Rapport du projet de substitution au stage Intelligence artificielle pour des jeux

Corentin PACILLY Hadrien JOSSE

7 juin 2021

Code disponible sur le dépot Git : https://github.com/JrSmooth6/Fil_rouge

Table des matières

1 Introduction

Dans le cadre de notre troisième année de licence Informatique, nous devions effectuer un stage de 8 semaines en entreprise dans le domaine de l'informatique. Cependant la situation sanitaire liée a la pandémie de la COVID-19 nous à fortement impactés dans la recherche de ce stage. Nous étions au préalable au sein de l'UE projet avec madame Veronique Terrier qui nous a fait part de son intention de ne pas continuer le projet. Nous avons donc décidés en dépit de consignes claires de commencer un projet qui consistait en la création d'un programme java avec interface graphique en MVC regroupant un maximum de jeux possible. Monsieur Niveau nous as contacté le 29 avril pour nous informer qu'il allait encadrer ce projet et que nous devions plutôt nous orienter sur des intelligences artificielles pour différents jeux. Nous avons ensuite eu un entretien avec Mr Niveau le 20 mai, nous donnant des consignes précises quant au travail que nous devions accomplir.

2 Présentation et organisation du projet

2.1 Contraintes et objectif

La contrainte majeure durant ce projet était le temps dont nous disposions, en effet nous étions en train de faire un projet similaire mais sans forcément faire d'intelligences artificielles. Nous avons, par conséquent dus effectuer plusieurs changement quant a l'architecture de notre programme. La compréhension des différents algorithmes était la clé pour la réussite de ce projet, certains nous ayant posés quelques problèmes dans la compréhension ou la réalisation. L'objectif était de faire fonctionner les algorithmes MinMax, NegaMax et ExpectiMinMax au sein de nos différents jeux.

2.2 Outils utilisés pour la réalisation du projet

2.2.1 Eclipse

Pour développer notre programme nous avons fait le choix d'utiliser Eclipse qui est un IDE (Integrated Development Environment) pour les programmes en Java. C'est un IDE sur lequel nous avons l'habitude de travailler, qui propose de nombreuses options pour améliorer les conditions de programmations. Un des avantages qui nous à fait opté pour cet IDE est le fait que nous puissions directement lancer notre programme sur l'IDE nous evitant un passage sur le terminal pour une compilation manuelle. Cet IDE est mis a disposition gratuitement par la Fondation Eclipse.

2.2.2 GitHub

Nous avons choisis d'utiliser GitHub pour le partage de fichier. En revanche nous avons décidés que seul Hadrien effectuerait les changements sur le repertoire Git, en effet suite à une mésaventure l'an passé sur la Forge qui nous à demander beaucoup de travail supplémentaire et les contraintes de temps étant courtes nous ne voulions prendre aucun risque.

2.2.3 Discord

Afin de se partager des fichiers de faibles poids ou échanger de discuter ou de fixer des réunions d'avancement hebdomadaires, nous avons mis en place un serveur Discord nous permettant de communiquer que ce soit depuis un navigateur, un téléphone ou depuis l'application, ce logiciel permettant également de partager son écran dans le but de coder le programme sur un seul ordinateur .

2.2.4 Overleaf

Pour la réalisation de ce rapport en LaTeX, nous avons utilisé le site Overleaf qui permet de réaliser des rapports LaTeX où les fichiers sont partagés et où plusieurs membres peuvent travailler en simultanés à l'image de Google Drive.

3 Les jeux

3.1 Jeu à somme nulle

Un jeu à somme nulle est un jeu où il n'y peut y avoir qu'un seul gagnant, si un joueur gagne la partie son adversaire perd automatiquement, la réciproque est aussi vraie. En d'autres termes, la somme des gains et des pertes de chaque joueur est égale à 0. Un jeu à somme nulle peut également comporter le cas d'une égalité. Dans notre cas, tout les jeux implémentés sont des jeux à sommes nulles. Ce qui va différencier ces jeux est le fait de savoir si ils sont a information complète ou non.

3.2 Jeu à information complète

Un jeu à information complète est un jeu dans lequel chaque joueur connaît au moment de décider du coup à jouer :

— Ses différents coups jouables

— Les différents coups des autres joueurs

— Les gains entraînés par la coup joué

— Le but des adversaires

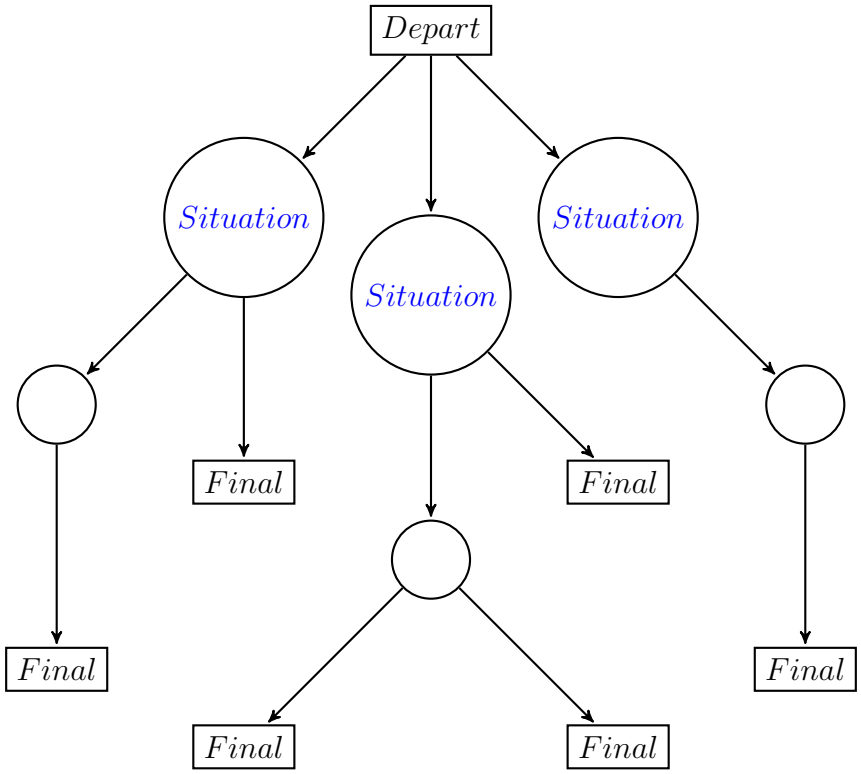
Dans notre cas, les jeux de nim, du morpion et du puissance 4 sont des jeux à informations complètes car ils réunissent tout ces critères. Dans le cas du morpion par exemple, au moment de choisir un coups le joueur connaît les gains de ce coup, il va connaître également les différents coups des adversaires après avoir joué, ainsi que les gains de coup tout en sachant que l'adversaire chercher soit a lui bloquer sa stratégie soit imposer la sienne.

3.3 Jeu à information incomplète

Un jeu à information incomplète est un jeu qui ne répondra pas à un ou plusieurs critères évoqués dans les jeux a information complète. Dans notre cas les jeux à informations non complètes sont le jeu du Black Jack et de la Bataille navale, en effet le joueur ne peut pas connaître le gain d'un coup car il y à ici une partie de hasard quant au gain : le joueur ne peut pas connaître la carte qu'il va tirer ni sa valeur pour le Black Jack, et le joueur ne peut pas savoir si il va faire mouche lors d'un coup à la bataille navale.

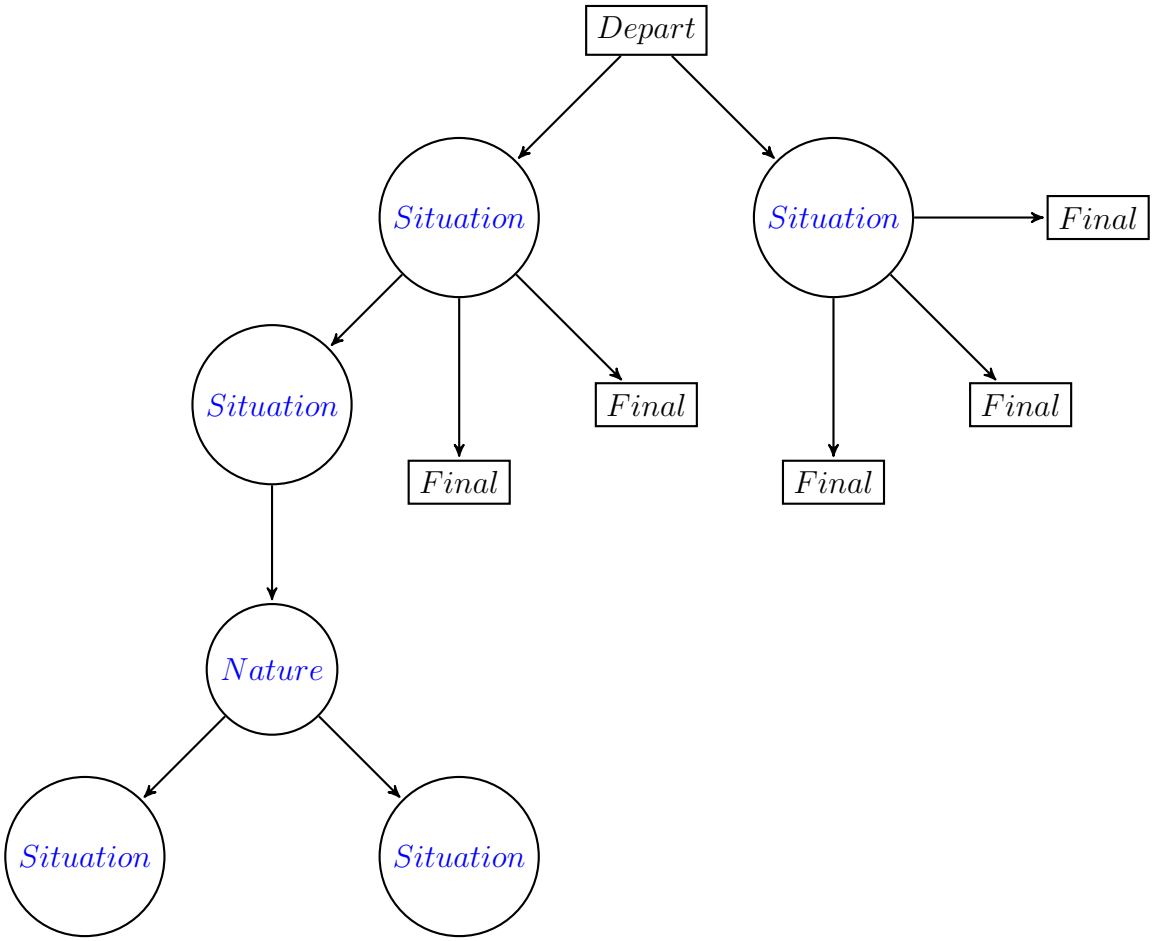
3.4 Représentation d'un jeu à information complète

Dans le cas de la création d'intelligences artificielles pour des jeux il est important de comprendre comment est représenté un jeu pour les algorithmes. Les jeux sont représenté comme des arbres dont les noeuds représentent chaque situation possible du jeu. Une feuille représente une situation terminale, ce qui signifie que le jeu est terminé. Les branches de chaque noeuds vont elles représenter chaque coups jouables pour un joueur en fonction du noeud.



3.5 Représentation d'un jeu a information incomplète

La représentation d'un jeu à information incomplète est sensiblement similaire a celle d'un jeu a information complète. L'endroit où la représentation va différer se situe sur le fait que lorsqu'un joueur choisis un coup a jouer c'est au tour du hasard d'opérer. Cela sera représenter sur l'arbre par le fait qu'un joueur va choisir une branche qui correspond a un coup, a l'extrémité de cette branche il y aura un noeud qui ne sera pas une situation de jeu mais un noeud qui correspond au hasard, les différentes branches de ce noeud hasard seront les différents gains possibles du coup joué par le joueur.



4 Réalisation du programme

4.1 L'architecture du programme

Le programme est constitué de 4 packages :

— le package games

— le package players

— le package orchestration

— le package cartesAJouer

4.1.1 Le package games

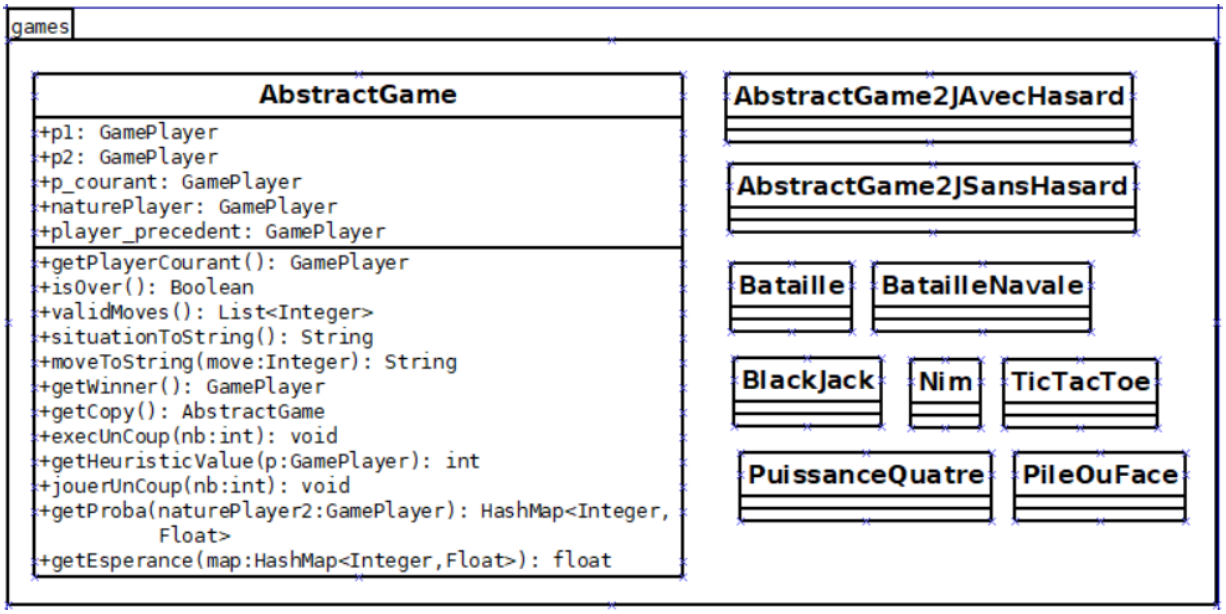
Le package game contient tout les différents jeux du programme. Il possède trois classes abstraites :

— AbstractGame

— AbstractGame2JSansHasard

— AbstractGame2JAvecHasard

Tout les différents jeux vont dépendre de la classe AbstractGame. La différenciation entre les jeux à information complète et ceux a information incomplète réside dans le fait que les jeux a information complète vont dépendre de la classe abstraite AbstractGame2JAvecHasard qui elle même dépend d'AbstractGame. Cette différentiation est due au fait que les jeux à information complète lorsqu'un joueur choisit un coup, ce dernier est appliqué et le joueur courant change, chose que ne fonctionne pas pour les jeux à information incomplète si on souhaite implémenter une intelligence artificielle dessus.



4.1.2 Le package players

Ce package va lui contenir les différents types de joueur :

— Le MinMaxPlayer

— Le NegemaxPLayer

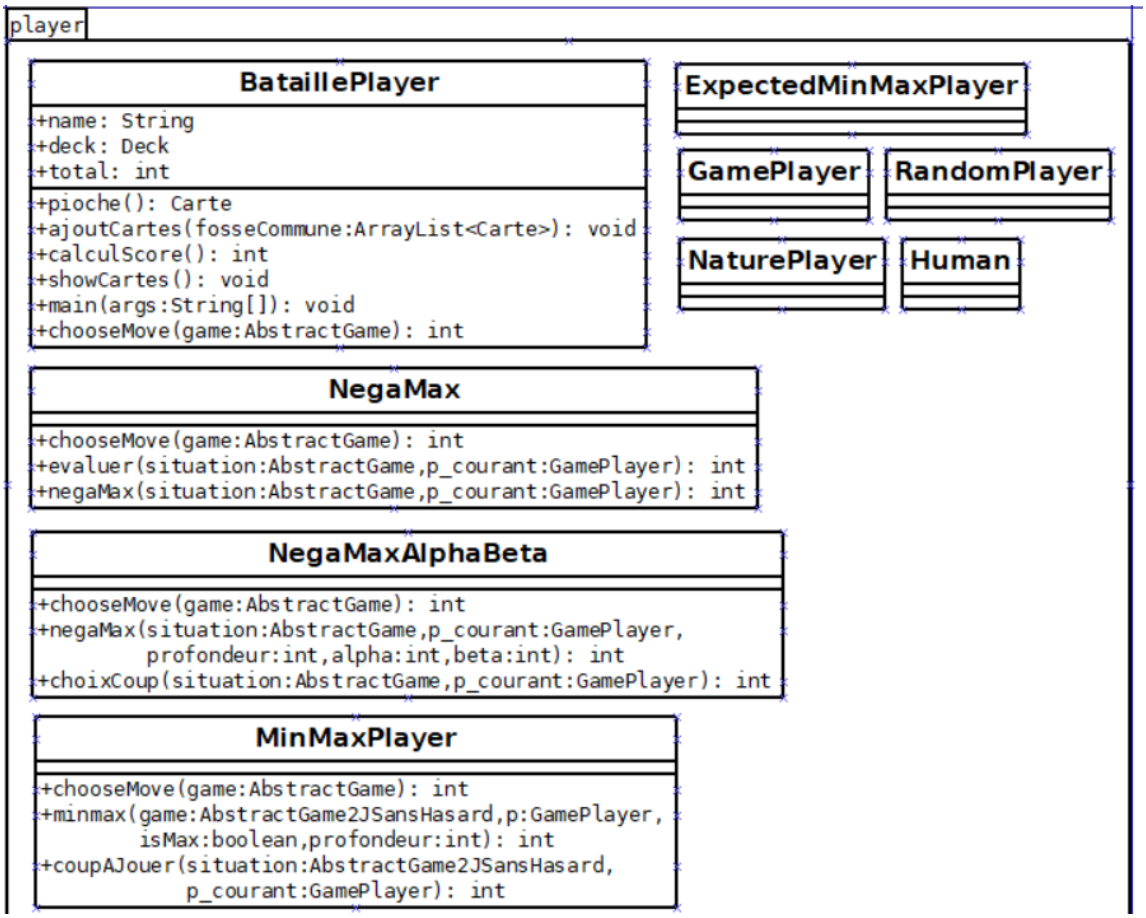
— Le ExpectiMinMaxPlayer

— Le RandomPlayer

— Le Human

— BataillePlayer

Tout ces différents joueur possède une propriété transverses qui est la méthode chooseMove() spécifiée dans l'interface GamePlayer.Seul le BataillePlayer n'implémente pas cette interface car le jeu de la bataille possède son propre moteur de jeu avec un seul type de joueur possible. Ce package va donc gérer les différents joueurs implémentés.



4.1.3 Le package orchestration

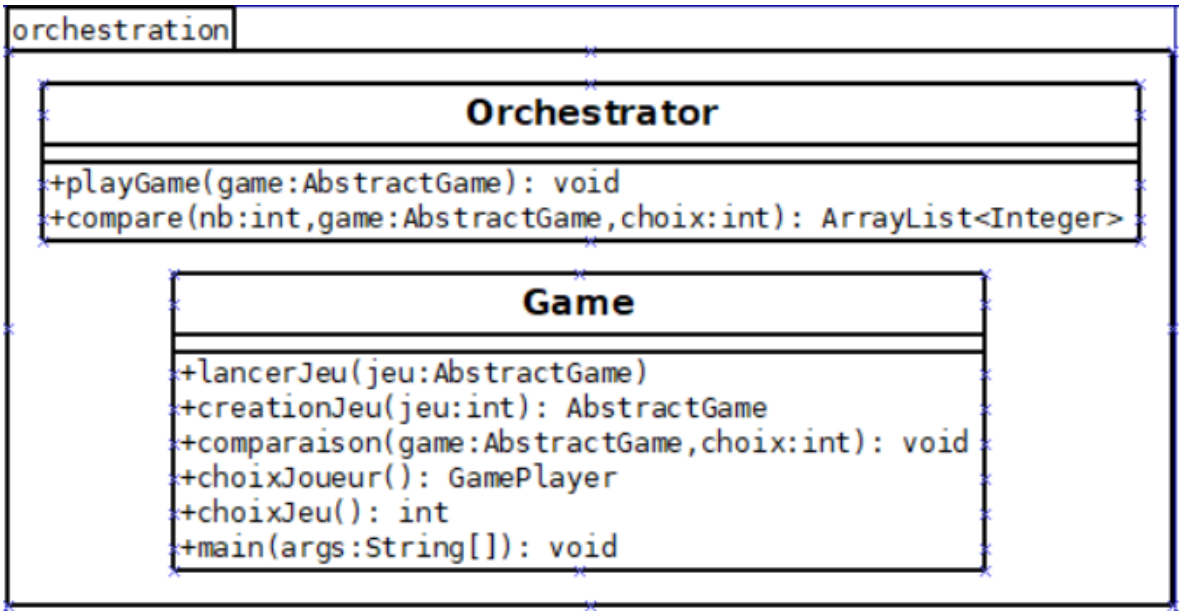
Ce package est le moteur pour les différents jeux sauf la bataille. Il possède deux classes :

— La classe Game

— La classe Orchestrator

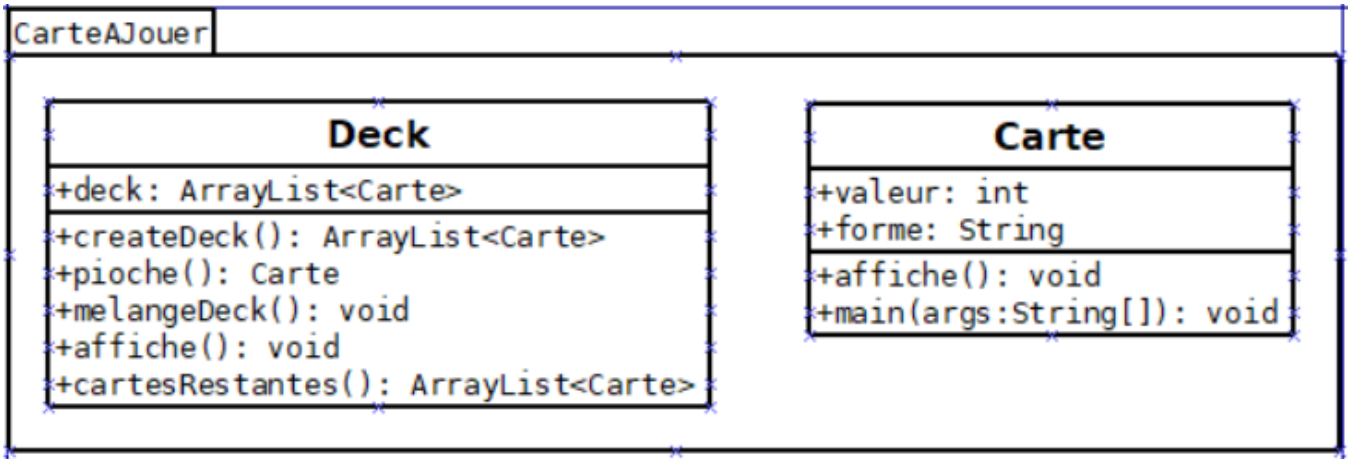
La classe Game est celle que l'on va lancer pour exécuter le programme, il s'agit en fait d'un menu de sélection dans lequel on va choisir le jeu auquel on souhaite jouer et les types de joueurs qui joueront a ce jeu. Elle va ensuite appeler la méthode playGame de la classe Orchestrator.

La classe Orchestrator est le moteur de jeu. En effet avec les méthodes génériques de la classe AbstractGame elle va exécuter les jeux selon les différentes règles qui elles sont implémentées dans les classes des différents jeux.

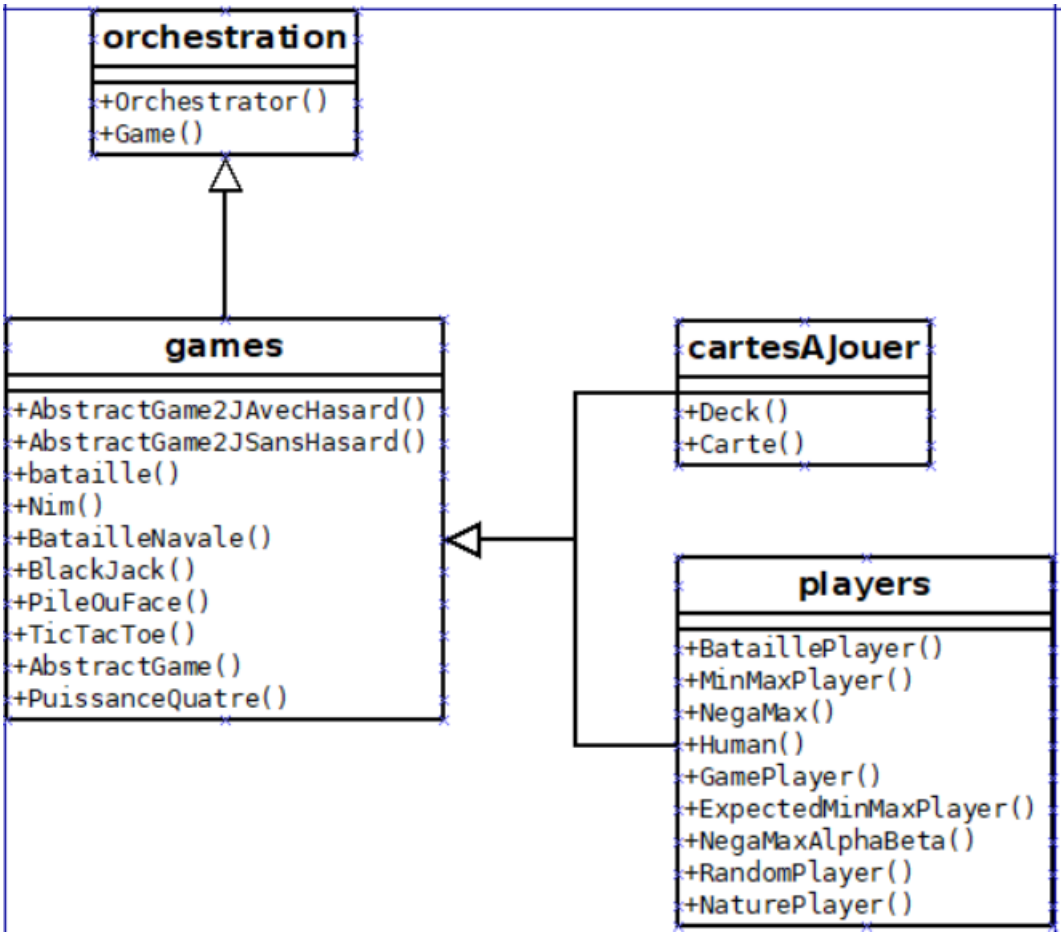


4.1.4 Le package carteAJouer

Ce package contient les deux classes Carte et Deck qui ne sont utiles que pour le jeu du Black Jack et le Bataille. L'objet Carte prends deux arguments qui sont sa valeur allant de 1 a 14. Et la forme de cette dernière, dans notre programme nous avons réduit le Deck de carte a seulement deux forme. Le deck quant à lui est juste une ArrayList des différentes cartes



4.1.5 Diagramme des packages



4.2 Éléments importants du programme

4.2.1 Le moteur de jeu

Le moteur de jeu est contenu dans la classe Ochestrator du package game, il s'agit simplement lorsqu'un jeu est initialisé via la classe game du package orchestration d'exécuter la méthode jouerUnCoup() tant que la la méthode isOver() retournera false. Une fois que que le jeu est terminé et que la méthode isOver() retourne true, la méthode getWinner() est appelée pour retourner le gagnant du jeu.

4.2.2 La classe AbstractGame

La classe AbstractGame est la classe abstraite de laquelle va découler les deux classes abstraites AbstractGame2JSansHasard et AsbtractGame2JAvecHasard respectivement faites pour les jeux à information complète et les jeux à information incomplète.
Il était nécessaire d'effectuer la distinction entre les deux types de jeux puisque pour les jeux à information complète implémentés il suffit d'exécuter le coup choisis via l'appel de la méthode execUnCoup() et de changer le joueur courant. Tandis que pour les jeux à information non complète il faut faire appel à un joueur de type nature qui va simuler l'action du hasard. On ne pouvait donc pas avoir la même méthode jouerUnCoup() générique de la classe abstraite AbstractGame2JSansHasard, par conséquent pour les jeux à informations incomplète la méthode sera implémentée directement au sein des classes de jeu

4.2.3 Utilisation du programme du programme

Pour lancer le programme, après la compilation il faut simplement exécuter la classe Game du package du même nom. La méthode creationJeu() est ensuite appelée avec comme argument la méthode choix jeux qui permet à l'utilisateur de choisir lequel des jeux il souhaite lancer, ensuite l'utilisateur devra juste choisir les types des deux joueurs pour la partie entre "human", "random" et les différents intelligences artificielles.

5 Les Intelligences artificielles

Les intelligences artificielles qui ont étés implémentées sont au nombres de trois, deux pour les jeux a information complète et une pour les jeux à information incomplète

5.1 Intelligences artificielles pour les jeux à information complète

5.1.1 MinMax

L'algorithme MinMax est un algorithme qui va s'appliquer pour les jeux à somme nulle et à information complète. Le principe de cet algorithme est de parcourir les noeuds de l'arbre représentant le jeu afin de déterminer quel coups est celui qui va minimiser les pertes du joueur. Les différents coups possibles vont se voir attribuer une valeur via un calcul d'heuristique, ce qui est possible puisqu'il s'agit de jeu à information complète donc on connaît les gains des chaque coup. Cet algorithme possède néanmoins une certaines limite qui réside dans la taille de l'arbre, si l'arbre est trop vaste il va falloir parcourir toutes les branches de l'arbre ce qui peut prendre un temps considérables pour le point de vue de la machine.

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth − 1, TRUE))
        return value
```

5.1.2 NegaMax

L'algorithme NegaMax découle de l'algorithme MinMax, il est lui aussi fait pour les jeux à somme nulle et à information complète. Il se base sur la simple règle qui dit que : le maximum entre a et b est égal a moins le minimum de moins a et moins b

$$\max(a,b) = -\min(-a,-b)$$

On peut appliquer cette règle car il opère sur des jeux à somme nulle, on peut en déduire que la valeur heuristique pour un joueur est égale à l'opposée de la valeur heuristique de son adversaire. Cet algorithme simplifie l'implémentation puisqu'il ne nécessite qu'un seul même bloc d'instruction appelé récursivement en prenant l'opposé pour l'adversaire.

```
function negamax(node, depth, color) is
    if depth = 0 or node is a terminal node then
        return color × the heuristic value of node
    value := −∞
    for each child of node do
        value := max(value, negamax(child, depth − 1, −color))
    return −value
```

5.2 Intelligence artificielle pour les jeux à information incomplète

5.2.1 ExpectiMinMax

L'algorithme ExpectiMinMax est une variante de l'algorithme minmax présenté ci-dessus, il va être identique au MinMax sauf qu'il s'applique aux jeux à somme nulle mais ici à information incomplète. En effet dans un jeu à information incomplète comme ceux implémentés dans notre programme on ne peut pas connaître ses gains sans avoir joué un coup. Dans l'algorithme ExpectiMinMax on va donc ajouter un cas de figure ou c'est la nature qui va jouer, autrement dit le hasard avec l'espérance de gains potentielle pour un coup. L'algorithme va se comporter comme le MinMax en parcourant tout les noeuds de l'arbre et à chaque fois qu'il va parcourir la branche d'un coup il va tomber sur un noeuds qui n'est pas une situation mais l'action de la nature. Arrivé sur ce noeud la l'algorithme va effectuer un calcul d'espérance afin de choisir le meilleur coup potentiel.

```
function expectiminimax(node, depth)
    if node is a terminal node or depth = 0
        return the heuristic value of node
    if the adversary is to play at node
        // Return value of minimum-valued child node
        let α := +∞
        foreach child of node
            α := min(α, expectiminimax(child, depth-1))
    else if we are to play at node
        // Return value of maximum-valued child node
        let α := -∞
        foreach child of node
            α := max(α, expectiminimax(child, depth-1))
    else if random event at node
        // Return weighted average of all child nodes' values
        let α := 0
        foreach child of node
            α := α + (Probability[child] × expectiminimax(child, depth-1))
    return α
```

5.3 Élagage Alpha-Bêta

Le principe de de l'élagage Alpha-Bêta est d'éviter à un algorithme de type MinMax de parcourir tout les noeuds de l'arbre quand cela n'est pas nécessaire. Il existe deux coupures pour cette élagage :

— la coupure Alpha

— la coupure Bêta

La coupure Alpha va s'effectuer si la valeur d'un noeud qui est l'enfant d'un noeud de type Min. Si ce noeud enfant possède une valeur inférieur à son ancêtre alors ses frères n'ont pas besoins d'êtres parcourus.
La coupure Bêta va s'effectuer si la valeur d'un noeud qui est l'enfant d'un noeud de type Max. Si ce noeud enfant possède une valeur supérieure à son ancêtre alors ses frères n'ont pas besoin d'êtres parcourus.

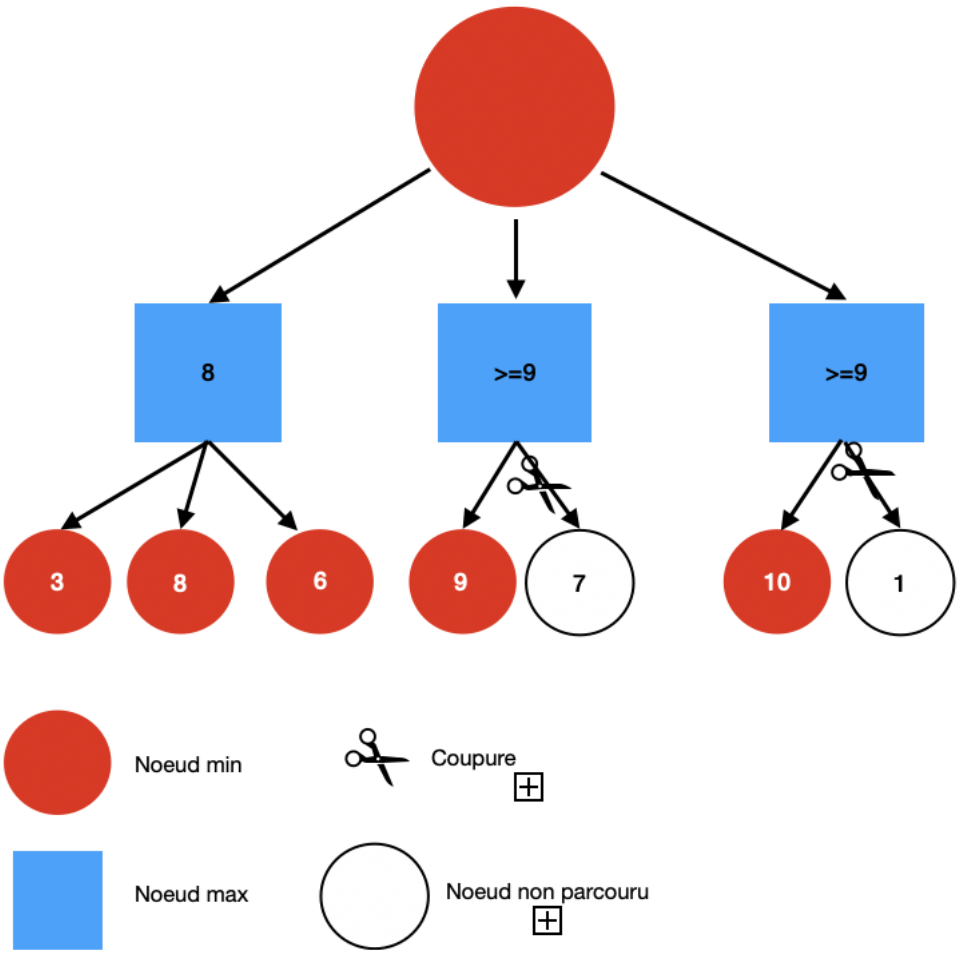


FIGURE 1: Schéma de coupure Bêta

Pour que l'élagage Alpha-Bêta soit le plus efficace possible il faut que les différents jeux possèdent une fonction d'évaluation qui soit la plus précise possible. Il parait peu pertinent d'écrire une fonction d'évaluation très précises pour le morpion étant donné que les différentes situation possibilités de coup sont au nombre de 9 au départ et s'amenuisent au fil des tours. De plus c'est situations sont souvent similaires en termes de gains. L'élagage Alpha-Bêta serait parfaitement légitime pour un jeu comme le jeu d'échec où les différentes situations peuvent avoir des valeurs heuristiques bien différentes.

5.4 L'obtention d'un coup à jouer

Pour obtenir un coup à jouer avec les différents algorithme, il suffit de "dérouler" une situation avec les différents coups possibles et ensuite appeler les méthodes des algorithmes qui vont par recursivité trouver le meilleur coup a jouer. Grâce au premier coup simulé on va pouvoir récupérer celui qui est le plus optimal pour le tour actuel.

6 Étude statistiques des résultats des parties

Pour étudier l'efficacité des différents algorithme nous avons dans la classe game décidés d'implémenter un méthode pour lancer un nombre de fois un jeu avec les différentes intelligences artificielles. Cette méthode nous retourne une ArrayList comprenant le nombre de victoire du joueur 1, le nombre d'égalité, le nombre de victoire du joueur 2. Cette méthode affiche aussi le temps d'exécution de ces parties. Les résultats de ces tests sont disponibles dans le tableau en annexe. Dans certains jeux, être le premier joueur à jouer peut donner un certain avantage, il nous à donc paru pertinent d'effectuer les tests où les différentes intelligences artificielles occuperaient les deux positions de départ.

6.1 Random vs Random

Pour le jeu du morpion on constate que jouer en premier génère un avantage, pour 1000 parties, 606 vont êtres gagnées par le premier joueur contre 272 pour le joueur 2. Le temps d'exécution de ces 1000 parties est significativement inférieur que lors d'une partie rencontrant une intelligence artificielle. Pour ce qui est du jeu de Nim nous avons fait l'expérience pour 10 000 parties et l'aléatoire prend le dessus avec 4986 victoires pour le premier joueur pour 0 égalités. Le résultat de victoire pour le jeu de nim tends vers 50% pour n'importe lequel des joueurs aléatoires. Pour le puissance 4 on observe un avantage conséquent pour le premier joueur qui pour 10 000 parties en remporte 9053 soit environ 90%.

6.2 NegaMax vs NegaMax

Pour le morpion sur 1000 partie jouées entre deux joueurs negamax il n'y à eu que des égalités. Aucun de ces deux joueurs n'a réussi à contrer la stratégie de l'autre ce qui semble logique puisqu'ils disposent de la même. Pour le puissance 4 en revanche le premier joueur a jouer remporte toute les partie ce qui est cohérent avec les résultats de deux joueurs random qui montrent bien que jouer en premier confère un avantage de taille. Pour le jeu de nim la aussi le premier joueur remporte toutes les parties jouées ce qui la aussi montre un avantage a jouer en premier quand le joueur possède une stratégie.

6.3 NegaMax vs Random

Le joueur Negamax va lui gagner 76% des parties contre le joueur aléatoire pour moins de 10% de défaites au jeu du morpion, pour ce qui est du puissance 4, il va gagner 92% des parties contre moins de 10% de défaites. Pour le jeu de nim il gagnera toutes les parties. Il apparaît que l'algorithme negamax est efficace contre le joueur aléatoire cependant il possède une certaine limite qui est le temps de calcul qui peut être assez long.

6.4 Random vs NegaMax

Nous avons cette fois ci réalisés des tests en laissant le joueur aléatoire jouer le premier coup car dans certains jeux jouer en premier octroie un avantage. Les résultats sont sensiblement identiques que lorsque que c'est le joueur negamax qui commence. On constate tout de même que le joueur aléatoire ne gagne pas de partie pour le morpion alors que précédemment il en gagnait environ 80. Le puissance 4 possède des résultats similaires, pour le jeu de nim le joueur aléatoire va gagner 3 parties contre 0 au préalable. Ces résultats montrent que le joueur negamax prendra la majeure partie du temps l'avantage sur un adversaire aléatoire.

6.5 MinMax vs Random

Les performances du joueur MinMax contre un joueur aléatoire sont sensiblement similaires. Avec 93% de victoires pour le morpion, 85 pour le puissance 4 et 96% pour le jeu de nim. La grosse différence et le gros avantage que l'on observe va etre le temps d'exécution qui est nettement plus court pour le minmax que pour le negamax. Par exemple, pour 1000 partie du jeu de nim d'un joueur negamax contre un joueur aléatoire il faut compter en moyenne 24 241 ms soit environ 24 secondes, quand pour 10 000 parties d'un joueur minmax contre un joueur aléatoire il faut 3063 ms soit environ 3 secondes.

6.6 MinMax vs Negamax

Pour le morpion les résultats sont similaires a ceux de 1000 parties de deux joueurs negamax avec 1000 égalités, cependant le temps d'exécution de ces 1000 parties est amputé d'environ 4 secondes. Le joueur MinMax à remporté toutes les parties de puissances 4 avec un temps plus, apres simulation d'une partie on constate que presque tout les coups sont joués ce qui peux expliquer un temps d'exécution plus long. Pour le jeu de nim le minmax va remporter toute les parties avec la aussi un temps d'exécution plus faible d'environ 0.7 secondes.

6.7 NegaMax avec élagage Alpha-Bêta vs random

Non avons également testés le NegaMax avec un élagage alpha béta contre un joueur random. L'élagage Alpha Beta n'est pas pertinent puisque nous utilisons les mêmes valeurs heuristiques que pour le NegaMax classique. Ici nous allons juste changer la profondeur de celui ci afin de voir les différents résultats et surtout les vitesses d'exécution

6.7.1 Profondeur de 1

Pour le jeu du morpion nous avons lancés 100 000 parties contre un joueur aléatoire, les résultats sont un peu en dessous de ceux du negamax mais le temps de calcul lui est fortement diminué avec environ 5secondes pour 100 000 partie contre environ 150 secondes pour 1000 partie pour le negamax.Remis sur 1000 parties cela donne un temps d'exécution 3000 fois inférieur pour le negaMax avec une profondeur de 1. Pour le puissance 4 les résultats sont aussi similaires avec un temps d'exécution similaire également. Pour le jeu de nim les résultats sont 8% en dessous du negamax classique mais un temps de calcul divisé par 10 pour 100 fois plus de parties.

6.7.2 Profondeur de 2

Nous avons aussi lancés 100 000 parties de morpion avec des résultats 1% moins efficaces que pour le négamax mais un temps de calcul 600 fois plus court pour le negamax avec une profondeur de 2. Pour le puissance 4 les résultats sont en deça avec un temps de calcul plus long. Pour le jeux de nim les résultats sont similaires à ceux avec une profondeur de 1 mais toujours avec un temps de calcul divisé par 1000.

6.7.3 Profondeur de 6

Pour le morpion les résultats sont 1% meilleurs que le negamax classique avec un temps de calcul divisé par 3. Pour le puissance 4 le temps de calcul est trop grand pour être calculé. Pour le jeu de nim les résultats sont les même que pour le negamax classique avec un temps d'exécution divisé par 14.

6.8 NegaMax avec élagage Alpha-Bêta vs Negamax

Pour le jeu du morpion les résultats vont êtres identiques a ceux d'une partie ou deux joueurs negamax s'affrontent mais avec un temps de calcul divisé par 25. Pour le puissance 4 le negamax l'emporte a chaque partie pour des profondeurs de 1,2 et 3. Pour le jeu de nim l'avantage du premier joueur à jouer ne sera retrouvé qu'à partir de la profondeur égale à 3 avec un temps de calcul 24 fois moins élevé que pour une partie où deux joueurs negamax classiques s'affrontent.

6.9 Synthèse

	Avantages	**Incovenient**
Negamax	-Plus simple à implémenter que le MinMax -Facilement améliorable avec un élagage alpha bêta	-Parcours de tout l'arbre de jeu donc un temps d'exécution conséquent
MinMax	-Facile a mettre en place pour des jeux simples -Facilement améliorable avec un élagage alpha bêta	-Parcours de tout l'arbre de jeu donc temps d'exécution conséquent
Negamax avec Elegage alpha Beta	-Simple d'implémentation -Temps d'exécution réduit	-Nécessite pour les jeux plus complexes des calculs d'heuristiques précis. -Nécessite de bien choisir la profondeur en fonction de chaque jeu
ExcpectiMinMax	-Prend en compte la part de hasard dans les différents jeu -Facilement améliorable avec un élagage alpha bêta	-Nécessite d'avoir une structure de jeu bien définie -Parcours tout l'arbre de jeu donc temps d'exécution conséquent

Pour cet étude de statistique il semblait peu pertinent de comparer l'algorithme ExpectiMinMax avec un joueur aléatoire, notamment parceque nos jeux à information incomplète ne possède que deux coups qui sont jouer ou non, il y aurait donc 50% de chance que le joueur aléatoire refuse de jouer des le premier tour ce qui donnerait des résultats peu pertinents. Nous ne pouvions pas non plus le comparer avec l'algorithme minmax car celui ci ne prend pas en compte la possibilité qu'il y ait un joueur nature au court d'une partie.

7 Amélioration

Il apparaît comme certain que de nombreuses améliorations sont possibles afin d'approfondir encore plus le sujet des intelligences artificielles pour des jeux

- ☐ Nous pourrions implémenter des jeux plus complexes comme le jeu d'échec ou de dame afin d'avoir une meilleure utilisation de l'élagage alpha-bêta
- ☐ Nous pourrions implémenter des calculs d'heuristiques plus précis encore une fois afin d'utiliser a meilleur escient l'élagage alpha-bêta
- ☐ Nous aurions souhaités avoir plus de temps pour réalisés ce programme avec une interface graphique afin que les jeux soient plus agréables pour l'utilisateur
- ☐ Nous pourrions rechercher d'autres types d'intelligences artificielles qui fonctionneraient pour nos jeux
- ☐ Nous avons aussi un problème dans le minmax player qui fait qu'il va trouver un coup optimal a jouer que si il débute la partie, nous n'avons pas pu parvenir a trouver la cause de ce problème, ce qui explique que nous n'avons pas essayé de le faire jouer en deuxième dans nos comparaisons entre les différents joueurs
- ☐ Le joueur expectiminmax n'est pas fonctionnel, en effet pour nos différents jeux il va toujours retourner 0, soit un refus de jouer qui va arrêter le jeu et faire gagner l'adversaire, nous avons essayé plusieurs fois de changer nos calculs d'heuristiques en vain.

8 Conclusion

À l'aide de nos études sur les différents résultats des types de joueurs, nous constatons que l'algorithme MinMax et son dérivé le NégaMax offre des résultats probant en terme de gains de partie. Cependant, on constate aussi qu'il est possible d'accroître drastiquement leurs performances en termes de vitesse d'exécution avec notamment l'élagage alpha-bêta et le choix d'une profondeur de recherche optimale pour chacun des jeux.
Grâce a ce projet et en dépit des conditions sanitaires compliquées, ce projet nous à permis d'avoir une bonne compréhension des algorithmes basiques d'intelligences artificielles pour des jeux à information complète. Il nous à également permis de mieux se représenter la façon dont un algorithme parcourait les arbres d'un jeu et par conséquent de mieux se représenter la modélisation d'un jeu sous forme d'arbre. L'utilisation d'intelligence artificielles connaît une tres forte croissance et ceux dans de multiples domaines, comme l'automobile, l'aérospatiale et bien d'autres encore. Il apparaît comme essentiel pour des futures personnes travaillant dans le domaine de l'informatique d'avoir quelques connaissances dans ce domaine qui selon nous représente le futur. Il est clair que toutes les intelligences artificielles pour les jeux à somme nulle découlent de l'algorithme MinMax, il s'agit a chaque fois de déclinaisons.

9 Sources

- ☐ Wikipedia contributors. (2021, 9 avril). Élagage alpha-bêta. Wikipedia.
- ☐ Wikipedia contributors. (2020, 21 avril). Expectiminimax. Wikipedia.
- ☐ Wikipedia contributors. (2021, mai 2). Negamax. Wikipedia.
- ☐ Cours sur le MinMax de l'université de Lille.

10 Annexe

	Morpion	Puissance 4	Nim
NegaMax vs Random	762/153/85 149 969ms	921/0/79 1 266ms	1000/0/0 24241ms
Random vs Negamax	0/194/806 17455ms	980/0/20 356ms	3/0/997 9440ms
NegaMax vs NegaMax	0/1000/0 165 548ms	1000/0/0 2185ms	1000/0/0 1803 ms
Minmax vs Negamax	0/1000/0 119 822ms	1000/0/0 14 482 ms	1000/0/0 1147ms
Random vs Random	606/122/272 12ms	9053/0/947 290ms	4986/0/5014 1760 ms
MinMax vs Random	936/48/16 86 ms	856/0/144 291ms	9662/0/338 3063 ms
NegaMaxAlphaBeta vs Random Profondeur = 1	73730/14828/11442 5117ms	848/0/152 324 ms	92532/0/7468 2394 ms
NegaMaxAlphaBeta vs Random Profondeur = 2	75231/13431/11338 26553 ms	835/0/165 1471 ms	92534/0/7466 3000ms
NegaMaxAlphaBeta vs Random Profondeur = 6	776/129/95 48 147 ms	Temps trop long	1000/0/0 1716ms
NegaMaxAlphaBeta vs Negamax Profondeur = 1	0/1000/0 6374 ms	0/0/1000 837ms	0/0/1000 1627ms
NegaMaxAlphaBeta vs Negamax Profondeur = 2	0/1000/0 19954 ms	0/0/1000 2595ms	0/0/1000 1092ms
NegaMaxAlphaBeta vs Negamax Profondeur = 3	0/1000/0 20813ms	0/0/1000 13172	1000/0/0 1139ms