LU2IN013

Rapport de projet

Quoridor

DM-IM L2
Besancenot Henri
Hua Zhichun
Sandlarz Nino

Enseignants
P.Shams
N. Benabbou

Année universitaire 2021 — 2022

Abstract

Quoridor est un jeu qui se joue à deux joueurs, l'un contre l'autre. La partie se déroule sur un plateau de 9x9 cases, les joueurs jouent à tour de rôle et peuvent choisir entre déplacer leur pion ou poser un mur pour tenter de gêner l'adversaire. Chaque joueur possède 10 murs qui font chacun deux cases de large, et il est interdit de bloquer complètement l'adversaire. De plus, il existe plusieurs cas particuliers qui régissent les mouvements entre joueurs : par exemple un joueur peut "sauter" par dessus son adversaire s'il est sur une case adjacente. Enfin, la partie se termine lorsque l'un des joueurs atteint le côté opposé.
Nous présenterons donc en premier lieu une modélisation de ce jeu avant de nous consacrer à la création de plusieurs agents capables de jouer, voire de gagner, contre des joueurs humains.

Contents

1 Introduction

Depuis les légendaires matchs de Garry Kasparov en 1996 et 1997 contre le super calculateur d'IBM Deep Blue, les ordinateurs et l'informatique sont au coeur des avancées techniques et stratégiques dans le monde des jeux.
Quoridor est un jeu de société stratégique où deux joueurs s'affrontent. Notre objectif est de proposer une modèlisation avec une interface graphique à la fois agréable et pratique. Nous présenterons une version multijoueurs opposant des adversaires en réseau, et une version opposant des agents contre des humains.
Pour ce faire nous avons choisi d'utiliser le moteur de jeu Unity qui utilise le langage C#. En effet malgré la barrière d'un nouveau langage à apprendre, Unity semblait tout indiqué car il répondait à la plupart de nos attentes :

- une interface graphique pratique et efficiente
- une architecture serveur facilitée
- un langage de programmation objet idéal pour l'écriture des agents

2 Modélisation

Proposer une modélisation robuste et instinctive est essentiel au bon déroulement du projet puisque toutes les futures implémentations (pathfinding, déplacement, mini max, ...) vont directement en dépendre.

2.1 Représentation du plateau

Quoridor se joue sur un plateau de 9x9 cases. Nous avons d'abord pensé utiliser une liste de listes de cases mais après reflexion nous avons opté pour un dictionnaire, `tilesDico`, où la clé de chaque case est un vecteur contenant ses coordonnées. Ce choix a notamment permis d'alléger la syntaxe en obtenant la case désirée à partir d'un seul paramètre.
Distinguer les coins des cases n'était pas non plus absolument nécessaire, mais ne pas le faire aurait rendu certaines méthodes plus lourdes, et leur réalisation moins évidente. Par exemple, à la pose d'un mur, comme celui-ci fait deux cases de large il aurait fallu l'associer à au moins deux cases différentes en plus de garder d'une manière ou d'une autre l'information que le passage au milieu du mur était obstrué. Pour éviter cette difficulté inutile nous avons décidé de créer des `corners` rangés dans un dictionnaire, `cornersDico`, accessibles directement à partir de leurs coordonnées. Ainsi la pose d'un mur se fait facilement : il suffit de choisir le coin correspondant et d'indiquer l'orientation (Horizontal ou Vertical) du mur que l'on veut poser.

2.2 Joueur et Agent

Maintenant que nous avons un plateau il faut à présent décider comment représenter les joueurs et les agents. Comme les comportements de ces deux entités sont similaires, nous avons créé la classe abstraite `BaseUnit` qui possède deux attributs public `wallCount` et `occupiedTile` qui représentent respectivement le nombre du mur et la case occupée par l'entité. La classe possède également la méthode `SetUnit()` qui permet de placer l'entité sur une case et `SpawnWall()` qui permet de poser un mur. Ainsi les Joueurs et Agents vont hériter des attributs et des méthodes de cette classe.
Pour les IAs nous avons ensuite créé une deuxième classe abstraite `BaseIA` qui définit la méthode abstraite `PlayIA()`. Cette méthode est redéfinie par l'agent, et lancée au début de son tour pour déterminer quel coup il va jouer.
Le Player, lui, sera soumis aux actions du joueur (souris au dessus d'une case ou d'un coin, click sur un bouton, ...). Il fera également attention à lancer les fonctions SetUnit et SpawnWall via le serveur pour que le joueur adverse reçoive les changements sur sa version du jeu.

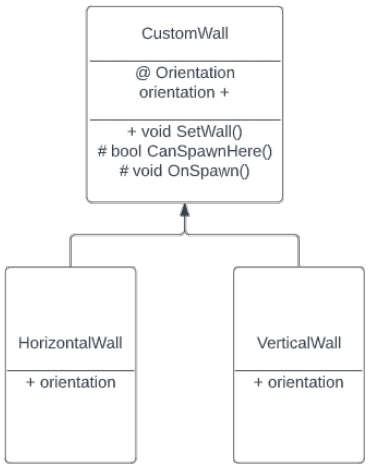

Figure 1: Schéma relationnel entre BaseUnit, BaseIA et Player

2.3 Mur

L'objet `Wall` est le dernier objet nécessaire à la modélisation du jeu. Chaque mur sera caractérisé par sa position et son orientation (horizontale ou verticale). Chaque coin de `cornersDico` représente un emplacement qui est occupable par un unique mur.
Pour faciliter la manipulation de cet objet nous avons créé les classes `VerticalWall` et `HorizontalWall` qui héritent de la classe `CustomWall` et possèdent comme attribut leur orientation. Ces classes redéfinissent également les méthodes `CanSpawnHere()` et `OnSpawn()` qui permettent respectivement de savoir si un tel mur peut être instancié à cet endroit et de bloquer le passage vers les cases adjacentes lors de leur pose.

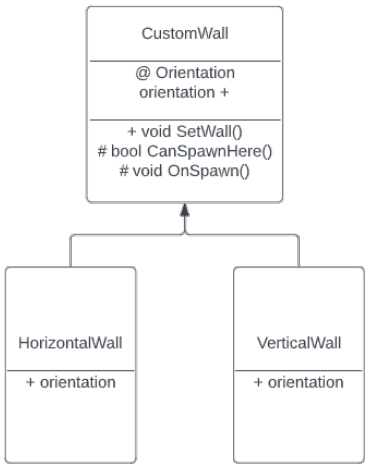

Figure 2: Schéma relationnel entre les différents types de mur.

2.4 Managers

La gestion du jeu a été assignée aux Managers, des singletons prenant en charge chacun un aspect précis du projet. En voici une liste complète avec leurs rôles :

Manager	Rôle
GameManager	Gère les phases de jeu
GridManager	Crée le plateau
SceneSetUpManagerManager	Prépare la scene pour l'arrivée des joueurs
ModeManager	Change le mode du joueur (mur/déplacement)
ReferenceManager	Référence les différents objets
UIManager	Régit l'affichage graphique
InputManager	Reçoit les inputs de la souris
ServeurManager	Gère la liaison avec le serveur
RegisterManager	Enregistre la partie

L'utilisation de singletons assure l'unicité d'une instance de ces classes, et permet surtout un accès immédiat aux différentes méthodes qu'elles proposent.

3 Pathfinding

L'algorithme de pathfinding, ou de plus court chemin, est une partie importante de la programmation des agents puisque chacun d'entre eux pourra l'utiliser pour faire des choix rationnels et efficaces sur le chemin à suivre pour arriver à la victoire.

3.1 Parcours en largeur

Une approche naïve de la recherche du plus court chemin est le parcours en largeur. Le principe est simple, partant d'une case, on explore les cases à proximité et on leur associe une distance de 1. On explore ensuite les voisines de ces dernières et on leur associe une distance de 2. On continue ainsi jusqu'à trouver la case que l'on veut atteindre, de distance n. Pour récupérer le plus court chemin il suffit ensuite de parcourir les cases avec une distance décroissante. Ainsi on obtient un chemin avec la distance la plus petite possible pour la case recherchée.

3.2 Algorithme A*

Nous avions d'abord pensé utiliser le parcours en largeur, mais l'algorithme A* a l'avantage d'être plus flexible et présente une meilleure complexité moyenne dans le sens où une bonne heuristique permet de visiter moins de sommets et donc d'accélérer l'algorithme. Il repose sur le principe suivant :
On déclare la liste `open` des cases à parcourir, la liste `close` des cases déjà parcourues, et la variable `current` désignant la case traitée à un instant t. On initialise la variable current avec la case de départ. Pour chaque case atteignable depuis la case current, si celle ci n'est pas déjà dans `close` on calcule :

1. Le G-cost qui est la distance de proche en proche depuis la case de départ
2. Le H-cost (aussi appelé heuristique) qui désigne la distance de Manhattan entre la case current et la case objectif
3. Le F-cost qui est la somme du G-cost et du H-cost

Ensuite, il suffit de répéter l'algorithme en associant la case d'`open` avec le plus petit F-cost à `current`. Une fois arrivée à la case objectif, on peut remonter les cases de proche en proche pour tracer le meilleur chemin.
Cette méthode est en moyenne plus efficace que le parcours en largeur car on peut espérer que moins de sommets soient visités grâce à l'heuristique qui guide la recherche. A noter que l'agorithme A* retourne une solution optimale à condition que l'heuristique soit minorante, c'est-à-dire qu'elle ne surévalue jamais le vrai coût jusqu'au but. Or, dans le cas du Quoridor ce n'est pas toujours le cas en utilisant la distance de Manhattan : Si l'adversaire se trouve sur le chemin à parcourir, alors le joueur peut le "sauter" gagnant ainsi une case. Nous avons donc dû adapter le calcul du H-cost pour que celui-ci vérifie si l'adversaire se trouve entre le current et son objectif, et s'ajuste en conséquence.

4 Implémentation d'IA joueuses

4.1 Base

Comme précisé plus tôt dans le rapport, les agents que nous allons développer héritent tous de la classe `BaseIA` qui possède une méthode publique `PlayIA()` permettant à l'IA de jouer. Comme cette classe hérite elle même de `BaseUnit` elle connaît aussi sa position et le nombre de murs qu'il lui reste.

4.2 Move and Wall

4.2.1 IAMove

Le premier agent implémenté, `IAMove`, a un comportement simple qui consiste à suivre le plus court chemin retourné par la fonction de pathfinding sans se soucier du comportement adverse. Son attitude élémentaire le rend prévisible, ce qui a aidé à la réalisation du mode de jeu `Joueur vs IA`.

4.2.2 IAWall

Peu après `IAMove`, l'agent `IAWall` fut créé. Cet agent a également un caractère prévisible puisque son unique but est de poser un mur sur le chemin de son adversaire. Nous avons posé l'hypothèse que poser un mur proche de son adversaire avait plus de sens que poser un mur proche de soi. Lorsque cette IA n'a plus de mur, elle se comporte comme `IAMove`.

4.2.3 IAMoveWall

Finalement, l'agent `IAMoveWall` est une combinaison des deux agents précédents qui choisit de manière équiprobable entre suivre le plus court chemin et poser un mur sur le chemin de son adversaire. Ce dernier agent n'est pas très fort, comme nous le verrons dans la partie 5, mais il produit un comportement plus naturel que les deux précédents.

4.3 Mini Max

Un des algorithmes que l'on retrouve le plus dans les jeux de stratégies est celui du Minmax. Son concept est simple : il faut parcourir toutes les positions possibles à partir d'une position initiale et déterminer celle qui maximise son gain, en supposant que le joueur adverse cherche à faire de même avec son propre gain.

4.3.1 Arbre des possibles

Pour parcourir les positions, la structure de données d'arbre semble toute indiquée. Ainsi, la racine représente la position initiale et tous ses fils sont des positions[1] que l'on peut atteindre en jouant une fois puis encore les . Pour cela on peut écrire la classe `Node` qui possède comme attribut un coup `coup`, une profondeur `depth`, un score `score`. Elle possédait initialement une liste de fils mais notre implémentation de minimax nous a permis de ne pas la prendre en compte.

4.3.2 Concept

Selon une analyse du jeu qui date de 2006, il y aurait approximativement $3.9905 * 10^{42}$ positions possibles. Calculer toutes les possibilités est donc impossible, nous allons donc nous restreindre à une profondeur `maxdepth`. Supposons $maxdepth = 2$ et considérons un arbre où chaque noeud possède 3 fils :

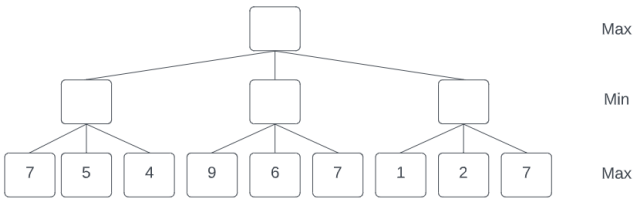

Figure 3: Arbre vide

Chaque noeud de la profondeur la plus basse possède un score, à partir desquels on déduira les scores des noeuds supérieurs. Ici pour le premier fils à gauche qu'on va appeler `current`, on voit que ces fils ont des scores de 7, 5 et 4, comme `current` est de type `Min` il prend le plus petit des trois scores, donc 4. En utilisant la même logique sur la racine et ses autres fils on obtient l'arbre final suivant :

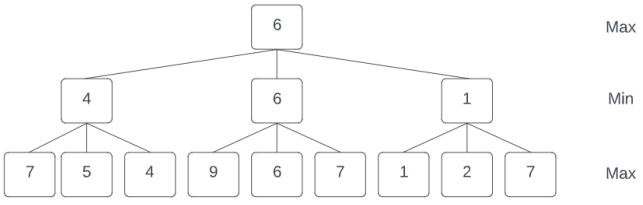

Figure 4: Arbre rempli

[1] On parle ici de la façon dont les joueurs et les murs sont placés sur le plateau

En particulier la racine est de type `Max` donc elle prend le plus grand score de ceux de ses fils. Ainsi, on sait que le coup à jouer est celui du deuxième fils.

4.3.3 Implémentation

Tout d'abord il faut définir jusqu'à quelle profondeur on veut calculer les positions. En faisant un calcul approximatif on voit rapidement que le nombre de position est exponentiel avec la profondeur. Prenons un exemple avec $maxdepth = 3$. On a une position de départ, on suppose que tous les coins sont libres et que les joueurs peuvent se déplacer librement, il y a donc 64 positions de mur possibles et 4 degrés de liberté pour le joueur. Au total il y a donc $64 * 2 + 4 = 132$ noeuds pour la première profondeur. Si on pose un mur chaque fils aura 130 fils et si on bouge, chaque fils aura 132 fils. Ainsi la racine aura entre $132 * 130 = 17160$ et$132 * 132 = 17424$ petit-fils. Si on continue vers la profondeur suivant, on peut donc estimer à plus de 2 millions le nombre de noeuds à générer et à évaluer, ce qui n'est pas raisonnable. Ainsi, nous avons décidé de travailler avec une profondeur de 2. Pour parcourir l'arbre nous avons envisagé deux options :

option 1 : Générer l'arbre en itérant sur les profondeurs puis remonter depuis les feuilles vers la racine en appliquant les régles de score.

option 2 : Générer l'arbre récursivement et évaluer le score des noeuds au fur et à mesure que les profondeurs sont générées.

La première solution nécessite de garder en mémoire tout l'arbre avant d'attribuer des scores, ce qui ne nous a pas semblé optimal étant donné le nombre de positions que l'on doit générer. On a donc opté pour la seconde option, qui a l'avantage d'être moins gourmande en mémoire et plus simple à implémenter. Voici un pseudo code de notre algorithme du max :

Algorithm 1 Max

```
1: if On est a la profondeur maximale then
2:     Calculer le score du noeud et retourner son coup
3: end if
4: for Chaque case atteignable par le joueur do
5:     Creer un nouveau noeud
6:     Appliquer Min sur ce noeud
7:     if Le score du noeud est plus grand que celui actuel then
8:         Actualiser le score
9:     end if
10: end for
11: for Chaque coin dans cornerDico do
12:     Creer deux nodes avec un mur vertical et un mur horizontal
13:     Appliquer Min sur ces noeuds
14:     if Le score d'un des noeud est plus grand que celui actuel then
15:         Actualiser le score
16:     end if
17: end for
18: Return le coup qui a généré le meilleur score
```

La fonction de Min est complètement symétrique, il suffit de remplacer les "Min" par "Max" et les "plus grand" par "plus petit". Il suffira ensuite d'appeler la méthode `Max()` sur la position initiale pour avoir le coup que l'on doit jouer.

4.4 Alpha Beta Pruning

L'algorithme d'AlphaBeta est très similaire à celui de MinMax, mais il présente l'avantage de ne pas parcourir certaines branches de l'arbre sans changer le résultat final. Reprenons l'arbre dessiné plus tôt, profitons en pour nommer les noeuds :

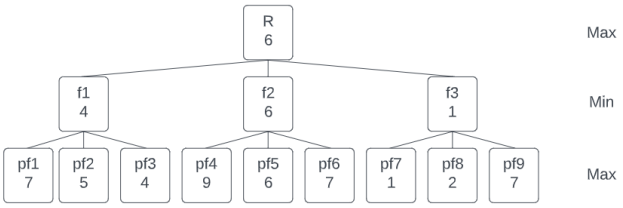

Figure 5: Alpha Beta

Prenons un exemple, supposons que nous avons déjà appliqué l'algorithme sur f1 et f2. R a une valeur courante de 6, qui est la borne inférieure. On appelle à présent la fonction Alpha Beta sur f3 qui l'appelle sur pf7, modifiant sa valeur

courante à 1. A cet instant, la borne supérieure devient 1 : on a donc que la borne inférieure est plus grande que la borne supérieure. Ainsi, quoi qu'il arrive dans la suite de l'appel, R aura un score de 6 : il est donc inutile de parcourir pf8 et pf9. Dans cet exemple l'arbre est petit et on ne peut donc pas enlever beaucoup de branches; dans le cas de Quoridor l'arbre est beaucoup plus grand et l'élagage montre tout son intérêt calculatoire.
Notre implémentation ne marchait malheureusement pas, nous avons donc décidé d'utiliser une version plus concise de `MiniMax`, `NegaMax` et d'élaguer dessus pour obtenir `IANegaAlphaBeta`. Cet algorithme a l'avantage de ne pas s'écrire en deux composantes et a la même complexité calculatoire que `AlphaBeta`. Il est également un peu plus simple à coder et est par conséquent plus facile à déboguer.

4.5 Fonction de score

4.5.1 Première approche

Pour l'algorithme du Minimax et ses variantes, une fonction d'évaluation de position est nécessaire. Déterminer comment savoir si une position est avantageuse ou non est probablement la tâche la plus ardue du programme. Il faut donc trouver des caractéristiques indiquant dans quelle mesure une position est favorable ou défavorable. En effet, comme on ne va pas jusqu'à un état final de l'arbre (où l'on pourrait savoir quel est le gain de chaque joueur en regardant juste quel joueur est vainqueur) il faut être capable de déterminer un score pour un état rendant comtpe le mieux possible compte de quel joueur est le plus avantagé.

Posons ces caractéristiques comme des fonctions, on aura :

- f_1 Distance entre le joueur et son objectif
- f_2 Distance entre l'IA et son objectif
- f_3 Le nombre de murs du joueur
- f_4 Le nombre de murs de l'IA

Il s'agit de features qui paraissent naturellement traduire si un joueur est dans une position favorable ou pas. Nous considérons que le score d'une position est une combinaison linéaire de ces fonctions. Ansi la formule du score est :

$$score = \sum_{i=1}^{n=4} \omega_i * f_i \tag{1}$$

où ω_i est le poids associé à la caractéristique f_i, $i \in \{1, 2, 3, 4\}$.

4.5.2 Optimisation des poids via un algorithme génétique

Pour avoir un agent fort, il faut donc une bonne fonction d'évaluation. Cependant, estimer les poids ω_i des caractèristiques données plus tôt n'est pas évident. Une solution pour trouver des paramètres optimaux est de passer par un algorithme génétique. Un algorithme génétique s'inspire de la séléction naturelle pour faire tendre les paramètres vers un optimum. On a considéré un algorithme génétique dit de séléction par tournois, dont le principe est le suivant :

1. On génère aléatoirement une population d'IA
2. Les IA font chacune un match contre une autre IA
3. Les IA qui perdent sont éliminées
4. On fait se reproduire les IAs restantes
5. On recommence à partir du point 2 (le nombre de fois souhaité)

Ainsi seuls les IA gagnantes survivent, et vont générer une nouvelle génération d'IA. La reproduction de deux IA se fait par croisement, puis éventuellement par mutation. Par exemple les parents suivants :

Poids	Père	Mère
ω_1	0.05	0.75
ω_2	0.29	0.39
ω_3	0.75	0.07
ω_4	0.20	0.94

peuvent produire les enfants suivants :

Poids	fils 1	fils 2	fils 3	fils 4
ω_1	0.75	0.75	0.05	0.05
ω_2	0.39	0.29	0.29	0.39
ω_3	0.07	0.747	0.07	0.07
ω_4	0.20	0.20	0.94	0.94

Les poids des enfants sont donc des combinaisons des poids des parents. On observe cependant que le poids ω_3 du `fils 2` est légèrement différent de celui de son père : ce phénomène est dû à une mutation aléatoire de ce gène. On fera la remarque que notre implémentation de l'algorithme part du postulat que L'IA veut réduire la distance de sa position à son objectif et que plus l'adversaire a de murs, moins la position est bonne.

5 Résultats

Afin de déterminer quelle IA était la plus performante, nous avons mis en place des parties d'IA contre IA. Comme nos IA sont déterministes, un match aller et retour suffit pour savoir qu'elle implémentation est la meilleure. La seule IA qu'on fera jouer un plus grand nombre de fois est IAMoveWall, qui possède un facteur aléatoire. Notons que nous n'avons pas mis d'`IANegaAlphaBeta` (abrégée `IANAB`) ayant suivi un processus de séléction car nous n'avons pas eu le temps de faire tourner l'algorithme génétique assez longtemps. Nous supposions que les IA `Wall` et `Move` seraient les moins performantes et les IA `MoveWall` et `NAB` les plus performantes. Le tableau suivant vient le confirmer :

p2 - p1	IAWall	IAMove	IAMoveWall	IAMiniMax	IANAB(1d)	IANAB(2d)
IAWall	1-1	2-0	2-8	0-2	0-2	0-2
IAMove	0-2	1-1	1-9	0-2	0-2	0-2
IAMoveWall	8-2	9-1	1-1	0-10	0-10	1-9
IAMiniMax	2-0	2-0	2-0	1-1	1-1	1-1
IANAB(1d)	2-0	2-0	10-0	1-1	1-1	0-2
IANAB(2d)	2-0	2-0	9-1	1-1	2-0	1-1

6 Amélioration possible

L'algorithme génétique ayant été fini tardivement, nous n'avons pas pu le faire tourner suffisamment longtemps pour avoir des résultats convaincants. Quelques jours, voire semaines, de plus auraient permis d'obtenir de meilleurs poids pour la fonction score de `IANegaAlphaBeta`.
D'autre part, la fonction de score elle même n'est sans doute pas assez précise. En effet dans certaines configurations l'agent à tendance à répéter une succession de coups à l'infini, l'empêchant d'atteindre son but. Par exemple dans la configuration où il se trouve face à un mur, il préfère rester derrière en anticipant un mouvement qui lui serait désavantageux. Une solution suggérée par P.J.C. Mertens serait de calculer la distance entre l'IA et la prochaine rangée, ainsi elle choisirait naturellement le chemin le plus court pour arriver à sa destination. Une autre façon de résoudre ce problème aurait été d'avoir une fonction de score non linéaire (ie : avoir certains coefficients avec des puissances).
Aussi, l'algorithme de `Minimax` et sa variante `Negamax` ne sont pas très efficaces. En effet, l'algorithme génère toutes les positions atteignables à partir d'une position initiale. Cependant, beaucoup de positions sont évidemment inutiles à prendre en compte. Par exemple, on pourrait simplement considérer des murs dans une région autour du joueur adverse et sur son plus court chemin. Cette solution présente l'avantage d'évaluer une moins grande quantité de branches,

qui sont la source majeure de calcul, mais elle présente aussi le désavantage de devoir calculer pour chaque position le plus court chemin et déterminer les cases suffisament proches pour avoir un intérêt.

7 Conclusion

Le projet final propose une grande variété de contenus : le joueur pourra choisir entre lancer une partie en ligne contre un ami, jouer contre des IA de différents niveaux comme `Move`, `Wall`, `MoveWall`, `MiniMax` et `NegaAlphaBeta`, ou encore lancer des matchs entre IA pour déterminer quelle stratégie semble la meilleure. En plus de cela, il aura a sa disposition un algorithme génétique permettant d'entrainer les IA jusqu'à obtenir de meilleurs paramètres.

Pour finir, travailler sur ce projet a été aussi amusant qu'instructif : Il nous a permis de découvrir le domaine de l'intelligence artificielle appliqué aux jeux de stratégies, et nous a donné l'occasion de mettre en pratique les connaissances de programmation objet acquises au cours de l'année.

References

[1] Lisa Glendenning. Mastering Quoridor. Thesis, university of New Mexico, May 2005

[2] P.J.C. Mertens. A Quoridor-Playing Agent. Paper, Maastricht University, June 2006

[3] Wikipedia (2022). Elagage alpha-bêta. https://fr.wikipedia.org/wiki/Élagage_alpha-bêta