui pondère chaque case du métamorpion afin de gagner du temps de calcul dans l'évaluation statique.

FIGURE 6.2 – Les attributs de State

Signatures

Fonctions :

```
int tictactoe_id_row(int row, int col);  
int tictactoe_id_col(int row, int col);  
void version(int (&move)[2], int row, int col);  
void theme(int (&move)[2], int row, int col);
```

Méthodes :

State

```
int winner();  
int winner_meta();  
int tictactoe_mult();  
bool update_available();
```

Node

```
int playable_next_generate();  
void children_generate(int player_turn);  
void minimax(int h, int player_turn, std::chrono::duration<double> & allocated_time);  
int exist_children();  
int possible_horizon(std::chrono::duration<double> & allocated_time);  
int eval();  
void sort_playable();
```

Nom	Retour	Effets de bord	Description	Complexité
tictactoe_id_row	Retourne un numéro de morpion.		Le numéro du morpion en coordonnées minimax depuis la ligne et la colonne en coordonnées codingame.	Constante
tictactoe_id_col	Retourne un numéro de case.		Le numéro de la case en coordonnées minimax depuis la ligne et la colonne en coordonnées codingame.	Constante
version		Modifie le tableau move passé en référence.	Traduit les coordonnées codingame en coordonnées minimax. En appelant tictactoe_id_row/col.	Constante
theme		Modifie le tableau move passé en référence.	Traduit les coordonnées minimax en coordonnées codingame. En appelant version (utilisation des propriétés de bijection entre les ensembles de départ et d'arrivée).	Constante
winner	Renvoie l'identité du gagnant du morpion (1/-1) si alignement trouvé sinon 0.		Via des comparaisons booléennes paresseuses de la grille de jeu occupée, cherche un alignement de trois 1 ou -1 dans un morpion. Le paramètre est un reliquat, il peut être supprimé et remplacé par last_row, mais il permet une souplesse pour tester de nouvelles heuristiques. Destinée à actualiser la valeur d'une case de available par la valeur du gagnant du morpion ciblé (identifié par le paramètre).	Constante
winner_meta	Renvoie l'identité du gagnant du meta_morpion (1/-1) si alignement trouvé sinon 0.		Via des comparaisons booléennes paresseuses de la grille des morpions available, cherche un alignement de trois morpions. Calcul aussi dans le cas d'un ex aequo si le joueur perd parce qu'il possède moins de morpions que son adversaire. Destinée à actualiser le statut de l'attribut win selon le gagnant, appelée dans children_generate.	Constante
tictactoe_mult	Renvoie 0 si une case au moins est libre dans un morpion ciblé.		Le morpion ciblé est celui passé en paramètre. En multipliant la valeur de chaque cases de ce morpion, entre elles, si l'une d'elles est libre (valeur 0) alors le 0 étant absorbant pour la multiplication indiquera qu'au moins une case est disponible sinon cela sera des multiplications de 1/-1 successives.	Constante
update_available	Renvoie la confirmation que available a été modifié.	Actualise la grille des morpions gagnés/libres.	Le booléen retour est utilisé pour limiter le nombre d'appel à la méthode winner_meta, une vérification du méta morpion ne s'effectuera que si un morpion vient d'être gagné par l'un des deux adversaires. L'intérêt principal est celui de l'effet de bord, qui suit toujours une modification de la grille de jeu occupée. Qui pourrait être fusionnée en 1 seule méthode par ailleurs, mais une effet de bord supplémentaire serait encore plus ambiguë. Le paramètre est donc celui du morpion ciblé, on regarde si la case jouée dans le morpion ciblé vient de faire remporter ce morpion au joueur et si oui available est actualisé.	Constante
playable_	Renvoie	Remplie	Si la partie est terminée alors il faut l'empêcher de	Constante

FIGURE 6.3 – Méthodes et fonctions de minimax

next_gene_rate	combien de simulations un nœud possède.	playable_next avec les coordonnées de ces simulations, et actualise l'attribut free si nécessaire.	créer les simulations suivantes, avec le booléen win qui englobe toute la méthode. Ensuite on traite le cas du premier tour avec la main à l'adversaire. Enfin il faut dissocier s'il s'agit d'un coup où le prochain coup est dirigé vers un morpion précis ou s'il sera possible de jouer dans tout le méta-morpion, auquel cas on en profite pour actualiser l'attribut free.	Réallocation possible via push_back : linéaire en taille de vector playable_next
children_generate		Ajoute un shared pointer à child_array.	Crée un shared pointer de type nœud, le configure selon les valeurs fournies par l'instance appelante, puis push_back dans le vector child_array.	Constante
minimax				Linéaire sur taille de child_array + Recursion : linéaire sur la taille de playable_next + Tri linéaire sur taille de playable_next et quadratique si réallocation (push back)
exist_children	Renvoie l'indice si le fils existe sinon -1.		Cherche dans child_array un nœud ayant les mêmes coordonnées que celles recherchées. Ce nœud est identifié par l'indice qu'il occupe dans child_array.	Linéaire sur la taille de vector child_array
possible_horizon	Renvoie une valeur d'horizon h.		Selon les implémentations, cette méthode retourne une valeur de h différentes, en fonction du nombre de simulations à venir, ou du temps restant.	Constante
eval	Renvoie la valeur d'une grille.		À un instant t de la partie, effectue une évaluation dite statique. Sert à diriger les coups, et est essentielle pour élaguer.	Constante
sortPlayable		Trie playable_next.	Échange playable_next avec un vector trié avec les coups au centre et au coins en dernier.	Linéaire sur la taille de vector playable_next Quadratique si réallocation via push_back.

FIGURE 6.4 – Méthodes et fonctions de minimax

Structures de données et statistiques - MCTS

Nom fonction/méthode	Liste paramètres	Type de retour	Brève description
getValue	entier i	Type template	Renvoie l'élément du conteneur à l'indice 'i'
getValueXY	entier i, entier j	Type template	Renvoie l'élément du conteneur à l'indice '(i, j)'
setValue	entier i, type_template value		Remplace l'élément du conteneur à l'indice 'i' par la valeur 'value'
setValueXY	entier i, entier j, type_template value		Remplace l'élément du conteneur à l'indice '(i, j)' par la valeur 'value'
getEmptyPositions		vector<Position>	Renvoie une liste de Positions vide correspondant aux cases vides de la grille visée
checkStatus		int	Renvoie le statut actuel de la grille visée (1 ou 2 = le joueur 1/2 est gagnant ; 0 = égalité ; -1 = partie en cours)
getValidActions		vector<Position>	Renvoie la liste des positions valides (que l'on a le droit de jouer) de la grille cible
getRandomChildNode		shared_ptr<Node>	Choisit aléatoirement un nœud fils parmi la liste des fils du nœud cible
getChildWithMaxScore		shared_ptr<Node>	Renvoie le nœud fils ayant le plus haut score « victoires/visites » parmi les fils du nœud cible
getAllPossibleStatesEnhanced		vector<State>	Renvoie une liste de State (≈ grille) possibles à partir du nœud cible (nécessaire à la phase d'expansion)
MonteCarloTreeSearch	shared_ptr<Node> root, int playerNo		Lance l'algorithme du MCTS sur le nœud 'root' avec 'playerNo' indiquant si on joue en premier ou pas
selectPromissingNode	shared_ptr<Node> root	shared_ptr<Node>	Applique la phase de sélection sur 'root'
findBestNodeWithUCT	shared_ptr<Node> node	shared_ptr<Node>	S'occupe de la partie calcul de l'UCT
expandNode	shared_ptr<Node> node		Applique la phase d'expansion sur 'node'
simulateRandomPlayout	shared_ptr<Node> selectedNode	int	Applique la phase de simulation sur le nœud 'selectedNode'
backPropagation	shared_ptr<Node> node, int playerNo, int randomPlayout		Applique la phase de BackPropagation en remontant l'arbre à partir de 'node'

FIGURE 6.5 – Méthodes et fonctions de MCTS

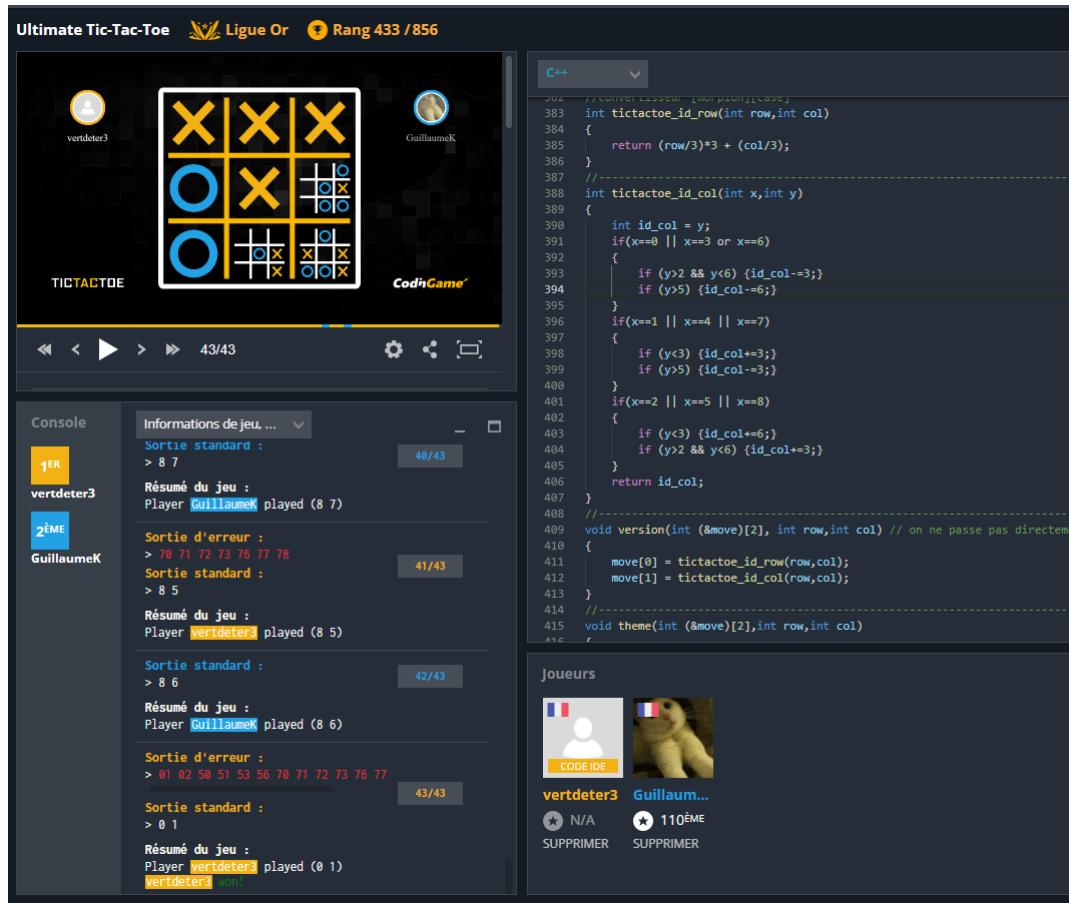


FIGURE 6.6 – Victoire face à l'encadrant

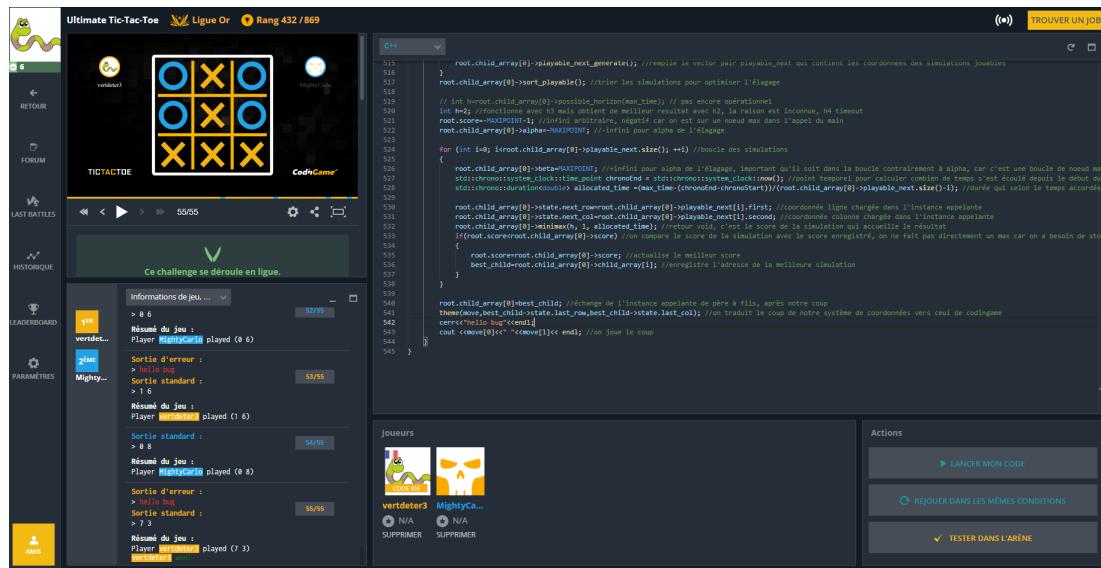


FIGURE 6.7 – Victoire minimax face au boss or

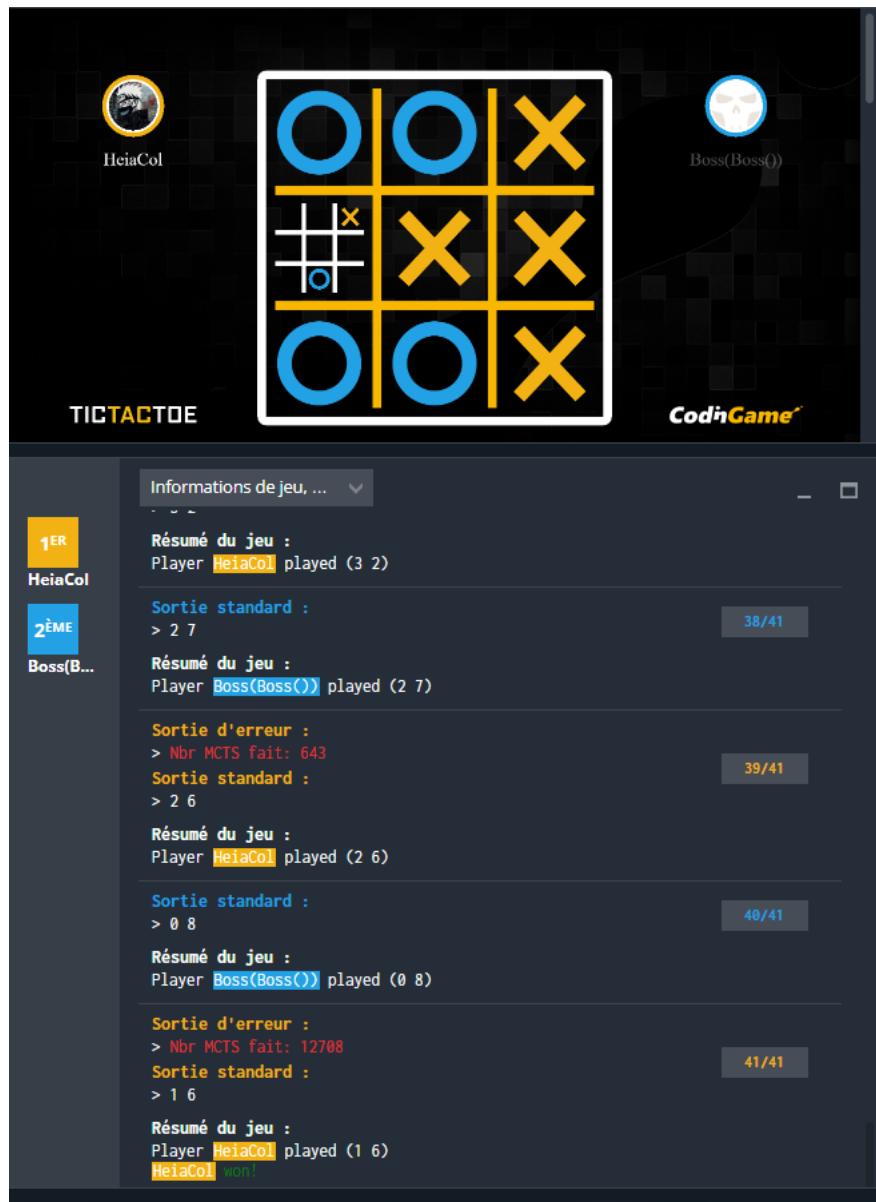


FIGURE 6.8 – Victoire Monte-Carlo face au boss argent

The screenshot shows a task management interface with the following columns:

- Nom**: Rendu livrables et rapport, démonstration pour que-moi et minmax marche sur coding game, Phase-3-UML+Latex+codage-basic+fonction évaluation minmax + implémentation, algo mc et éventuellement codage, ligue-bois-algo minmax, revue-C++, Phase-2-Organisation(git+bitbucket)+UML+Ligue-Bois (minimax) + clôture-MG, Phase-1-Recherche d'informations, Soutenance, Projet, Vacances de Printemps, algo minmax-alpha-beta, ercer-organigramme, Distribuer les tâches.
- Activité**: le 7 mai, 19:00, le 28 avril, 15:30, le 28 avril, 15:24, le 28 avril, 15:23, le 22 février, 23:12, le 10 février, 10:55, le 14 février, 16:15, le 14 février, 15:41, le 14 février, 15:41, le 14 février, 15:41, le 14 février, 13:39, le 11 février, 13:03, le 11 février, 13:03.
- Date limite**: Demain, 19:00, le 22 mars, 04:00, le 11 mars, le 22 février, 21:00, le 22 février, 23:00, le 18 février, 19:00, non spécifié, le 25 mai, 19:00, le 25 mai, 19:00, Pas de date limite, le 14 février, 20:00, non spécifié, non spécifié.
- Crée par**: Lang Gu, tomisserantm@gmail.com, Lang Gu, tomisserantm@gmail.com, Lang Gu, tomisserantm@gmail.com, tomisserantm@gmail.com, tomisserantm@gmail.com, Lang Gu, Lang Gu.
- Responsable**: Lang Gu, tomisserantm@gmail.com, Lang Gu, tomisserantm@gmail.com, Lang Gu, tomisserantm@gmail.com, tomisserantm@gmail.com, tomisserantm@gmail.com, Lang Gu, Lang Gu.
- Projet**: Projet Programmation 2 ..., Projet Programmation 2 ...
- Tags**: (tags visible in the UI)

FIGURE 6.9 – Bitrix24

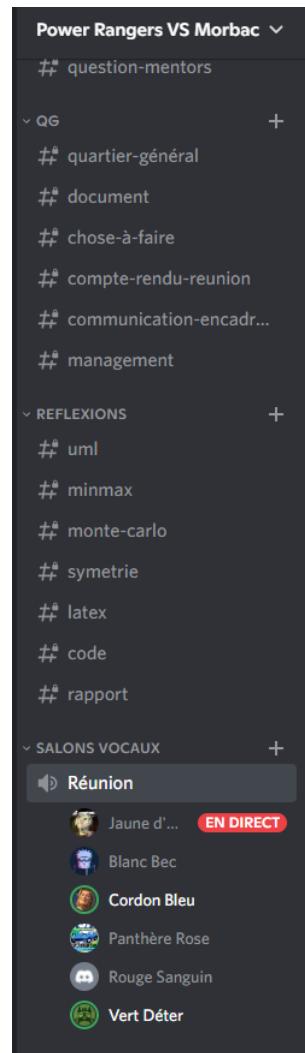


FIGURE 6.10 – Organisation des salons

Vous pouvez me le renvoyer en PM si vous préférez (mais je partagerai certainement des infos avec Tom pour le management, sinon faites une mention "ne pas divulguer" quelle que soit vos raisons)
Répondez au maximum svp

Prenom:

groupe L3:

Couleur :

Motivation : nombre d'heures que je pense consacrer au projet en moyenne / semaine entre la recherche, le code... (jusqu'à l'impression d'azerty sur mon front ou que mort s'en suive, 2h, 4h....):

(ENLEVER le choix qui ne va pas)

Je préfère travailler seul ? oui non

Je veux travailler en binôme ? oui non

Je pourrais travailler en binôme/trinôme ? oui non

Je vous mets des idées pour pouvoir remplir les prochaines questions:

Langage, outil(jira, mockito...ou type d'outil (organisation, comptabilité, IDE...), web, test (unitaires...), parler lors des réunions/oraux, travailler en visio, s'enregistrer, animer une réunion, rédiger le rapport, planifier des tâches, faire des rétrospectives(parler des gens, du projets, ce qu'on trouve bien, ce qui peut s'améliorer, c'est une réunion à part)...langues vivante, soft skill(patience, communication, observation, négociation...), C++, prog dynamique, gestion mémoire, marketing, management (planning poker, coaching...), story, pâtisserie...

N'hésitez pas à mettre des trucs pas trop en rapport, ca aidera à former des sous groupes par affinité par exemple

Compétent en:

Notion en :

Je voudrais m'améliorer en :

Je voudrais apprendre :

Ne souhaite pas faire :

Absolument pas faire de :

Ce que vous attendez de ce projet :

Ce que vous attendez des autres membres du projet en général et individuellement :

Ce que vous attendez du professeur(s) encadrant :

Troubles, handicaps, difficultés particulières:

Autre choses que vous voudriez partager ou avertir:

FIGURE 6.11 – Questionnaire remis aux membres

Index

A
Algorithmes Minimax 24
Algorithmes Monte-Carlo 32
Attributs 43

C

C++ 12
Cahier des charges 8
Codingame 14
Contexte 6

D

Diagramme de Gantt 11
Différentes approches 8
Débogage 36

G

Généricité 22

M

Minimax 17
Monte-Carlo Tree Search 19
Méthodes et fonctions 44

N

Nos motivation 7

O

Outils 12

S

Smart pointers 23
Structures de données et statistiques - MCTS
48
Structures de données et statistiques - mi-
nimax 43

U

UML 19