RÉPUBLIQUE DU BÉNIN

MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR
ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITÉ D'ABOMEY-CALAVI

**INSTITUT DE FORMATION ET DE
RECHERCHE EN INFORMATIQUE**

BP 526 Cotonou Tel : +229 21 14 19 88
http://www.ifri-uac.net Courriel : contact@ifri.uac.bj

Mémoire pour l'obtention du Master en Informatique

Détermination des paramètres d'une heuristique générique pour les jeux de type « n-alignés » par apprentissage automatique

Abdel-Hafiz ABDOULAYE

hafiz_abdoulaye@yahoo.fr

Sous la supervision de :

Dr Ing. Vinasétan Ratheil HOUNDJI

&

Dr Eugène C. EZIN, Maître de Conférence

Année Académique : **2016-2017**

Table des matières

Dédicace **iv**

Remerciements **v**

Liste des figures **vi**

Liste des tableaux **viii**

Liste des algorithmes **ix**

Liste des sigles et abréviations **x**

Résumé **1**

Abstract **2**

Introduction **3**

1 État de l'art **5**

1.1 Les jeux de type « n-alignés » . . . 5
1.1.1 Généralités . . . 5
1.1.2 Les règles . . . 6
1.1.3 Notion de menace . . . 9
1.1.4 Heuristique générique . . . 11
1.2 Généralités sur les algorithmes minimax et alpha-bêta . . . 13
1.2.1 L'algorithme minimax . . . 13
1.2.2 L'algorithme alpha-bêta . . . 16
1.3 Apprentissage automatique . . . 19
1.3.1 Types d'apprentissage automatique . . . 19
1.3.2 Approfondissement de la notion d'apprentissage par renforcement . . . 21

2 Matériel et méthodes **27**

2.1 Matériel . . . 27

2.2 Le cycle en V 28
2.3 Modélisation 29
2.3.1 Le plateau : la classe *Board* 29
2.3.2 Le nœud : la classe *Node* 31
2.3.3 Le joueur : la classe *AlphaBeta* 32

3 Solutions **34**
3.1 Détermination de la méthode d'apprentissage 34
3.2 Amélioration de l'heuristique générique 35

4 Résultats et discussion **42**
4.1 Présentation des performances de la solution proposée 43
4.1.1 Test des performances de la solution proposée pour Gomoku 44
4.1.2 Test des performances de la solution proposée pour Tic Tac Toe 48
4.1.3 Test des performances de la solution proposée pour Connect four 52
4.2 Discussion 54

Conclusion et perspectives **56**

Bibliographie **57**

Webographie **58**

Dédicace

À

Mes parents Abdou-Razizou ABDOULAYE et Clarisse ALI YERIMA qui ont toujours répondu présents quand j'ai eu besoin d'eux. Je ne saurais exprimer ce que vous êtes pour moi. Trouvez ici la satisfaction morale pour l'amour que vous m'avez donné et les sacrifices que vous avez consentis pour moi.

Remerciements

Sincères remerciements à tous ceux qui ont participé à l'élaboration de ce travail particulièrement :

- Messieurs Eugène C. EZIN et Vinasétan Ratheil HOUNDJI mes superviseurs pour tout ce qu'ils m'ont apporté dans la concrétisation de ce travail ;
- Tous les enseignants de l'IFRI pour avoir accepté de partager une partie de leurs connaissances avec moi ;
- Tous mes camarades étudiants de l'IFRI pour leur sens de partage ;
- A tous les membres de la Famille ABDOULAYE pour les divers soutiens qu'ils m'apportent ;
- A mon beau-frère Monsieur Walis BIO OBEGUI pour ses efforts répétés.

Table des figures

1.1 Une partie de Gomoku [34] 7
1.2 Une partie de Tic Tac Toe [22] 7
1.3 Une partie de Connect Four [19] 8
1.4 Menaces de type *quatre* [26] 10
1.5 Exécution de l'algorithme *minimax* [2] 15
1.6 Elagage *alpha-bêta* [25] 17
1.7 Interaction *agent-environnement* [7] 22
1.8 Passage d'un état à un autre [7] 24

2.1 Cycle de développement en V [32] 28
2.2 Diagramme de classes du système des jeux de type «n-alignés» basé sur alpha-bêta 30
2.3 Aperçu de la classe Board 31
2.4 Aperçu de la classe Node 32
2.5 Aperçu de la classe AlphaBeta 33

3.1 Méthode de mise à jour du paramètre de la menace la plus importante du joueur 40

Liste des tableaux

4.1 Gomoku : heuristique améliorée vs heuristique brute pour $n = 3$, test n°1 44
4.2 Gomoku : heuristique améliorée vs heuristique brute pour $n = 3$, test n°2 44
4.3 Gomoku : heuristique améliorée vs heuristique brute pour $n = 4$, test n°3 45
4.4 Gomoku : heuristique améliorée vs heuristique brute pour $n = 4$, test n°4 45
4.5 Gomoku : heuristique améliorée vs heuristique brute pour $n = 5$, test n°5 45
4.6 Gomoku : heuristique améliorée vs heuristique brute pour $n = 5$, test n°6 45
4.7 Gomoku : heuristique améliorée vs heuristique de Shevchenko pour $n = 3$, test n°7 46
4.8 Gomoku : heuristique améliorée vs heuristique de Shevchenko pour $n = 3$, test n°8 46
4.9 Gomoku : heuristique améliorée vs heuristique de Shevchenko pour $n = 4$, test n°9 47
4.10 Gomoku : heuristique améliorée vs heuristique de Shevchenko pour $n = 4$, test n°10 47
4.11 Gomoku : heuristique améliorée vs heuristique de Shevchenko pour $n = 5$, test n°11 47
4.12 Gomoku : heuristique améliorée vs heuristique de Shevchenko pour $n = 5$, test n°12 48
4.13 Tic tac toe : heuristique améliorée vs heuristique brute pour $n = 3$, test n°13 . . . 48
4.14 Tic tac toe : heuristique améliorée vs heuristique brute pour $n = 3$, test n°14 . . . 48
4.15 Tic tac toe : heuristique améliorée vs heuristique brute pour $n = 4$, test n°15 . . . 49
4.16 Tic tac toe : heuristique améliorée vs heuristique brute pour $n = 4$, test n°16 . . . 49
4.17 Tic tac toe : heuristique améliorée vs heuristique brute pour $n = 5$, test n°17 . . . 49
4.18 Tic tac toe : heuristique améliorée vs heuristique brute pour $n = 5$, test n°18 . . . 49
4.19 Tic tac toe : heuristique améliorée vs heuristique de Chua Hock Chuan pour $n = 3$, test n°19 50
4.20 Tic tac toe : heuristique améliorée vs heuristique de Chua Hock Chuan pour $n = 3$, test n°20 50
4.21 Tic tac toe : heuristique améliorée vs heuristique de Chua Hock Chuan pour $n = 4$, test n°21 50
4.22 Tic tac toe : heuristique améliorée vs heuristique de Chua Hock Chuan pour $n = 4$, test n°22 51
4.23 Tic tac toe : heuristique améliorée vs heuristique de Chua Hock Chuan pour $n = 5$, test n°23 51

4.24 Tic tac toe : heuristique améliorée vs heuristique de Chua Hock Chuan pour $n = 5$, test n°24 . . . 51
4.25 Connect four : heuristique améliorée vs heuristique brute pour $n = 3$, test n°25 . 52
4.26 Connect four : heuristique améliorée vs heuristique brute pour $n = 3$, test n°26 . 52
4.27 Connect four : heuristique améliorée vs heuristique brute pour $n = 4$, test n°27 . 52
4.28 Connect four : heuristique améliorée vs heuristique brute pour $n = 4$, test n°28 . 53
4.29 Connect four : heuristique améliorée vs heuristique brute pour $n = 5$, test n°29 . 53
4.30 Connect four : heuristique améliorée vs heuristique brute pour $n = 5$, test n°30 . 53

Liste des Algorithmes

1 Algorithme de catégorisation d'une menace [1] 11
2 Calcul des valeurs du tableau de coefficients des menaces moins importantes [1] . 13
3 Algorithme minimax [2] 16
4 Algorithme alpha-bêta [2] 18
5 Algorithme TD(0) [7] 23
6 Algorithme Sarsa [7] 24
7 Algorithme Q-learning [7] 25

8 Algorithme de mise à jour des paramètres 38

Liste des sigles et abréviations

α-β : Alpha-bêta

IA : Intelligence Artificielle

MCTS : Monte Carlo Tree Search

TD : Temporal Difference

Résumé. Les jeux en situation d'adversité font l'objet de nombreuses recherches dans le domaine de l'Intelligence Artificielle (IA). Parmi ces jeux, nous distinguons les jeux de la catégorie « n-alignés ». Les travaux effectués par Gael AGLIN [1] ont permis la détermination d'une heuristique pouvant s'appliquer à chacun des jeux de type « n-alignés ». Les paramètres de cette heuristique, fixés expérimentalement, n'assurent pas toujours une bonne évaluation des coups possibles. L'idéal est de trouver une manière d'adapter les paramètres à chaque jeu. Pour la résolution de ce type de jeu, des algorithmes de recherche basés sur le parcours d'arbre comme alpha-bêta sont utilisés.

Dans ce travail, nous proposons une méthode d'apprentissage automatique permettant à l'heuristique générique de s'adapter au jeu en résolution. Nous avons utilisé l'algorithme Q-learning pour déterminer de meilleurs paramètres pour cette heuristique. Nous l'associons à alpha-bêta en limitant la profondeur. Les résultats des tests montrent qu'en moyenne notre approche est meilleure que l'heuristque générique brute.

Mots clés : *n-alignés, heuristique, alpha-bêta, apprentissage automatique, Q-learning*.

Abstract. Games in situation of adversity are subject of much research in the field of artificial intelligence. Among these games, we distinguish the games of the « n-aligned » category. The work carried out by Gael AGLIN [1] permits to determine an heuristic which can be used for each of the « n-aligned » games. The parameters of this heuristic, fixed experimentally, do not always provide a good evaluation for the possibles actions. The best way is to find a manner to adapt the parameters to each game. For the resolution of this type of game, search algorithms based on the tree path as alpha-beta are used.

In this work, we propose a machine learning method that permits the adaptation of the heuristic to each game in resolution. We have used the Q-learning algorithm to determine better parameters for this heuristic. We associate it with alpha-beta by limiting the depth. The results of the tests show that our approach is better on average than the gross generic heuristic.

Keys words: *n-alignés, heuristic, alpha-beta, machine learning, Q-learning*.

Introduction

L'intelligence artificielle est une branche de l'informatique qui a pour but de comprendre des entités intelligentes et d'en construire. Elle intervient dans une variété de domaines dont les jeux. Un jeu est un terrain d'expérimentation idéal pour l'IA. Les règles relativement simples et peu nombreuses des jeux à deux joueurs (chacun son tour), les situations bien définies modélisent un mode simplifié mais quand même intéressant. Ainsi, des méthodes de recherche basées essentiellement sur le parcours d'arbre comme alpha-bêta [6, 10] ont été proposées pour gagner plus facilement et plus rapidement ces jeux. La plupart de ces méthodes utilisent une fonction d'évaluation pour améliorer le résultat final. L'espace de recherche des jeux pouvant être très grand, il est difficile de parcourir tout l'arbre. Dans ce cas, l'utilisation d'une heuristique représente une alternative. Une heuristique est une méthode ou une fonction qui permet d'évaluer l'état du plateau en estimant les gains des joueurs afin d'aider dans le choix du coup le plus prometteur.

Récemment Gael AGLIN (Master thesis, IFRI 2016) [1] a défini une catégorie de jeux qu'il a appelé « n-alignés »[1]. Ce sont les jeux de plateaux à deux joueurs (voir section 1.1.1), à somme nulle (voir section 1.1.1), à coups alternatifs (voir section 1.1.1) qui consistent à poser à chaque coup une pierre de la couleur du joueur courant sur un plateau carré. L'environnement est complètement observable, déterministe et discret. Le joueur qui aura aligné consécutivement et exactement n pierres de sa couleur soit verticalement ou horizontalement ou diagonalement a gagné la partie. Pour ce type de jeux, une heuristique générique basée essentiellement sur la notion de menace (voir section 1.1.3) a été proposée. Cette heuristique fait une évaluation des coups qui dépend des paramètres associés aux menaces. Les paramètres ont donc des valeurs précises favorisant le choix du meilleur coup à un instant bien défini du jeu.

[1] La catégorie "test" de jeux que nous utilisons pour ce travail et qui est composée des jeux Tic Tac Toe, Gomoku, Connect Four, etc.

Problématique

Il faut noter que les paramètres de l'heuristique générique proposée ont été fixés expérimentalement. Ceci ne garantit pas toujours que ce sont de bons paramètres quel que soit le jeu. L'heuristique peut être améliorée par l'utilisation de l'apprentissage automatique.

Motivation

Ce travail permettra d'orienter les regards vers l'apprentissage automatique afin de permettre à l'heuristique générique de s'adapter au mieux à chaque jeu de notre catégorie.

Les tests ont été réalisés spécifiquement sur les jeux que sont *Gomoku* (voir section 1.1.2.1), *Tic Tac Toe* (voir section 1.1.2.2) et *Connect Four* (voir section 1.1.2.3). Nous précisons que notre étude ne s'arrête pas uniquement à la forme standard connue de ces jeux. Nous nous étendons à toutes redéfinitions personnelles de ces jeux qui modifient la taille du plateau tant qu'il est carré.

Objectifs

Nous proposons et expérimentons une approche qui, grâce à l'apprentissage automatique, permet de déterminer les paramètres de l'heuristique générique mentionnée plus haut afin d'avoir de meilleurs paramètres. De façon spécifique, il s'agira :

- d'étudier les différents types d'apprentissage automatique ;
- d'identifier lesquels pourraient être utilisés pour notre travail et choisir la méthode la mieux adaptée ;
- de mettre en œuvre la méthode choisie et l'intégrer à l'heuristique générique ;
- d'utiliser l'heuristique générique pour guider Alpha-bêta dans sa phase de simulation.

Organisation du travail

La suite de ce document est organisée comme suit. Le chapitre 1 fait l'état de l'art des recherches effectuées dans le cadre de la résolution des jeux en étude, de l'apprentissage automatique et de son utilisation dans les jeux. Le chapitre 2 présente le matériel ainsi que les méthodes que nous utilisons pour la mise en œuvre de la solution proposée et le chapitre 3 détaille ladite solution. Enfin, nous présentons dans le chapitre 4 les résultats obtenus après les tests de notre solution ainsi que l'analyse de ces résultats.

Chapitre 1

État de l'art

Résumé. Les jeux de type « n-alignés » sont une famille de jeux à somme nulle et à coups alternatifs pour laquelle une heuristique générique a été déterminée. Des algorithmes de recherche comme *alpha-bêta* sont utilisés pour les résoudre et peuvent être couplés avec l'heuristique générique en limitant la profondeur. L'apprentissage automatique est utilisé dans ces jeux pour en faciliter la résolution.

Introduction

Dans ce chapitre, nous définissons les jeux de type « n-alignés », faisons l'état de l'art des travaux effectués sur cette catégorie de jeu. Nous donnons le fonctionnement de l'algorithme minimax puis de l'une de ses formes appelée *alpha-bêta* et nous faisons le point des types et méthodes d'apprentissage automatique.

1.1 Les jeux de type « n-alignés »

1.1.1 Généralités

Notre étude porte sur une catégorie de jeux appelée « n-alignés ». Ce sont les jeux de plateaux à deux joueurs, à somme nulle, à coups alternatifs qui consistent à poser à chaque coup une pierre de la couleur du joueur courant (blanc ou noir) sur un plateau carré de taille $X \times X$. L'environnement est complètement observable, déterministe et discret. Ce sont des jeux combinatoires [9]. Le joueur qui aura aligné consécutivement et exactement n pierres de sa couleur soit verticalement ou horizontalement ou diagonalement a gagné la partie.

On parle de somme nulle lorsqu'on somme les gains des joueurs en fin de partie dans le cas d'un jeu à deux joueurs où les joueurs sont en rivalité. En d'autres termes lorsque l'un gagne, l'autre perd $(1, -1)$ ou la partie est nulle $(0, 0)$. Le jeu a une propriété de coups alternatifs lorsque chaque joueur joue un coup à tour de rôle. Il est à deux joueurs lorsqu'il fait intervenir deux adversaires. On parle de déterminisme lorsque l'état suivant du jeu peut être déterminé grâce à son état courant et l'action du joueur.

Plusieurs jeux qui correspondent à cette catégorie sont connus dans la littérature. Voici une liste non exhaustive de ces jeux :

- Renju [15] ;
- Tic Tac Toe [20, 21] ;
- Gomoku ou n-in-row [13, 14] ;
- Pente [16, 17] ;
- Connect Four ou Puissance 4 [19, 18] ;
- Atari-Go [23].

1.1.2 Les règles

Dans le cadre de ce travail, nous allons présenter spécifiquement Gomoku, Tic Tac Toe et Connect Four.

1.1.2.1 Gomoku

Gomoku est un jeu de plateau originaire du Japon. Il se joue sur un plateau de 15×15 ou 19×19. Dans le jeu, les joueurs jouent à tour de rôle en plaçant chacun une pierre de sa couleur sur le plateau jusqu'à ce qu'un joueur ait réussi à connecter cinq (05) pierres dans une rangée (horizontale, verticale ou diagonale). Le joueur qui commence la partie est le Noir. Contrairement aux jeux de plateau tels que les échecs, dames, etc. Gomoku est joué sur un plateau de Go (le goban) (Voir figure 1.1). Cela signifie que les pierres sont placées sur les intersections et non à l'intérieur des cellules. Comme le plateau est une matrice 18×18, il y a en fait 19×19 places jouables sur le plateau. Lorsqu'aucun des joueurs ne gagne et que le plateau est rempli, la partie est alors déclarée nulle.

Le jeu possède de nombreuses variantes. La variante qui nous intéresse dans ce travail est celle qui donne toute liberté aux deux joueurs c'est-à-dire le *freestyle Gomoku*. Dans la suite, nous l'appelerons tout simplement Gomoku. La figure 1.1 présente une partie de Gomoku.

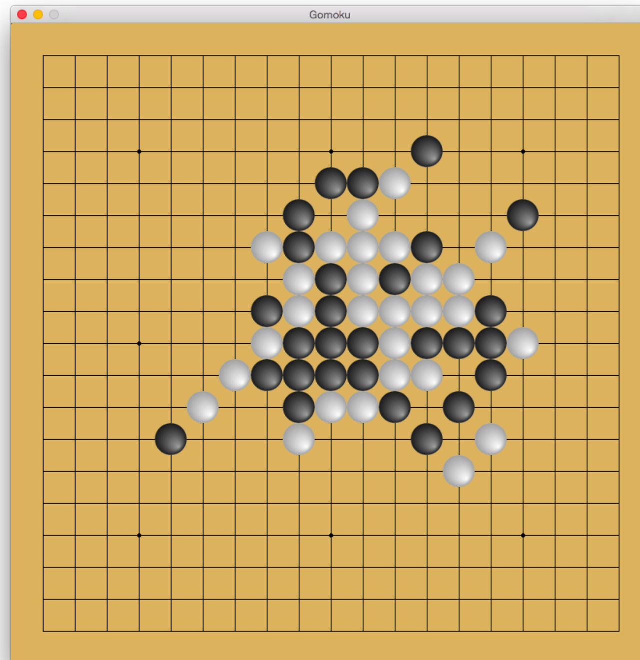

FIGURE 1.1 – Une partie de Gomoku [34]

1.1.2.2 Tic Tac Toe

Le Tic Tac Toe est un jeu de réflexion se pratiquant à deux (02) joueurs au tour par tour et dont le but est de créer le premier un alignement. Le jeu se joue généralement avec papier et crayon. Il se joue sur une grille de 3×3 cases. Les joueurs doivent remplir une case de la grille avec les symboles qui leur sont attribués : **'O'** ou **'X'**. Le gagnant est celui qui arrive à aligner trois symboles identiques, horizontalement, verticalement ou diagonalement. Le jeu est illustré par la figure 1.2.

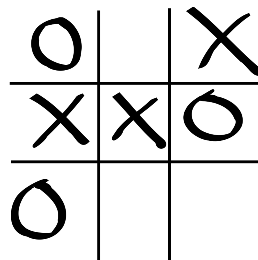

FIGURE 1.2 – Une partie de Tic Tac Toe [22]

1.1.2.3 Connect Four

Connect Four est un jeu de stratégie combinatoire dont le but est d'aligner quatre (04) pierres sur une grille comptant 6 rangées et 7 colonnes. Tour à tour, chaque joueur place une pierre dans la colonne de son choix, la pierre coulisse alors jusqu'à la position la plus basse possible dans ladite colonne à la suite de quoi c'est à l'adversaire de jouer. Le vainqueur est le joueur qui réalise le premier un alignement (horizontal, vertical ou diagonal) d'au moins quatre pierres de sa couleur. Si toutes les cases de la grille de jeu sont remplies et si aucun des deux joueurs n'a réalisé un tel alignement, la partie est alors déclarée nulle. La figure 1.3 montre une partie en cours.

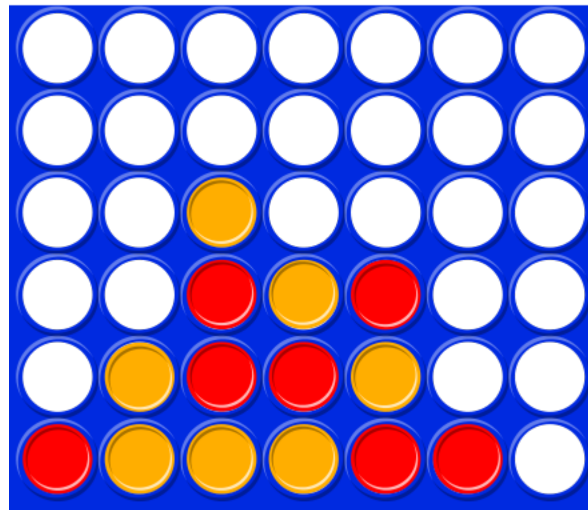

FIGURE 1.3 – Une partie de Connect Four [19]

1.1.2.4 Conventions de travail

Le type de jeux sur lequel nous travaillons exige de notre plateau qu'il soit carré. La taille du plateau est paramétrable mais elle doit être fixe durant la période d'un jeu. Ainsi, les tailles des plateaux précedemment citées dans les versions standards des jeux seront modifiées afin de correspondre à notre nomenclature. Par exemple, le plateau du jeu de Connect Four peut avoir une taille de 6×6 ou 7×7 et le nombre de pierres pour un alignement gagnant sera 4. On peut aussi avoir un plateau de Tic Tac Toe de taille 10×10 ou encore un plateau de Gomoku de taille 11×11 avec une combinaison gagnante de 5 pierres.

Chaque jeu est vu comme étant un environnement de guerre où chaque pierre posée représente un certain degré de menace. Chaque pierre peut travailler en synergie avec ses pairs de sorte que lorsqu'elles sont alignées, elles représentent toutes ensemble un degré de menace supérieure au degré de menace d'une pierre solitaire. Ce degré de menace varie en fonction de la configuration que présentent les pierres alignées. Il y a certaines configurations qui avantagent le joueur plus que d'autres. Chaque joueur voulant poser ses pierres sur le plateau essayera à

chaque fois de créer des menaces supérieures à celles de l'adversaire afin d'accroître la chance de gagner. Les joueurs pourront aussi poser des pierres qui ne représenteront aucune menace mais qui auront pour but d'atténuer voire même d'annuler les menaces de certaines pierres de l'adversaire.

1.1.3 Notion de menace

Dans les jeux de type « n-alignés », la menace est une notion très importante. Elle représente "l'*unité de mesure*" de l'état du plateau. Elle permet aux programmes informatiques de penser comme un humain dans cet immense espace de recherche. La menace comme décrite plus haut est une configuration de pierres alignées d'une certaine façon pouvant assurer à son joueur une certaine tendance de gain qui peut être avantageuse ou non car elles se compensent toujours avec les menaces adverses pour l'évaluation de la position. Dans un premier temps, pour rendre les notions plus compréhensibles, nous allons prendre un jeu de type « n-alignés » dont le but est d'aligner 5 pierres. Nous donnons les différentes menaces possibles dans un environnement où la combinaison de cinq (05) pierres alignées est gagnante : les *quatre*, les *trois*, les *deux* et les *uns* . On appelle *quatre* un alignement de quatre pierres d'un même joueur que ce soit horizontalement, verticalement ou diagonalement. Il en existe plusieurs types classés en trois catégories. Voici entre autres, une liste des six types de configuration qui donnent des quatre :

- **Type 1** : quatre pierres alignées consécutivement dont les emplacements de part et d'autre des extrémités sont libres.
- **Type 2** : quatre pierres alignées consécutivement dont l'emplacement au bout d'une extrémité est libre tandis que le second est occupé.
- **Type 3** : quatre pierres alignées consécutivement dont les emplacements de part et d'autre des extrémités sont occupés.
- **Type 4** :quatre pierres alignées avec un saut d'emplacement et dont les emplacements de part et d'autre des extrémités sont libres.
- **Type 5** : quatre pierres alignées avec un saut d'emplacement et dont l'emplacement au bout d'une extrémité est libre tandis que le second est occupé.
- **Type 6** : quatre pierres alignées avec un saut d'emplacement et dont les emplacements de part et d'autre des extrémités sont occupés.

Les trois catégories en lesquelles on peut les regrouper sont les *quatre ouverts* contenant le type 1, les *quatre mi-ouverts* contenant les types 2 ,4, 5, 6 et les *quatre fermés* contenant le type 3. La figure 1.4 présente les menaces de type 4.

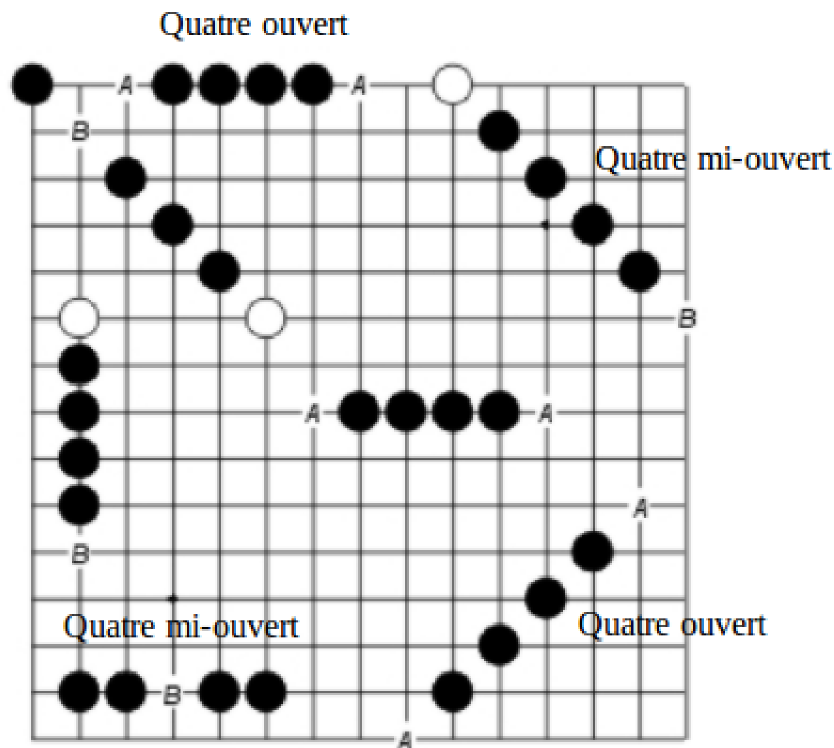

FIGURE 1.4 – Menaces de type *quatre* [26]

La catégorie des quatre fermés n'apparaît pas sur la figure parce qu'un quatre fermé n'est pas une menace mais plutôt une stratégie de défense qui annule la menace de l'adversaire lorsque aucun coup du joueur ne le met en avance sur lui ou que le fait de ne pas le bloquer lui assure la victoire au prochain coup.

Quand un joueur réussit à avoir un quatre ouvert, plus aucun coup de l'adversaire ne peut le contrer puisque les deux extrémités étant libres, quoi qu'en soit le coup de l'adversaire, il y aura au moins une extrémité libre pour réussir à aligner cinq pierres et remporter la partie. L'un des principaux objectifs du joueur sera alors d'avoir un quatre ouvert et ces derniers ne proviennent que des trois ouverts.

Il en est de même pour les *trois*, les *deux* et les *uns*. Même si le joueur humain dans ce cas de jeu, s'intéressera aux menaces de types trois et quatre, l'ordinateur ne peut pas avoir cette intuition. Les quatre ouverts provenant toujours des trois ouverts, il serait impérieux de bloquer automatiquement les trois ouverts car ils deviendront des quatre ouverts au prochain coup et deviendront imparables.

Dans le cas d'un jeu où la taille de la combinaison gagnante serait de n, les menaces ouvertes et mi-ouvertes de taille $n - 1$ à 1 et aussi la menace de taille n (dans ce cas, les notions d'ouverture et de mi-ouverture n'existent plus) sont déterminées pour chaque état du plateau. Pour cela, une catégorisation des menaces a été faite parce que certaines sont plus importantes que

d'autres et des coefficients leur sont affectés en fonction de leur importance. C'est le cas des menaces ouvertes qui sont plus importantes que les fermées. Aussi, les coefficients différents des menaces permettent à l'ordinateur de préférer les coups donnant des menaces plus importantes que ceux donnant de moins importantes. C'est le cas d'un quatre ouvert qui est nettement plus prometteur qu'un deux ouvert. L'algorithme 1 permet de déterminer si la menace détectée est ouverte, mi-ouverte ou fermée.

Algorithme 1 : Algorithme de catégorisation d'une menace [1]

Entrées : a est un entier positif représentant la taille de la menace
n est un entier positif représentant la taille de la combinaison gagnante

Output : 0 si la menace est mi-ouverte
1 si la menace est ouverte
-1 si la menace est fermée

```
si a = n − 1 ET MENACEATROU() = Vrai alors
    retourner 0
sinon
    si NBALIGNPOS() = n alors
        retourner 0
    sinon
        si NBALIGNPOS() < n alors
            retourner (-1)
        sinon
            retourner 1
        fin
    fin
fin
```

La routine MENACEATROU() détermine si oui ou non la menace qu'on essaie de catégoriser contient un saut d'emplacement.

La routine NBALIGNPOS() détermine grâce aux emplacements vides se situant de part et d'autres de la menace, quel est le nombre maximum de pierres qu'on peut aligner dans le sens de la menace.

Des algorithmes de parcours du plateau ont été écrits [1] afin de détecter les éventuelles menaces tant sur les lignes que sur les colonnes et sur les deux diagonales (pas uniquement les centrales).

1.1.4 Heuristique générique

L'heuristique générique dont il est question est le résultat des travaux de Gael AGLIN (Master thesis, IFRI 2016) [1]. Certaines règles ont été définies pour la détermination de la formule.

Pour un jeu où le nombre d'alignement pour gagner est n, la grande menace qu'il faut créer (respectivement éviter si elle vient de l'adversaire) et qui conduit toujours à une victoire (respectivement à un échec) est la menace de taille $n-1$ de type ouvert et la seule combinaison qui peut déboucher dans cette configuration est la menace de taille $n-2$ de type ouvert. En revanche, la menace mi-ouverte de taille $n-2$ n'est pas intéressante en soi. Aussi, quoique la menace de taille $n-1$ de type mi-ouvert soit très proche de la victoire, elle est moins importante qu'une de taille $n-2$ de type ouvert car dans ce dernier cas, on peut déboucher sur la menace de taille $n-1$ de type mi- ouvert comme on peut déboucher également sur la menace ouverte de taille $n-1$, ce qui est très intéressant. En général, le jeu devient décisif lorsque l'on a des menaces de taille $n-2$ et $n-1$. Les menaces de taille inférieure à $n-2$ ne sont pas intéressantes. Contrairement à la plupart des jeux à somme nulle, les menaces ne sont pas coefficiées de la même pondération qu'elles soient adverses ou non. Une grande pondération est donnée aux menaces adverses qu'aux menaces du joueur pour éviter que l'adversaire prenne un avantage irréversible. En revanche, les menaces qualifiées de non intéressantes (de taille inférieure à $n-2$ et la menace mi-ouverte de taille $n-2$) sont coefficiées de la même manière indépendamment des joueurs. Voici la formule de la fonction d'évaluation [1] proposée :

$$A = \begin{cases} \sum_{i=1}^{n-3} (a_{2i-1}p_{i,1} + a_{2i}p_{i,2}) + a_{2(n-2)-1}p_{n-2,1} + 100p_{n-2,2} + 80p_{n-1,1} + 250p_{n-1,2} + 1000000p_n \text{ si } n > 3 \\ a_1p_{n-2,1} + 100p_{n-2,2} + 80p_{n-1,1} + 250p_{n-1,2} + 1000000p_n \text{ si } n = 3 \end{cases}$$

$$B = \begin{cases} \sum_{i=1}^{n-3} (a_{2i-1}q_{i,1} + a_{2i}q_{i,2}) + a_{2(n-2)-1}q_{n-2,1} + 1300q_{n-2,2} + 2000q_{n-1,1} + 5020q_{n-1,2} + 1000000q_n \text{ si } n > 3 \\ a_1q_{n-2,1} + 1300q_{n-2,2} + 2000q_{n-1,1} + 5020q_{n-1,2} + 1000000q_n \text{ si } n = 3 \end{cases}$$

$$f = A - B$$

A — évaluation des menaces du joueur sur le plateau
B — évaluation des menaces de l'adversaire sur le plateau
a_i — coefficient de la menace moins importantes d'indice i
$p_{i,1}$ — nombre de menaces mi-ouvertes de taille i du joueur
$p_{i,2}$ — nombre de menaces ouvertes de taille i du joueur
p_i — nombre de menaces sans trou de taille i du joueur
$q_{i,1}$ — nombre de menaces mi-ouvertes de taille i de l'adversaire
$q_{i,2}$ — nombre de menaces ouvertes de taille i de l'adversaire
q_i — nombre de menaces sans trou de taille i de l'adversaire
n — nombre d'alignement conduisant à la victoire

Les coefficients $100, 80, 250, 1000000$ et $1300, 2000, 5020, 1000000$ sont choisis par logique (priorité de certaines menaces sur d'autres) et par expérimentation.

Etant donnée la priorité de certaines menaces sur d'autres, la suite des coefficients des menaces moins importantes (partant de 1 à $n-3$) est déterminé grâce à l'algorithme 2

Algorithme 2 : Calcul des valeurs du tableau de coefficients des menaces moins importantes [1]

Entrées : n est un entier positif représentant la taille de la combinaison gagnante
pas est un réel positif
i est un entier positif
tmp est un réel positif
a est un tableau de $2(n-3)+1$ réels positifs

Output : a tableau de coefficients

```
pas ← 10/longueur(a)
tmp ← pas
i ← 1
a[i] ← tmp
si n = 3 alors
  | retourner a
sinon
  | pour i ← 1 à longueur(a) faire
  |   | tmp ← tmp + pas
  |   | a[i + 2] ← tmp
  |   | tmp ← tmp + pas
  |   | a[i + 1] ← tmp
  |   | i ← i + 1
  | fin
  | retourner a
fin
```

1.2 Généralités sur les algorithmes minimax et alpha-bêta

1.2.1 L'algorithme minimax

L'algorithme minimax [5, 10] est un algorithme qui s'applique à des jeux à deux joueurs, à somme nulle. Cet algorithme amène l'ordinateur à passer en revue toutes les possibilités pour un nombre limité de coups et à leur assigner une valeur qui prend en compte les bénéfices pour le joueur et pour son opposant. Le meilleur choix est alors celui qui maximise les gains du

joueur tout en supposant que l'opposant cherche au contraire à les minimiser.

1.2.1.1 Principe de base

La méthode générale est de partir d'une position et de générer l'arbre de jeux qui représente tous les coups possibles jusqu'à une certaine profondeur :

- un nœud est une situation légale du jeu ;
- les fils d'un nœud sont les situations que l'on peut atteindre à partir de ce nœud en respectant les règles du jeu ;
- un nœud **MAX** choisit un coup de score maximal parmi ses fils ;
- un nœud **MIN** choisit un coup de score minimal parmi ses fils ;
- par convention MAX commence la partie ;
- les feuilles correspondent aux fins de partie et chaque feuille est évaluée avec une fonction d'évaluation.

On appelle Minimax à la fois l'algorithme et la valeur obtenue par exécution de cet algorithme. On visite l'arbre de jeu pour faire remonter à la racine la valeur Minimax. La valeur est calculée récursivement comme suit :

- Minimax(e) = h(e), si e est une feuille de l'arbre et h la fonction d'évaluation
- Minimax(e) = max(Minimax(e_1),. . ., Minimax(e_n)), si e est un nœud Joueur avec les fils $f(e) = e_1, \ldots, e_n$
- Minimax(e) = min(Minimax(e_1),. . ., Minimax(e_n)), si e est un nœud Adversaire avec les fils $f(e) = e_1, \ldots, e_n$

Il est rarement possible de développer l'ensemble de l'arbre de jeu. Il est donc nécessaire de créer l'arbre de jeu à une profondeur fixée et de disposer d'une fonction d'évaluation pour évaluer les feuilles (qui ne représentent plus une fin de partie), la valeur obtenue sera d'autant plus grande que la situation sera favorable. La figure 1.5 présente l'exécution de l'algorithme minimax sur un arbre.

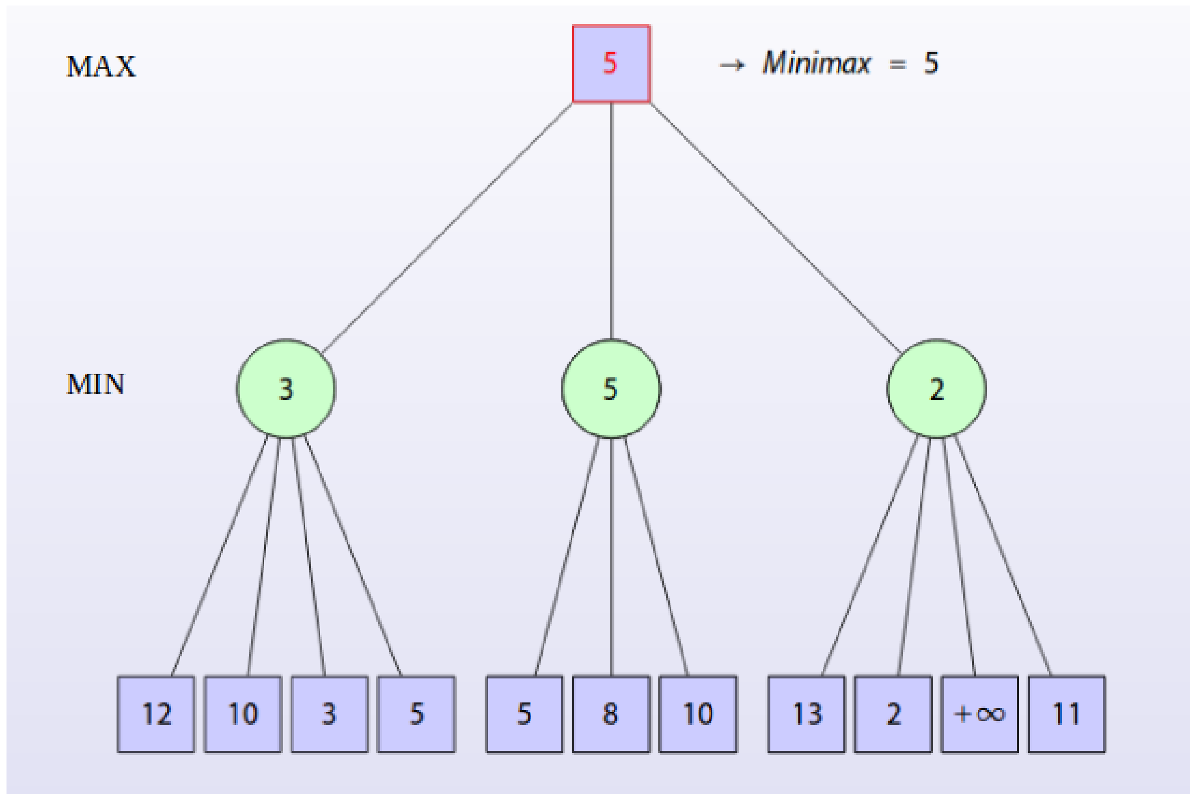

FIGURE 1.5 – Exécution de l'algorithme *minimax* [2]

L'algorithme minimax effectue une exploration complète de l'arbre de recherche alors qu'une exploration partielle de l'arbre est généralement suffisante : lors de l'exploration, il n'est pas nécessaire d'examiner les sous-arbres qui conduisent à des configurations dont la valeur ne contribuera pas au calcul du gain à la racine de l'arbre. L'élagage *alpha-bêta* nous permet de réaliser cette exploration partielle.

1.2.1.2 Pseudo-code

L'algorithme 3 est un pseudo-code de l'algorithme minimax.

Algorithme 3 : Algorithme minimax [2]

```
Fonction MINIMAX(e, d)
  Entrées : nœud e, profondeur d

  Output : Valeur Minimax du nœud e

  si d = 0 (e est un nœud terminal) alors
    retourner h(e) // valeur de l'heuristique

  sinon
    si e nœud joueur alors
      pour tout enfant (fils) e_i de e faire
        retourner max(minimax(e_i, d − 1))
      fin
    sinon
      pour tout enfant (fils) e_i de e faire
        retourner min(minimax(e_i, d − 1))
      fin
    fin
  fin
```

1.2.2 L'algorithme alpha-bêta

L'élagage alpha-bêta [6, 10] (abrégé élagage α-β) est une technique permettant de réduire le nombre de nœuds évalués par l'algorithme minimax. Il évite d'évaluer des nœuds dont on est sûr que leur qualité sera inférieure à un nœud déjà évalué, il permet donc d'optimiser grandement l'algorithme minimax sans en modifier le résultat.

1.2.2.1 Principe de base

Pour réaliser l'élagage alpha-bêta, on va associer à :

- chaque nœud e de type MAX une valeur auxiliaire appelée *Alpha(e)* égale à la valeur de $h(e)$ si e est un nœud terminal ou à la meilleure valeur (valeur maximale) de ses fils trouvée jusqu'ici ;

- chaque nœud e de type MIN une valeur *Beta(e)*, valeur égale à $h(e)$ si e est un nœud terminal ou à la valeur minimum de ses successeurs trouvée jusqu'ici.

La figure 1.6 montre un exemple de coupure alpha et de coupure bêta.

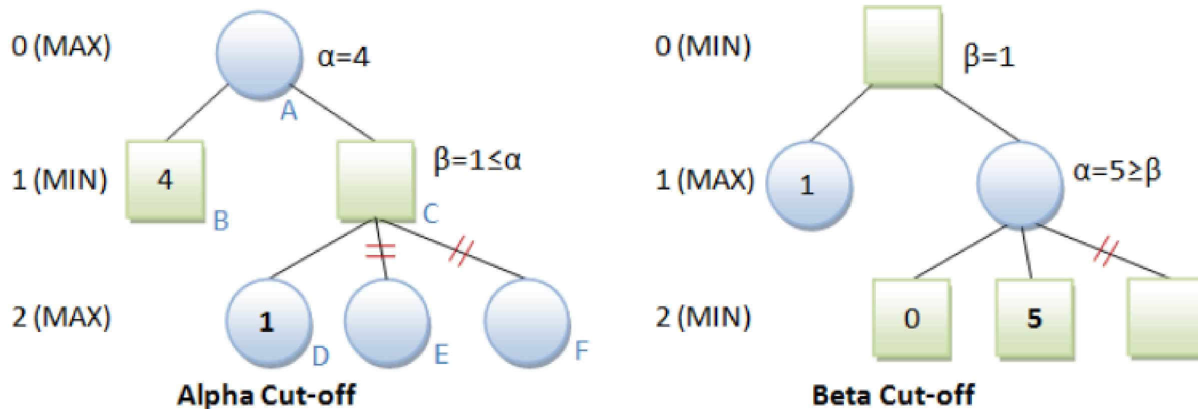

FIGURE 1.6 – Elagage *alpha-bêta* [25]

Coupure Alpha : le premier enfant du nœud Min C vaut 1 donc C vaudra au plus 1. Le nœud Max A prendra donc la valeur 4 (maximum entre 4 et une valeur inférieure ou égale à 1).

Coupure Bêta : le second enfant du nœud Max C vaut 5 donc C vaudra au minimum 5. Le nœud Min A prendra donc la valeur 1 (minimum entre 1 et une valeur supérieure ou égale à 5).

Dans les cas généraux ci-dessus, on peut voir que l'élagage α-β peut nous aider à couper des sous branches inutiles pour diminuer l'espace de recherche.

1.2.2.2 Pseudo-code

L'algorithme 4 donne un exemple de pseudo-code pour réaliser l'élagage Alpha-bêta (α-β).

Algorithme 4 : Algorithme alpha-bêta [2]

```
Fonction alphabeta(e, alpha, beta, d)
    // alpha toujours inférieur à bêta

    Entrées : nœud e, alpha= −∞, beta = +∞, profondeur d

    Output : Valeur correspondante au meilleur coup à partir de e

    si d = 0 (e est un nœud terminal) alors
        retourner h(e) // valeur de l'heuristique

    sinon
        si e nœud joueur alors
            val ← −∞
            pour tout enfant (fils) e_i de e faire
                val ← max(val, alphabeta(e_i, alpha, beta, d − 1))
                si val ≥ beta alors
                    COUPURE BETA
                    retourner val
                fin
                alpha ← max(val, alpha)
            fin
        sinon
            val ← +∞
            pour tout enfant (fils) e_i de e faire
                val ← min(val, alphabeta(e_i, alpha, beta, d − 1))
                si alpha ≥ val alors
                    COUPURE ALPHA
                    retourner val
                fin
                beta ← max(val, beta)
            fin
        fin
        retourner val
    fin
```

1.2.2.3 Optimisations possibles

Des améliorations supplémentaires peuvent être obtenues sans sacrifier la précision en recherchant des parties de l'arbre qui sont susceptibles de forcer tôt les coupes alpha-bêta. Cela s'obtient grâce à des heuristiques. Une heuristique commune et très peu coûteuse est celle où le dernier mouvement qui a provoqué une coupure au même niveau dans la recherche d'arbre est toujours examiné en premier. Cette idée peut également être généralisée dans un ensemble de tableaux de réfutation [4].

D'autres améliorations permettent de rétrécir l'intervalle de départ entre les valeurs alpha et bêta utilisées dans le nœud racine. Plutôt que d'appeler alpha-Bêta avec des valeurs initiales de *-INFINI* pour alpha et *+INFINI* pour bêta, on peut lui permettre de couper plus de branches, si on augmente (respectivement diminue) la valeur initiale de alpha (respectivement bêta). Si la valeur finale trouvée est comprise entre les alpha et bêta initiaux, le résultat sera tout de même juste, bien que l'on ait coupé plus de branches inutiles qu'avec des valeurs infinies. La recherche aspirante [4] permet de régler les valeurs initiales pour alpha et bêta en prenant en compte les résultats de la recherche précédente. Au début de chaque itération, les valeurs maximales (respectivement minimales) sont initialisées avec les valeurs remontées de l'itération précédente, additionnées (respectivement diminuées) de la valeur d'un pion.

1.3 Apprentissage automatique

1.3.1 Types d'apprentissage automatique

L'*apprentissage automatique* concerne la conception, l'analyse, le développement et l'implémentation de méthodes permettant à une machine (au sens large) d'évoluer par un processus systématique, et ainsi de remplir des tâches difficiles ou problématiques par des moyens algorithmiques [27]. L'analyse peut concerner des graphes, arbres, ou courbes au même titre que de simples nombres. L'objectif est d'extraire et exploiter automatiquement l'information présente dans un jeu de données.

Les algorithmes utilisés permettent, dans une certaine mesure, à un système piloté par ordinateur ou assisté par ordinateur, d'adapter ses analyses et ses comportements en réponse, en se fondant sur l'analyse de données empiriques provenant d'une base de données ou de capteurs. La difficulté réside dans le fait que l'ensemble de tous les comportements possibles compte tenu de toutes les entrées possibles devient rapidement trop complexe à décrire (on parle d'explosion combinatoire). On confie donc à des programmes le soin d'ajuster un modèle pour simplifier cette complexité et de l'utiliser de manière opérationnelle.

Les algorithmes d'apprentissage peuvent se catégoriser selon le mode d'apprentissage qu'ils emploient.

Apprentissage supervisé [28] : Les classes sont prédéterminées et les exemples connus ; le système apprend à classer selon un modèle de classement. Un expert doit préalablement étiqueter des exemples. Le processus se passe en deux phases. Lors de la première phase (hors ligne, dite d'apprentissage), il s'agit de déterminer un modèle des données étiquetées. La seconde phase (en ligne, dite de test) consiste à prédire l'étiquette d'une nouvelle donnée, connaissant le modèle préalablement appris. Parfois il est préférable d'associer une donnée non pas à une classe unique, mais une probabilité d'appartenance à chacune des classes prédéterminées (on parle alors d'apprentissage supervisé probabiliste). Les machines à vecteurs de support ou séparateurs à vaste marge (SVM) [1] en sont des exemples typiques.

Apprentissage non supervisé ou clustering [28] : Le système ou l'opérateur ne dispose que d'exemples, mais non d'étiquettes et le nombre de classes et leur nature n'ont pas été prédéterminés. Aucun expert n'est requis. L'algorithme doit découvrir par lui-même la structure plus ou moins cachée des données. Le partitionnement des données, data clustering en anglais, est un algorithme d'apprentissage non supervisé. Le système doit ici, dans l'espace de description, cibler les données selon leurs attributs disponibles, pour les classer en groupe homogènes d'exemples. La similarité est généralement calculée selon une fonction de distance entre paires d'exemples. C'est ensuite à l'opérateur d'associer ou déduire du sens pour chaque groupe et pour les motifs d'apparition de groupes, ou de groupes de groupes, dans leur « espace ». Divers outils mathématiques et logiciels peuvent l'aider. On parle aussi d'analyse des données en régression (ajustement d'un modèle par une procédure de type moindres carrés ou autre optimisation d'une fonction de coût). Si l'approche est probabiliste (c'est-à-dire que chaque exemple, au lieu d'être classé dans une seule classe, est caractérisé par un jeu de probabilités d'appartenance à chacune des classes), on parle alors de « soft clustering » (par opposition au « hard clustering »).

Apprentissage par renforcement [28] : L'algorithme apprend un comportement étant donnée une observation. L'action de l'algorithme sur l'environnement produit une valeur de retour qui guide l'algorithme d'apprentissage. le but est d'apprendre, à partir d'expériences, ce qu'il convient de faire en différentes situations, de façon à optimiser une récompense quantitative au cours du temps. L'agent cherche, au travers d'expériences itérées, un comportement décisionnel[2] optimal, en ce sens qu'il maximise la somme des récompenses au cours du temps. Il est très utile dans le cadre de problèmes où :

- des stratégies comportementales efficaces sont inconnues a priori ou sont difficilement automatisables

[1] ensemble de techniques d'apprentissage supervisé destinées à résoudre des problèmes de discrimination et de régression.

[2] stratégie ou politique qui est une fonction associant à l'état courant l'action à exécuter.

- il y a de l'incertain dans la manière dont l'environnement évolue.

Dans le cadre de notre travail, nous avons choisi l'*apprentissage par renforcement* pour la détermination des paramètres de l'heuristique dans la recherche du meilleur coup (ce qu'il convient de faire) à jouer en différentes situations. Il convient donc de parcourir plus en profondeur cette notion.

1.3.2 Approfondissement de la notion d'apprentissage par renforcement

Le problème d'apprentissage de renforcement est censé être un encadrement direct du problème de l'apprentissage par interaction pour atteindre un objectif. L'apprenant ou le décideur s'appelle l'agent. Ce qu'il interagit avec s'appelle l'environnement. Ceux-ci interagissent continuellement, l'agent sélectionnant les actions et l'environnement répondant à celles-ci et présentant de nouvelles situations à l'agent. L'environnement donne lieu à des récompenses, des valeurs numériques spéciales que l'agent essaie de maximiser. Une spécification complète d'un environnement définit une tâche, une instance du problème d'apprentissage de renforcement.

Formellement, la base du modèle d'apprentissage par renforcement consiste en :

- un ensemble d'états S de l'agent dans l'environnement ;
- un ensemble d'actions A que l'agent peut effectuer ;
- un ensemble de valeurs scalaires "récompenses" R que l'agent peut obtenir.

À chaque pas de temps t de l'algorithme, l'agent perçoit son état $s_t \in S$ et l'ensemble des actions possibles $A(s_t)$. Il choisit une action $a \in A(s_t)$ et reçoit de l'environnement un nouvel état s_{t+1} et une récompense r_{t+1} (qui est nulle la plupart du temps et vaut classiquement 1 dans certains états clefs de l'environnement). Fondé sur ces interactions, l'algorithme d'apprentissage par renforcement doit permettre à l'agent de développer une politique $\Pi : S \rightarrow A$ qui lui permette de maximiser la quantité de récompenses. Ainsi, la méthode de l'apprentissage par renforcement est particulièrement adaptée aux problèmes nécessitant un compromis entre la quête de récompenses à court terme et celle de récompenses à long terme. La figure 1.7 résume le modèle.

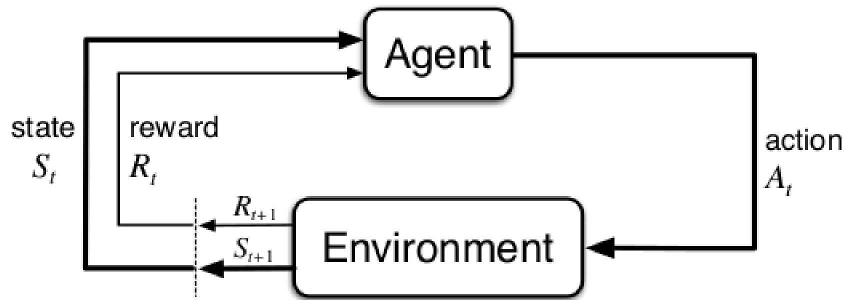

FIGURE 1.7 – Interaction *agent-environnement* [7]

Si l'on devait identifier une idée centrale et nouvelle pour renforcer l'apprentissage, il s'agirait certainement de l'apprentissage par Temporal Difference (TD). L'apprentissage par TD [7] est une méthode d'apprentissage par machine basée sur la prédiction. TD apprend en échantillonnant l'environnement selon une certaine politique, et est liée à des techniques de programmation dynamique car elle se rapproche de son estimation actuelle basée sur des estimations précédemment acquises : on parle de *bootstrapping* [3]. L'idée fondamentale de l'apprentissage par TD est que l'on ajuste les prédictions pour correspondre à d'autres prévisions plus précises sur l'avenir.

Les méthodes TD utilisent l'expérience pour résoudre le problème de prédiction. Étant donné une certaine expérience suite à une politique Π, elles mettent à jour leur estimation v de v_{Π} [4] pour les états non terminaux s_t se produisant dans cette expérience. Une politique est une règle selon laquelle l'agent suit le choix des actions, compte tenu de l'état dans lequel il se trouve. À l'instant $t+1$, ils forment immédiatement une cible et font une mise à jour utile en utilisant la récompense observée R_{t+1} et l'estimation $V(S_{t+1})$. L'équation (1.1) donne la formule de mise à jour.

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{1.1}$$

[3]référence aux méthodes Différence Temporelle (TD) dont la mise à jour se base en partie sur une estimation existante

[4]valeur obtenue en suivant la politique Π

L'algorithme 5 donne la forme générale de l'apprentissage par TD.

Algorithme 5 : Algorithme TD(0) [7]

Entrées : la politique Π à évaluer

Initialiser $V(s)$ arbitrairement (par exemple, $V(s) = 0, \forall s \in S$)
répéter
  $Initialiser S$
  répéter pour chaque étape de l'épisode
    Choisir A `// action donnée par` Π `pour` S

    Prendre l'action A
    Observer la récompense R et l'état suivant S'
    $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
    $S \leftarrow S'$
  jusqu'à *S terminal*
jusqu'à *la fin de l'épisode*

Nous passons maintenant à l'utilisation des méthodes de prédiction TD pour le problème de contrôle. Ici, survient le dilemme de l'exploration et de l'exploitation. Le choix de l'action doit garantir un équilibre entre l'exploration et l'exploitation de l'apprentissage déjà réalisé : faire confiance à l'estimation courante pour choisir la meilleure action à effectuer dans l'état courant (exploitation) ou, au contraire, choisir une action a priori sous-optimale pour observer ses conséquences (exploration). Naturellement, on conçoit intuitivement que l'exploration doit initialement être importante (quand l'apprentissage est encore très partiel, on explore) puis diminuer au profit de l'exploitation quand l'apprentissage a été effectué pendant une période suffisamment longue. Les nouvelles approches entrent dans deux classes principales : « *on-policy* » (Sarsa) et « *off-policy* » (Q-learning).

1.3.2.1 Sarsa : on-policy TD control

Sarsa [7, 8, 12] une méthode de contrôle TD basée sur la politique. La première étape consiste à apprendre une fonction action-valeur plutôt qu'une fonction état-valeur. En particulier, pour une méthode basé sur la politique, il faut estimer $q_{\Pi}(s, a)$ pour la politique de comportement actuelle Π et pour tous les états s et actions a. Rappelons qu'un épisode consiste en une séquence alternée d'états et paires état-action. La figure 1.8 présente les transitions d'un état à un autre.

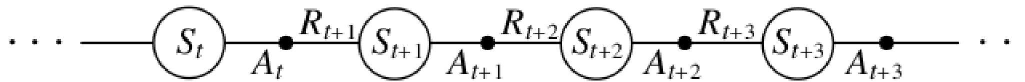

FIGURE 1.8 – Passage d'un état à un autre [7]

Nous considérons les transitions de la paire état-action à la paire état-action et apprenons la valeur des paires état-action grâce à la formule (1.2)

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{1.2}$$

Cette mise à jour se fait après chaque transition d'un état non terminal S_t. Si S_{t+1} est terminal, alors $Q(S_{t+1}, A_{t+1})$ est défini comme étant nul. Cette règle utilise chaque élément du quintuple des événements, $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, qui constituent une transition d'une paire état-action à la suivante. Ce quintuple donne naissance au nom *SARSA (State-Action-Reward-State-Action)*[12] pour l'algorithme. Comme dans toutes les méthodes en ligne, q_Π est continuellement estimé pour la politique de comportement Π et, en même temps, Π est changé en faveur de la gourmandise en respectant q_Π. La forme générale de l'algorithme de contrôle de Sarsa est donnée par l'algorithme 6.

Algorithme 6 : Algorithme Sarsa [7]

Entrées : la politique Π à évaluer

Initialiser $Q(s, a), \forall s \in S, a \in A(s)$, arbitrairement, et $Q(etat - terminal, .) = 0$

répéter
- $InitialiserS$
- Choisir A de S en utilisant une politique dérivée de Q
- **répéter** pour chaque étape de l'épisode
  - Prendre l'action A
  - Observer la récompense R et l'état suivant S'
  - $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
  - $S \leftarrow S'$
  - $A \leftarrow A'$
- **jusqu'à** *S terminal*

jusqu'à *la fin de l'épisode*

1.3.2.2 Q-Learning : Off-Policy TD Control

L'une des avancées les plus importantes dans le renforcement de l'apprentissage a été le développement d'un algorithme de contrôle TD hors politique appelé *Q-learning* [7, 8, 12]. Q-

learning est utilisé pour trouver une politique de sélection d'action optimale. Cela fonctionne en apprenant une fonction action-valeur qui donne finalement l'utilité attendue de prendre une action donnée dans un état donné et de suivre la politique optimale par la suite. Lorsqu'une telle fonction de valeur d'action est apprise, la politique optimale peut être construite en sélectionnant simplement l'action avec la valeur la plus élevée dans chaque état. L'une des forces de Q-learning est qu'il est capable de comparer l'utilité attendue des actions disponibles sans nécessiter un modèle de l'environnement. Q-learning trouve éventuellement une politique optimale, en ce sens que la valeur attendue de la récompense totale retourne sur toutes les étapes successives, à partir de l'état actuel, est le maximum réalisable.

L'algorithme dispose donc d'une formule de mise à jour (1.3) qui calcule la quantité d'une combinaison état-action :

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{1.3}$$

Avant que l'apprentissage ne commence, Q renvoie une valeur fixe (arbitraire) choisie par le concepteur. Ensuite, chaque fois que l'agent sélectionne une action et observe une récompense et un nouvel état qui peut dépendre à la fois de l'état précédent et de l'action sélectionnée, Q est mis à jour. Le noyau de l'algorithme est une mise à jour d'itération de valeur simple. Il assume l'ancienne valeur et fait une correction basée sur les nouvelles informations. L'algorithme 7 donne le pseudo-code de l'algorithme Q-learning.

Algorithme 7 : Algorithme Q-learning [7]

Entrées : la politique Π à évaluer

Initialiser $Q(s, a), \forall s \in S, a \in A(s)$, arbitrairement, et $Q(etat - terminal, .) = 0$
répéter
  $Initialiser S$
  répéter pour chaque étape de l'épisode
    Choisir A de S en utilisant une politique dérivée de Q
    Prendre l'action A
    Observer la récompense R et l'état suivant S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma max Q(S', a) - Q(S, A)]$
    $S \leftarrow S'$
  jusqu'à *S terminal*
jusqu'à *la fin de l'épisode*

1.3.2.3 Paramètres des formules de mise à jour

Les formules de mise à jour du TD(0) (1.1), de Sarsa (1.2) et du Q-learning (1.3) utilisent toutes les mêmes paramètres α et γ. Ces paramètres jouent un rôle important dans l'efficacité des méthodes.

Taux d'apprentissage (alpha α) [29] : Le taux d'apprentissage détermine dans quelle mesure les informations nouvellement acquises annuleront les anciennes informations. Un facteur de 0 fera en sorte que l'agent n'apprenne rien, alors qu'un facteur de 1 permettrait à l'agent d'examiner uniquement les informations les plus récentes. Dans des environnements entièrement déterministes, un taux d'apprentissage de 1 est optimal.

Facteur de réduction (gamma γ) [29] : Le facteur de réduction détermine l'importance des récompenses futures. Un facteur de 0 rendra l'agent «opportuniste» en considérant seulement les récompenses courantes, alors qu'un facteur approchant 1 fera en sorte qu'il s'efforce d'obtenir une récompense élevée à long terme.

Conclusion

Dans ce chapitre, nous avons abordé les jeux de type « n-alignés » et l'heuristique générique. Nous avons aussi présenté les types d'apprentissage automatique et indiqué celui qui convient le mieux pour notre travail pour ensuite étudier des méthodes qui s'y rapportent. Nous exposons dans le chapitre suivant le matériel et les méthodes utilisés pour aborder le problème.

Chapitre 2

Matériel et méthodes

Résumé. Le système est modélisé de façon à s'adapter à différentes tailles de plateau et à faciliter l'ajout d'un nouveau jeu. Il suffit de créer la classe implémentant le plateau de jeu, redéfinir la méthode déterminant les différents coups possibles dans un état donné du plateau et la méthode vérifiant si leu jeu est résolu.

Introduction

Dans ce chapitre, nous faisons la description du matériel utilisé lors de la phase de développement. Nous détaillons la manière dont nous avons préparé notre environnement dans le but de la création de notre agent intelligent. Plus précisément, nous exposons le modèle de développement qui a été utilisé (le cycle en V) et son implémentation.

2.1 Matériel

Pour notre implémentation et nos tests, nous avons travaillé sur un ordinateur présentant les caractéristiques ci-après :

- Système d'exploitation : Ubuntu 16.04 LTS ;
- Type de système d'exploitation : 64 bits ;
- Processeur : AMD E2-1800 APU with Radeon(tm) HD Graphics ;
- Mémoire : 5,4 Go.

Aussi, avons-nous opté pour le langage de programmation JAVA. En effet, c'est un langage orienté objet et qui a un caractère multi-plateforme. Nous avons utilisé JAVA SE avec la version 8 du JDK.

2.2 Le cycle en V

Le modèle du cycle en V [30, 31] est un modèle conceptuel de gestion de projet composé de 9 étapes et découpé en 3 parties :

- sur la partie gauche du V, *la phase de conception* ;
- sur la pointe du V, *la réalisation* ;
- sur la partie droite du V, *la phase de tests*.

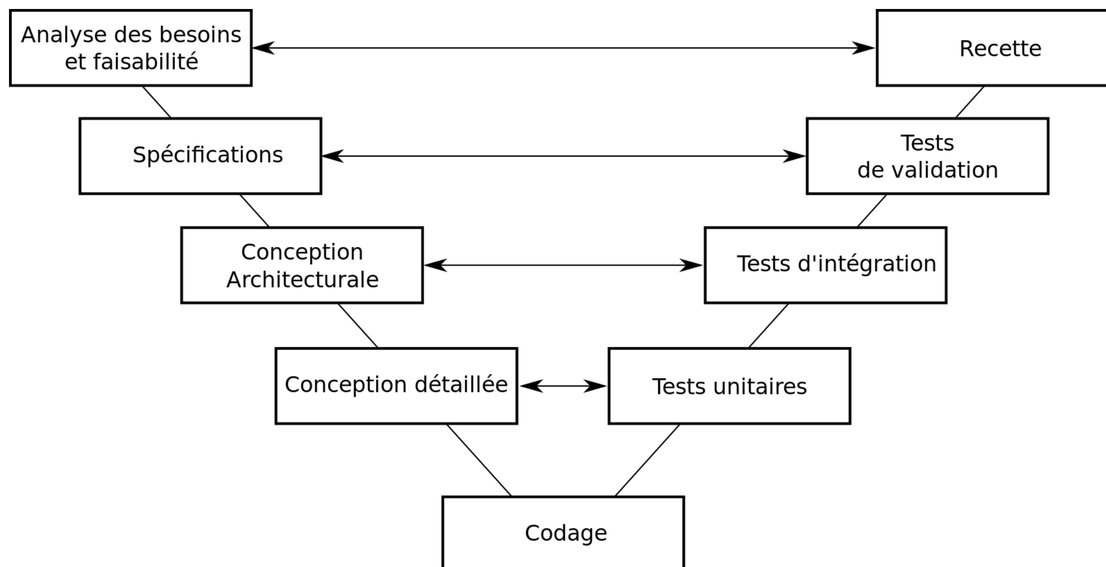

FIGURE 2.1 – Cycle de développement en V [32]

Chaque étape d'une branche a son pendant dans l'autre branche, c'est à dire qu'une étape de conception correspond à une étape de test qui lui est spécifique. A tel point, d'ailleurs, qu'une étape de test peut être élaborée dès que la phase de conception correspondante est terminée, indépendamment du reste du projet. Les phases de la partie montante doivent renvoyer de l'information sur les phases en vis-à-vis lorsque des défauts sont détectés afin d'améliorer le logiciel. Une fois l'ensemble des besoins capturés et les spécifications établies, il arrive que dès le niveau de l'architecture, voire en phase de conception détaillée ou de codage, des difficultés d'ordre de cohérence, technique et humain interviennent. C'est la différence entre la théorie et la pratique.

En pratique, il est difficile voire impossible de totalement détacher la phase de conception d'un projet de sa phase de réalisation. C'est souvent au cours de l'implémentation qu'on se rend compte que les spécifications initiales étaient incomplètes, fausses ou irréalisables sans compter les ajouts de nouvelles fonctionnalités par les clients. C'est principalement pour cette raison que le cycle en V n'est pas toujours adapté à un développement logiciel. D'autres modèles tels que les *méthodes agiles* [33] permettent plus facilement des modifications (parfois radicales) de la conception initiale à la suite d'une première implémentation ou série d'implémentations. Par contre, ces méthodes manquent parfois de traçabilité, ce qui nécessite d'impliquer le client à la fois en termes de conception et également en termes juridiques alors que ce n'est pas le cas du cycle en V. Avec un cycle en V, le client est censé recevoir ce qu'il a commandé, alors qu'avec les méthodes agiles le client devient co-développeur et intervient donc au niveau du projet. De plus, il est facile de prévoir les tests à réaliser au moment où l'on conçoit une fonctionnalité, le travail s'enchaîne donc de façon assez naturelle.

Dans le cadre de notre recherche, les spécifications ne sont pas rigides puisqu'elles peuvent changer suivant les résultats et les contraintes de développement et aucun client n'est sensé intervenir. C'est la raison pour laquelle, nous avons opté pour le cycle en V comme modèle de développement. Pour éviter beaucoup de retour en arrière pour des difficultés de quelque nature que ce soit, un compromis consiste à appliquer un cycle en V le plus court possible et de faire évoluer le projet sous forme de versions, prenant ainsi en compte le fait que le projet ne sera pas parfait et qu'il sera amélioré au fil des versions. Cela permet également de bénéficier du retour d'expérience des versions précédentes et c'est l'objectif visé par ce travail car nous n'avons pas la prétention de réaliser un travail parfait et nous comptons poursuivre les recherches au delà de la présentation de ce document.

2.3 Modélisation

Pour la modélisation de nos jeux, nous n'avons fixé aucune valeur, de sorte que notre solution puisse vraiment conserver le caractère adaptatif de la solution générique face à la variabilité des paramètres comme la taille du plateau, le premier joueur à commencer la partie, etc. Aussi, avons-nous opté pour la programmation orientée objet afin que notre code (les classes) soit ré-utilisable. La figure 2.2 résume sous forme d'un diagramme de classes UML, ce à quoi ressemblera le système.

2.3.1 Le plateau : la classe *Board*

Nous représentons le plateau par une grille en 2D où chaque cellule repérée par $(i; j)$ représente un emplacement du plateau réel de jeu. Ainsi, notre plateau est de dimension $X \times X$. Les jeux sur lesquels portent notre étude sont des jeux de plateau à dimension variable. Nous

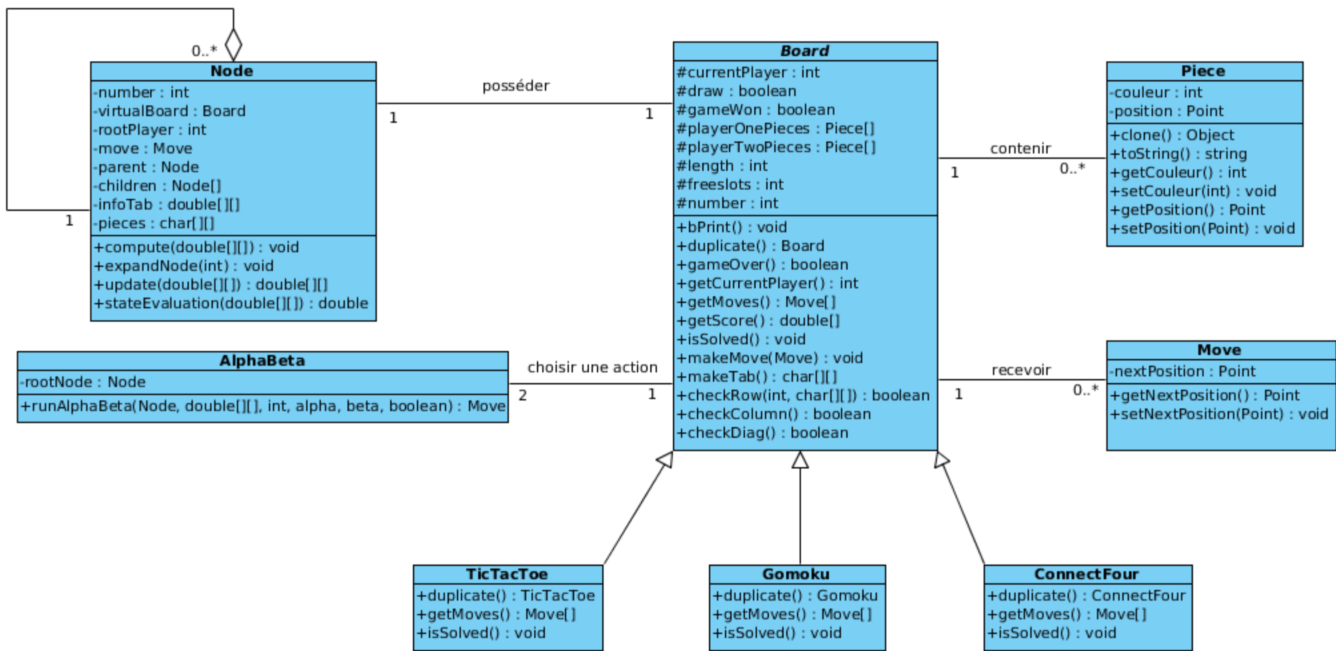

FIGURE 2.2 – Diagramme de classes du système des jeux de type «n-alignés» basé sur alpha-bêta

n'avons donc pas fixé la taille du plateau au sein de notre classe Board qui le représente. Ces valeurs sont des variables pouvant être modifiées et notre implémentation est faite de sorte à s'y adapter. Les éléments du plateau seraient bien représentés par un tableau statique si sa taille était a priori connue mais cette contrainte de non connaissance de la taille du tableau exclut cette possibilité. Pour cela, nous avons pensé à utiliser des tableaux dynamiques. Pour le cas d'espèce, nous en utilisons deux : la première pour garder toutes les pierres posées par le joueur 1 et la deuxième pour le joueur 2 sachant que chaque pierre est un objet qui contient essentiellement des coordonnées représentant leur position sur le plateau. L'emplacement de coordonnées $(0; 0)$ est l'origine du repère et il représente le coin haut à gauche. Pour le cas du Gomoku, le jeu se joue sur un Goban[1] donc les pierres sont posées sur les intersections et non dans les cellules. Mais pour une représentation facile de l'état des plateaux, nous avons choisi de représenter les pierres dans les cellules du plateau. Grâce à la taille du plateau contenue dans les variables de notre objet, nous pouvons à chaque fois vérifier que les pierres qui seront stockées dans nos tableaux dynamiques peuvent réellement être posées sur le plateau. En d'autres termes, pour un plateau de taille (5×5), les coordonnées des pierres doivent être comprises dans l'intervalle $[0; 4] \times [0; 4]$. Pour faire des opérations sur le plateau comme la recherche de combinaison gagnante, etc. nous appliquons sur nos deux tableaux dynamiques une fonction afin de créer un tableau statique de caractères et dont les pierres du joueur 1 sont toujours représentées par la lettre $'O'$ et celles du joueur 2 par la lettre $'X'$. Cette structure devient alors facile à manipuler et à afficher. Nous utilisons des méthodes de vérification des menaces tant

[1]surface plane sur laquelle on joue au jeu d'alignements

```java
package main;

import java.util.ArrayList;

public abstract class Board {

  protected int currentPlayer;
  protected int winner;
  protected boolean draw;
  protected boolean gameWon;
  protected int freeslots;
  protected ArrayList<Piece> playerOnePieces;
  protected ArrayList<Piece> playerTwoPieces;
  protected int length;
  public static int number;

  public abstract Board duplicate();
  public abstract ArrayList<Move> getMoves();
  public char[][] makeTab(){
  }
  public void bprint(){
  }
  public void makeMove(){
  }
  public boolean isSolved(){
  }
  public boolean gameOver(){
  }
  public double getScore(){
  }
  public boolean checkRow(int row, char[][] t){
  }
  public boolean checkColumn(int column, char[][] t){
  }
  public boolean checkDiag(int row, int column,char[][] t){
  }
  public int getCurrentPlayer(){
  }
}
```

FIGURE 2.3 – Aperçu de la classe Board

sur les lignes que sur les colonnes et sur les diagonales pour trouver toutes les menaces présentes sur le plateau de jeu côté joueur et côté adversaire. La figure 2.3 présente un aperçu de la classe Board dépourvue des corps de méthodes.

2.3.2 Le nœud : la classe *Node*

L'algorithme alpha-bêta se résume à exécuter de manière itérative plusieurs simulations aléatoires de l'état actuel jusqu'à une certaine profondeur où des scores sont calculés pour chaque simulation considérée comme étant un état terminal. Ainsi, la notion de nœud prend tout son sens dans notre implémentation. Nous avons donc créé une classe représentant un nœud de notre arbre de recherche itérativement construit par alpha-bêta. La notion de nœud est intimement liée à celle de parent et d'enfants dans la mesure où notre algorithme doit faire une retro-propagation des gains obtenus à la fin de la simulation jusqu'à l'étape où la simulation est lancée. L'heuristique générique que nous exploitons utilisant différents paramètres en fonction

```
package main;

import java.awt.Point;
import java.util.ArrayList;

public class Node {

	public Board virtualBoard;
	public Move move;
	public ArrayList<Node> children;
	public Node parent;
	public int rootPlayer;
	public int player;
	public double[][] infoTab;
	public char[][] pieces;

	public Node(Board b) {
	}
	public Node(Board b, Move m, Node prnt,int ref) {
	}
	public void printinfo() {
	}
	public void expandNode(int ref) {
	}
	public double stateEvaluation(double[][] tab) {
	}
	public double[][] update(double[][] tab){
	}
	public void compute(double[][] tab ) {
	}
}
```

FIGURE 2.4 – Aperçu de la classe Node

du joueur et de l'adversaire pour l'évaluation, il est donc capital que chaque nœud garde l'indice du joueur pour lequel la simulation a été initiée (joueur au nœud racine). Nous gardons en plus, un tableau à double dimension qui se contentera de stocker en mémoire les informations pertinentes concernant les types de menaces détectés au niveau du nœud selon la politique d'évaluation de l'heuristique générique. Cela simplifie la mise à jour des paramètres associés aux menaces que nous aborderons dans le prochain chapitre.Un aperçu de la classe en question est représenté par la figure 2.4.

2.3.3 Le joueur : la classe *AlphaBeta*

Le joueur est représenté par l'algorithme d'élagage alpha-bêta en lui-même. Cette classe implémente l'algorithme 4 défini dans l'état de l'art. Mais nous avons du faire certaines modifications en fonction des exigences de notre travail. Cette classe exploite l'heuristique générique pour l'évaluation des nœuds avec une limitation de profondeur. Nous utilisons un tableau contenant les paramètres des menaces du joueur et de l'adversaire à prendre en compte par l'heuristique pendant l'exécution de l'algorithme. La figure 2.5 en dit plus sur sa structure.

```
1 package main;
2 
3 import java.awt.Point;
4 
5 public class AlphaBeta {
6   private Node rootNode;
7   public Move runAlphaBeta(Node node,double[][] tab, int depth, double alpha, double beta,boolean maxP){
8   }
9 }
```

FIGURE 2.5 – Aperçu de la classe AlphaBeta

Conclusion

Dans ce chapitre, nous avons justifié le choix du langage de programmation et décrit les caractéristiques de l'ordinateur utilisé pour l'implémentation. Nous avons présenté le cycle de développement de V et la modélisation du système. Le chapitre suivant présente les solutions que nous proposons.

Chapitre 3

Solutions

Résumé. Les éléments stratégiques concourant à la détermination des paramètres ont été trouvés en se basant sur la recherche des menaces et sur l'apprentissage automatique. Certaines menaces sont importantes et irréversibles pendant que d'autres sont moins importantes. L'inventaire de ces dernières à un instant t du jeu et à un instant $t+1$ permet d'améliorer la politique de choix des actions en jouant sur les paramètres utilisés pour l'évaluation d'une position. Ces paramètres, attribués à chaque menace, sont déterminés par l'application de l'algorithme Q-learning.

Introduction

Dans ce chapitre, nous présentons la méthode d'apprentissage pour laquelle nous avons opté pour la détermination des paramètres de l'heuristique générique afin de les rendre meilleurs pour chaque jeu pris en considération.

3.1 Détermination de la méthode d'apprentissage

Nous rappelons que l'heuristique générique permet d'évaluer les positions du plateau de jeu. Cette évaluation dépend essentiellement des paramètres associés aux différents types de menaces. A partir d'un état donné, faire une action (jouer un coup) conduit à la création ou à l'annihilation d'une menace. Il existe donc un lien étroit entre action et menace. Dans ce sens, nous devons donc nous intéresser à des méthodes d'apprentissage prenant en compte les actions pour assurer l'atteinte de nos objectifs. Comme détaillé dans le chapitre 1, il existe principalement trois (03) méthodes d'apprentissage basées sur les différences temporelles.

L'algorithme 1.1 du TD(0) travaille sur les valeurs des états et son fonctionnement n'est pas adapté pour le problème en résolution. Il permet d'avoir une évaluation du plateau du jeu à un instant donné. Il aurait été idéal si notre objectif était de remplacer l'heuristique générique elle-même vu qu'elle évalue aussi les états.

Les deux (02) autres méthodes que sont Q-Learning et Sarsa se ressemblent beaucoup. Cependant, il faut noter une différence essentielle. Le terme $Q(S_{t+1}, A_{t+1})$ apparaissant dans l'équation de mise à jour (1.2) de la méthode Sarsa a été remplacé par le terme $maxQ(S_{t+1}, a)$ dans l'équation de mise à jour (1.3) du Q-learning. Il apparaît donc que l'algorithme Sarsa effectue les mises à jour en fonction des actions choisies effectivement (qui peuvent ne pas être optimales) alors que l'algorithme Q-learning effectue les mises à jour en fonction de la valeur optimale des actions possibles après l'état suivant même si ce n'est pas l'action qui correspond à cette valeur que l'agent réalise. Il apprend donc en supposant qu'on fait des mouvements optimaux. Cette petite différence rend Q-learning un peu plus efficace [11].

Les mises à jour effectuées par l'algorithme Sarsa impliquent que l'agent détermine avec un pas de regard en avant quelle est l'action A_{t+1} qu'il réalisera lors du pas de temps suivant, lorsque l'action a dans l'état s l'aura conduit dans l'état S_{t+1}. Il résulte de cette implication que l'agent est contraint de réaliser l'apprentissage uniquement sur la base de la politique qu'il suit effectivement. La dépendance que cela induit entre l'exploration et l'apprentissage complique considérablement la mise au point de preuves de convergence pour cet algorithme, ce qui explique que les preuves de convergence soient apparues beaucoup plus tard pour les algorithmes « off-policy » tels que Q-learning [12].

Au regard des spécifications de notre travail, des principes de fonctionnement et des différences notables entre les méthodes d'apprentissage présentées, nous avons choisi d'utiliser le Q-learning pour améliorer l'heuristique générique.

3.2 Amélioration de l'heuristique générique

Dans cette section nous présentons la solution que nous proposons dans le but d'améliorer les évaluations faites par l'heuristique pour chaque jeu de type « n-alignés ».

L'heuristique générique est fondée sur la notion de menace. Les menaces présentes sur le plateau de jeu sont identifiées puis catégorisées. Les paramètres de l'heuristique sont attribués selon l'importance des menaces et sont les mêmes pour chaque jeu. L'idéal est de pouvoir déterminer les paramètres adaptés aux situations rencontrées au cours de la résolution du jeu et aux règles de déplacement. Nous proposons donc un moyen de mettre à jour automatiquement les paramètres de l'heuristique avec la méthode du Q-learning.

Q-learning utilise une fonction Q d'évaluation de la qualité. Ceci permet d'avoir une table de valeurs (valeurs Q) qui aidera dans le choix d'une action lorsqu'on se trouve dans un état donné. La valeur des paramètres influe grandement sur l'évaluation d'une position et par

conséquent sur le choix du coup à jouer. Nous considérons donc, qu'à laide du Q-learning, nous devons évaluer la qualité des paramètres de l'heuristique. Les valeurs Q sont les paramètres à déterminer. Avant que l'apprentissage ne commence, la fonction Q renvoie une valeur fixe choisie par le concepteur. Nous rappelons que l'heuristique utilisait des paramètres fixes déterminés expérimentalement et qui ont fait leur preuve. Ce sont ces paramètres qui nous servent de valeurs Q initiales.

L'algorithme permet de mettre à jour une valeur par pas de temps, un paramètre dans notre cas. Il convient de trouver le paramètre à mettre à jour pour un pas donné. On obtient un nouveau pas lorsqu'un coup est joué. L'algorithme Alpha-bêta nous renvoie le coup estimé comme meilleur pour passer au pas de temps suivant. Nous cherchons toutes les menaces présentes sur le plateau de jeu après un mouvement et identifions la menace la plus importante. C'est sur le paramètre associé à cette dernière que porte la mise à jour.

Aussi, en appliquant le principe de l'apprentissage par renforcement, nous associons une récompense à chaque type de menace puisque l'action choisie conduit à une certaine configuration de menaces. En général, elle est nulle sauf pour l'état goal (état du plateau où le joueur a les "n" pions requis alignés). L'identification de la menace la plus importante permet, à cette étape, de savoir quelle récompense utiliser pour la mise à jour.

La mise à jour effective se fait en suivant la formule donnée lors de la présentation du Q-learning et nécessite de trouver la valeur Q maximale lorsqu'on se trouve dans l'état suivant. Il faudra déterminer les types de menace que les mouvements légaux pourraient créer afin de prendre le maximum de leurs paramètres. Si nous disposons d'un état S qui, après une action A, conduit à un état S', nous simulons les coups possibles à partir de l'état S', recueillons les menaces les plus importantes susceptibles d'être créées après chaque coup et identifions la plus importante de toutes. Son paramètre est donc la valeur maximale Q recherchée.

L'évaluation d'une position prend en compte les menaces du joueur et de l'adversaire. Par conséquent, nous mettons aussi à jour les paramètres relatifs aux menaces de l'adversaire pour éviter toute mauvaise évaluation de l'heuristique. Cette mise à jour se fait en suivant les mêmes principes que nous avons décrit pour le joueur. Les paramètres constituent les valeurs Q initiales et nous identifions la menace adversaire la plus importante créée par une action. La fonction Q est utilisée pour le paramètre correspondant et requiert de trouver la valeur maximale des paramètres pour les menaces adversaires les plus importantes obtenues après la simulation à partir de l'état S'.

Nous effectuons deux mises à jour à chaque pas de temps : une sur le paramètre associé à la menace la plus importante du joueur et l'autre sur le paramètre associé à la menace la plus importante de l'adversaire.

En outre, le taux d'apprentissage et le facteur de réduction de la formule de mise à jour ont un grand impact sur le déroulement de l'apprentissage. L'environnement des jeux en étude étant entièrement déterministe, nous avons choisi un taux d'apprentissage de 1. Le facteur de

réduction a été choisi de façon à respecter les exigences de base de l'heuristique. Une valeur trop proche de 1 ferait que les coefficients des menaces soient trop proches les uns des autres. Ce qui entraîne une mauvaise évaluation des positions et rend donc l'heuristique moins performante. Une valeur proche de 0 assure dans une certaine mesure un écart favorable entre les paramètres. Après de nombreux tests, nous avons retenu $\gamma = 0.1$. La formule de mise à jour (1.3) devient :

$$Q(S, A) = R + \gamma max Q(S', a) \tag{3.1}$$

L'algorithme 8 nous permet d'adapter l'algorithme 7 du Q-learning à notre solution.

Algorithme 8 : Algorithme de mise à jour des paramètres

```
INITIALISER LA TABLE DE VALEURS AVEC LES PARAMÈTRES FIXES EXISTANTS
répéter
    Initialiser S avec l'état courant du plateau
    répéter pour chaque étape de l'épisode
        Choisir A // mouvement issu de l'algorithme alpha-bêta
            utilisant l'heuristique générique

        Exécuter l'action A
        Trouver le paramètre P_J associé à la menace la plus importante du joueur
        Trouver le paramètre P_A associé à la menace la plus importante de l'adversaire
        Recevoir la récompense R_J correspondante à la menace la plus importante du joueur

        Recevoir la récompense R_A correspondante à la menace la plus importante de l'adversaire

        Observer l'état suivant S'
        P_J(S, A) ← R_J + γ maxP_J(S', a) // maxP_J(S', a) est la valeur maximale
            des paramètres pour les menaces les plus importantes du
            joueur susceptibles d'être créées à partir du nouvel état
            S'

        P_A(S, A) ← R_A + γ maxP_A(S', a) // maxP_A(S', a) est la valeur maximale
            des paramètres pour les menaces les plus importantes de
            l'adversaire susceptibles d'être créées à partir du
            nouvel état S'

        S ← S'
    jusqu'à S terminal
jusqu'à la fin de l'épisode
```

A chaque étape de l'épisode, l'algorithme alpha-bêta utilise l'heuristique générique avec les nouveaux paramètres pour la sélection d'une action (d'un mouvement). Les mises à jour provoquent parfois des décalages au niveau des paramètres. Les coefficients de certaines menaces deviennent inférieurs aux coefficients de menaces moins intéressantes. Ceci n'est pas favorable à l'évaluation des positions faite par l'heuristique générique. Nous avons effectué des ajustements pour gérer ces cas de figure et assurer au mieux le fonctionnement idéal de l'heuristique.

Au bout d'un nombre suffisant d'épisodes d'apprentissage, nous observons une convergence des paramètres vers des valeurs supposées être meilleures à celles fixées expérimentalement.

Pour faire la mise à jour d'un paramètre du joueur, nous devons simuler tous les coups possibles à partir de l'état courant. Nous obtenons plusieurs états possibles du plateau de jeu et nous utilisons des algorithmes de parcours de notre tableau bidimensionnel (représentation du plateau de jeu) afin de détecter les éventuelles menaces tant sur les lignes, que sur les colonnes et sur les deux diagonales pour chaque état simulé. La méthode globale de détection des menaces utilisant les différents algorithmes de parcours a une complexité de $O(m^3)$ [1] (avec m la taille du plateau) et est appelée en boucle afin d'être appliquée à tous les états simulés. Soit b le nombre d'états simulés à partir de l'état courant. Notre algorithme a donc une complexité de $O(bm^3)$. Toutes les menaces détectées sont consignées dans un tableau que nous parcourons ensuite pour déterminer la menace la plus importante et donc le paramètre qui lui est associé. Cette méthode est utilisée dans l'algorithme de mise à jour que nous avons écrit. Etant donné que nos jeux sont à deux joueurs, à somme nulle, il nous faut faire de même pour l'adversaire. Dans chaque cas, notre algorithme a une complexité de $O(bm^3)$. La figure 3.1 montre la méthode java permettant de mettre à jour le paramètre associé à la menace la plus importante du joueur détectée sur le plateau de jeu.

```
1  public double[][] update(double[][] tab){
2   double y=0.1;
3   double[][] maxSearch;
4   maxSearch = new double[2][infoTab[0].length];
5
6
7   if(indMenJ == infoTab[0].length -1){
8         infoTab[1][indMenJ] = infoTab[1][indMenJ];
9   }
10  else if(indMenJ != -1){
11        virtualBoard.currentPlayer = (virtualBoard.getCurrentPlayer() == 1) ? 2 : 1;
12        int joueur = virtualBoard.currentPlayer;
13        expandNode(joueur);
14     for (int i = 0; i < children.size(); i++){
15        Board newBoard= children.get(i).virtualBoard.duplicate();
16      newBoard.makeMove(children.get(i).move);
17      children.get(i).pieces=newBoard.makeTab();
18          children.get(i).compute(tab); //détection des menaces
19          int menJ=0;
20
21          for (int j = 0; j < children.get(i).infoTab[0].length; j++){
22            if(children.get(i).infoTab[0][j]!=0 && j >= menJ){
23              menJ = j;
24            }
25          }
26          maxSearch[0][menJ] += children.get(i).infoTab[0][menJ];
27     }
28     double maxJ=0;
29     for (int i = 0; i < maxSearch[0].length ; i++){
30          if(maxSearch[0][i]!=0 && infoTab[1][i] > maxJ){
31            maxJ = infoTab[1][i];
32          }
33     }
34        infoTab[1][indMenJ]= infoTab[4][indMenJ] + (y * maxJ);
35        virtualBoard.currentPlayer = (virtualBoard.getCurrentPlayer() == 1) ? 2 : 1;
36  }
37 }
```

FIGURE 3.1 – Méthode de mise à jour du paramètre de la menace la plus importante du joueur

Conclusion

Ce chapitre a présenté la solution proposée pour notre problème. Nous avons abordé les particularités des méthodes de contrôle des différences temporelles et abouti à l'utilisation de l'algorithme Q-learning pour la détermination automatique des paramètres de l'heuristique générique. Le chapitre suivant présente les résultats des tests de notre implémentation.

Chapitre 4

Résultats et discussion

Résumé. L'heuristique intégrant les paramètres déterminés de façon automatique, est combinée avec l'algorithme Alpha-bêta et testée suivant plusieurs critères à savoir le jeu pris en considération, le premier joueur, la taille du plateau et le nombre de pierres à aligner. Elle se montre un peu plus performante que l'heuristique dont les paramètres ont été fixés expérimentalement.

Introduction

Dans ce chapitre, nous exposons les résultats des expérimentations que nous avons faites après implémentation de notre solution. Après cela, nous faisons une analyse afin de pouvoir tirer des connaissances de ces résultats.

Nous avons testé plusieurs profondeurs limites pour l'exécution de l'algorithme Alpha-bêta. A cause du compromis entre temps et précision, nous avons retenu de faire nos expériences à une profondeur limite de 4 et dans un environnement de taille variable pour chaque jeu en étude car le véritable problème de l'algorithme Alpha-bêta dans le domaine des jeux de type « n-alignés » est l'immensité de l'espace de recherche. Le nombre de pierres à aligner pour remporter une partie est variable mais nous nous limitons à un maximum de 5. L'obtention de meilleurs paramètres résultant de la mise à jour nécessite un temps d'apprentissage. Nous avons effectué de nombreux tests afin de déterminer le temps qui permettrait à la solution proposée de fournir un ensemble de paramètres stables. Après analyse des résultats, nous faisons jouer l'agent intelligent contre lui-même pendant 5 épisodes car à partir de ce niveau, les paramètres semblent ne plus changer et chacun d'eux semble avoir été mis à jour au moins une fois.

Aussi avons nous recherché des heuristiques basées sur la notion de menace, spécifiques à chaque jeu. Nous prenons en compte dans les tests l'heuristique proposée par Shevchenko [24] pour le jeu Gomoku et l'heuristique proposée par Chua Hock Chuan [35] pour Tic Tac Toe.

Notons que pour parvenir à des résultats faciles à analyser, nous avons créé des catégories de test au sein desquelles plusieurs expérimentations ont été faites.

4.1 Présentation des performances de la solution proposée

Après avoir implémenté l'apprentissage automatique des paramètres, nous avons fait les tests avec un joueur qui utilise alpha-bêta et l'*heuristique améliorée* [1] à une profondeur limite de 4. Nous vérifions ses performances en le faisant jouer contre un joueur qui utilise alpha-bêta et l'*heuristique brute*[2] pour chacun des jeux que sont Gomoku, Tic tac toe et Connect four.

Pour Gomoku, nous faisons également des tests contre un joueur qui utilise alpha-bêta et l'heuristique de Shevchenko [24]. Pour cette heuristique, une analyse des combinaisons de pièces présentes sur le plateau de jeu est faite tant sur les lignes que sur les colonnes et les diagonales. L'analyse concerne uniquement les pièces du joueur. Dans ce sens, Shevchenko prend uniquement en compte les menaces du joueur présentes sur le plateau. Il s'agit des menaces de taille n, $n-1$ et $n-2$ sans distinction entre les types mi-ouvert et ouvert. Des paramètres sont associés aux menaces selon leur taille et restent inchangés jusqu'à la fin de la partie. Ces paramètres sont : 100 pour la menace de taille n, 10 pour la menace de taille $n-1$ et 1 pour la menace de taille $n-2$.

Pour Tic tac toe, en plus des tests portant sur l'heuristique améliorée et l'heuristique brute, nous comparons l'heuristique améliorée et l'heuristique de Chua Hock Chuan [35]. L'application de cette heuristique nécessite de trouver les alignements de pièces du joueur et de l'adversaire tant sur les lignes que sur les colonnes et sur les diagonales. Elle considère donc les menaces du joueur et de l'adversaire présentes sur le plateau mais seulement celles de taille n, $n-1$ et $n-2$. Il n'y a pas de distinction entre les types mi-ouvert et ouvert. Les paramètres associés aux menaces du joueur et de l'adversaire sont fixes jusqu'à la fin de la partie. Nous avons :

- menace de taille n : 100 pour le joueur et -100 pour l'adversaire ;
- menace de taille $n-1$: 10 pour le joueur et -10 pour l'adversaire ;
- menace de taille $n-2$: 1 pour le joueur et -1 pour l'adversaire.

Nous avons procédé à plusieurs essais regroupés en catégories. Pour chaque jeu pris en considération, nous distinguons trois (03) catégories axées sur le nombre de pièces à aligner

[1] heuristique avec les paramètres mis à jour automatiquement avec le Q-learning
[2] heuristique aux paramètres fixés expérimentalement

pour remporter la partie :

- catégorie 1 : nombre de pièces à aligner n = 3 ;
- catégorie 2 : nombre de pièces à aligner n = 4 ;
- catégorie 3 : nombre de pièces à aligner n = 5.

Les tests sont réalisés pour chaque catégorie en tenant compte du joueur qui commence la partie.

4.1.1 Test des performances de la solution proposée pour Gomoku

4.1.1.1 Heuristique améliorée et heuristique brute

Catégorie 1 : nombre de pièces à aligner $n = 3$

TABLEAU 4.1 – Gomoku : heuristique améliorée vs heuristique brute pour $n = 3$, test n°1

Opposants	Premier joueur	Taille = 3	Taille = 4	Taille $\in$ [5,11]
Alpha-bêta + heuristique avec Q-learning	Oui	Match nul	Défaite	Victoire
Alpha-bêta + heuristique brute	Non	Match nul	Victoire	Défaite

TABLEAU 4.2 – Gomoku : heuristique améliorée vs heuristique brute pour $n = 3$, test n°2

Opposants	Premier joueur	Taille = 3	Taille = 4	Taille = $\{$5,6$\}$	Taille $\in$ [7,11]
Alpha-bêta + heuristique avec Q-learning	Non	Match nul	Victoire	Défaite	Victoire
Alpha-bêta + heuristique brute	Oui	Match nul	Défaite	Victoire	Défaite

Analyse 1 Avec $n = 3$, les tests ont montré que pour une taille supérieure ou égale à 7, notre approche remporte toujours la partie qu'elle joue en premier ou non. Pour les tailles inférieures, la victoire est obtenue par le premier joueur sauf pour une taille de 4 où c'est le second joueur qui se crée une situation favorable. Remarquons que comme second joueur, notre approche arrive à se défendre et à remporter des parties.

Catégorie 2 : nombre de pièces à aligner $n = 4$

TABLEAU 4.3 – Gomoku : heuristique améliorée vs heuristique brute pour $n = 4$, test n°3

Opposants	**Premier joueur**	**Taille ∈ [4,7]**	**Taille ∈ [8,11]**
Alpha-bêta + heuristique avec Q-learning	Oui	Match nul	Victoire
Alpha-bêta + heuristique de brute	Non	Match nul	Défaite

TABLEAU 4.4 – Gomoku : heuristique améliorée vs heuristique brute pour $n = 4$, test n°4

Opposants	**Premier joueur**	**Taille ∈ [4,7]**	**Taille ∈ [8,11]**
Alpha-bêta + heuristique avec Q-learning	Non	Match nul	Victoire
Alpha-bêta + heuristique brute	Oui	Match nul	Défaite

Analyse 2 Avec $n = 4$, notre approche remporte toujours la partie qu'elle joue en premier ou non pour une taille supérieure ou égale à 8.

Catégorie 3 : nombre de pièces à aligner $n = 5$

TABLEAU 4.5 – Gomoku : heuristique améliorée vs heuristique brute pour $n = 5$, test n°5

Opposants	**Premier joueur**	**Taille ∈ [5,10]**	**Taille = 11**
Alpha-bêta + heuristique avec Q-learning	Non	Match nul	Victoire
Alpha-bêta + heuristique brute	Oui	Match nul	Défaite

TABLEAU 4.6 – Gomoku : heuristique améliorée vs heuristique brute pour $n = 5$, test n°6

Opposants	**Premier joueur**	**Taille ∈ [5,11]**
Alpha-bêta + heuristique avec Q-learning	Non	Match nul
Alpha-bêta + heuristique brute	Oui	Match nul

Analyse 3 Avec $n = 5$, les parties sont plutôt équilibrés indépendamment du premier joueur. Mais notre approche en tant que premier joueur remporte la partie et tient un match nul comme second joueur lorsque la taille du plateau est de 11. Sa défense semble meilleure. $n = 5$ ne permet pas à notre approche de se créer aisément des situations avantageuses.

4.1.1.2 Heuristique améliorée et heuristique de Shevchenko

Catégorie 1 : nombre de pièces à aligner $n = 3$

TABLEAU 4.7 – Gomoku : heuristique améliorée vs heuristique de Shevchenko pour $n = 3$, test n°7

Opposants	Premier joueur	Taille $\in$ [3,11]
Alpha-bêta + heuristique avec Q-learning	Oui	Victoire
Alpha-bêta + heuristique de Shevchenko	Non	Défaite

TABLEAU 4.8 – Gomoku : heuristique améliorée vs heuristique de Shevchenko pour $n = 3$, test n°8

Opposants	Premier joueur	Taille $\in$ {3,4,5,6,8,9,10,11}	Taille = 7
Alpha-bêta + heuristique avec Q-learning	Non	Défaite	Victoire
Alpha-bêta + heuristique de Shevchenko	Oui	Victoire	Défaite

Analyse 4 Nous avons décidé de contrer notre approche avec une heuristique déjà utilisée pour ce jeu. Avec $n = 3$ le premier joueur gagne toujours la partie quelle que soit la taille du plateau à une exception près car pour une taille de 7 notre approche remporte la partie en tant que second joueur. Elle s'est donc défendu et s'est créée des situations avantageuses. Nous concluons que pour $n = 3$, le premier joueur a un avantage qui le conduit à la victoire. Pour de meilleures analyses, nous changeons n.

Catégorie 2 : nombre de pièces à aligner $n = 4$

TABLEAU 4.9 – Gomoku : heuristique améliorée vs heuristique de Shevchenko pour $n = 4$, test n°9

Opposants	Premier joueur	Taille = 4	Taille ∈ [5,11]
Alpha-bêta + heuristique avec Q-learning	Oui	Match nul	Victoire
Alpha-bêta + heuristique de Shevchenko	Non	Match nul	Défaite

TABLEAU 4.10 – Gomoku : heuristique améliorée vs heuristique de Shevchenko pour $n = 4$, test n°10

Opposants	Premier joueur	Taille = 4	Taille ∈ [5,11]
Alpha-bêta + heuristique avec Q-learning	Non	Match nul	Victoire
Alpha-bêta + heuristique de Shevchenko	Oui	Match nul	Défaite

Analyse 5 Avec $n = 4$, notre approche remporte toujours la partie qu'elle joue en premier ou non pour une taille supérieure à n. Elle se montre donc très performante à ce niveau.

Catégorie 3 : nombre de pièces à aligner $n = 5$

TABLEAU 4.11 – Gomoku : heuristique améliorée vs heuristique de Shevchenko pour $n = 5$, test n°11

Opposants	Premier joueur	Taille ∈ [5,11]
Alpha-bêta + heuristique avec Q-learning	Oui	Victoire
Alpha-bêta + heuristique de Shevchenko	Non	Défaite

TABLEAU 4.12 – Gomoku : heuristique améliorée vs heuristique de Shevchenko pour $n = 5$, test n°12

Opposants	**Premier joueur**	**Taille = 5**	**Taille ∈ [6,11]**
Alpha-bêta + heuristique avec Q-learning	Non	Match nul	Victoire
Alpha-bêta + heuristique de Shevchenko	Oui	Match nul	Défaite

Analyse 6 Avec $n = 5$, notre approche remporte toujours la partie qu'elle joue en premier ou non pour une taille supérieure à n. Elle se montre plus performante que l'heuristique de Shevchenko même si la partie se termine par un match nul qaund elle joue en second pour $n = 5$.

4.1.2 Test des performances de la solution proposée pour Tic Tac Toe

4.1.2.1 Heuristique améliorée et heuristique brute

Catégorie 1 : nombre de pièces à aligner $n = 3$

TABLEAU 4.13 – Tic tac toe : heuristique améliorée vs heuristique brute pour $n = 3$, test n°13

Opposants	**Premier joueur**	**Taille = 3**	**Taille = 4**	**Taille ∈ [5,11]**
Alpha-bêta + heuristique avec Q-learning	Oui	Match nul	Défaite	Victoire
Alpha-bêta + heuristique brute	Non	Match nul	Victoire	Défaite

TABLEAU 4.14 – Tic tac toe : heuristique améliorée vs heuristique brute pour $n = 3$, test n°14

Opposants	**Premier joueur**	**Taille = 3**	**Taille = 4**	**Taille = {5,6}**	**Taille ∈ [7,11]**
Alpha-bêta + heuristique avec Q-learning	Non	Match nul	Victoire	Défaite	Victoire
Alpha-bêta + heuristique brute	Oui	Match nul	Défaite	Victoire	Défaite

Analyse 7 Les résultats sont pareils aux résultats de Gomoku (heuristique améliorée et heuristique brute) avec $n = 3$.

Catégorie 2 : nombre de pièces à aligner $n = 4$

TABLEAU 4.15 – Tic tac toe : heuristique améliorée vs heuristique brute pour $n = 4$, test n°15

Opposants	Premier joueur	Taille ∈ [4,7]	Taille ∈ [8,11]
Alpha-bêta + heuristique avec Q-learning	Oui	Match nul	Victoire
Alpha-bêta + heuristique de brute	Non	Match nul	Défaite

TABLEAU 4.16 – Tic tac toe : heuristique améliorée vs heuristique brute pour $n = 4$, test n°16

Opposants	Premier joueur	Taille ∈ [4,7]	Taille ∈ [8,11]
Alpha-bêta + heuristique avec Q-learning	Non	Match nul	Victoire
Alpha-bêta + heuristique brute	Oui	Match nul	Défaite

Analyse 8 Les résultats sont pareils aux résultats de Gomoku (heuristique améliorée et heuristique brute) avec $n = 4$.

Catégorie 3 : nombre de pièces à aligner $n = 5$

TABLEAU 4.17 – Tic tac toe : heuristique améliorée vs heuristique brute pour $n = 5$, test n°17

Opposants	Premier joueur	Taille ∈ [5,10]	Taille = 11
Alpha-bêta + heuristique avec Q-learning	Non	Match nul	Victoire
Alpha-bêta + heuristique brute	Oui	Match nul	Défaite

TABLEAU 4.18 – Tic tac toe : heuristique améliorée vs heuristique brute pour $n = 5$, test n°18

Opposants	Premier joueur	Taille ∈ [5,11]
Alpha-bêta + heuristique avec Q-learning	Non	Match nul
Alpha-bêta + heuristique brute	Oui	Match nul

Analyse 9 Les résultats sont pareils aux résultats de Gomoku (heuristique améliorée et heuristique brute) avec $n = 5$.

4.1.2.2 Heuristique améliorée et heuristique de Chua Hock Chuan

Catégorie 1 : nombre de pièces à aligner $n = 3$

TABLEAU 4.19 – Tic tac toe : heuristique améliorée vs heuristique de Chua Hock Chuan pour $n = 3$, test n°19

Opposants	**Premier joueur**	**Taille = 3**	**Taille = {4,6,7}**	**Taille = {5,8,9,10,11}**
Alpha-bêta + heuristique avec Q-learning	Oui	Match nul	Défaite	Victoire
Alpha-bêta + heuristique de Chua Hock Chuan	Non	Match nul	Victoire	Défaite

TABLEAU 4.20 – Tic tac toe : heuristique améliorée vs heuristique de Chua Hock Chuan pour $n = 3$, test n°20

Opposants	**Premier joueur**	**Taille = 3**	**Taille $\in$ [4,11]**
Alpha-bêta + heuristique avec Q-learning	Non	Match nul	Défaite
Alpha-bêta + heuristique de Chua Hock Chuan	Oui	Match nul	Victoire

Analyse 10 En tant que second joueur, l'heuristique générique améliorée est totalement surpassée et n'est pas très efficace en tant que premier joueur.

Catégorie 2 : nombre de pièces à aligner $n = 4$

TABLEAU 4.21 – Tic tac toe : heuristique améliorée vs heuristique de Chua Hock Chuan pour $n = 4$, test n°21

Opposants	**Premier joueur**	**Taille = {4,5}**	**Taille $\in$ [6,11]**
Alpha-bêta + heuristique avec Q-learning	Oui	Match nul	Victoire
Alpha-bêta + heuristique de Chua Hock Chuan	Non	Match nul	Défaite

TABLEAU 4.22 – Tic tac toe : heuristique améliorée vs heuristique de Chua Hock Chuan pour $n = 4$, test n°22

Opposants	Premier joueur	Taille = $\{4,5\}$	Taille $\in$ [6,11]
Alpha-bêta + heuristique avec Q-learning	Non	Match nul	Victoire
Alpha-bêta + heuristique de Chua Hock Chuan	Oui	Match nul	Défaite

Analyse 11 Avec $n = 4$, notre approche remporte toujours la partie qu'elle joue en premier ou non pour une taille supérieure ou égale à 6. Elle ne concède aucune défaite. Ce nombre de pièces à aligner lui est donc favorable.

Catégorie 3 : nombre de pièces à aligner $n = 5$

TABLEAU 4.23 – Tic tac toe : heuristique améliorée vs heuristique de Chua Hock Chuan pour $n = 5$, test n°23

Opposants	Premier joueur	Taille $\in$ [5,7]	Taille $\in$ [8,11]
Alpha-bêta + heuristique avec Q-learning	Oui	Match nul	Victoire
Alpha-bêta + heuristique de Chua Hock Chuan	Non	Match nul	Défaite

TABLEAU 4.24 – Tic tac toe : heuristique améliorée vs heuristique de Chua Hock Chuan pour $n = 5$, test n°24

Opposants	Premier joueur	Taille = $\{5,6\}$	Taille $\in$ [7,11]
Alpha-bêta + heuristique avec Q-learning	Non	Match nul	Victoire
Alpha-bêta + heuristique de Chua Hock Chuan	Oui	Match nul	Défaite

Analyse 12 Notre approche remporte toujours la partie qu'elle joue en premier ou non pour une taille supérieure ou égale à 8. Elle ne concède aucune défaite. Tout comme $n = 4$, $n = 5$ est favorable pour notre approche.

4.1.3 Test des performances de la solution proposée pour Connect four

Catégorie 1 : nombre de pièces à aligner $n = 3$

TABLEAU 4.25 – Connect four : heuristique améliorée vs heuristique brute pour $n = 3$, test n°25

Opposants	Premier joueur	Taille = 3	Taille = {4,8,6,11}	Taille = {5,7,9,10}
Alpha-bêta + heuristique avec Q-learning	Oui	Match nul	Victoire	Défaite
Alpha-bêta + heuristique brute	Non	Match nul	Défaite	Victoire

TABLEAU 4.26 – Connect four : heuristique améliorée vs heuristique brute pour $n = 3$, test n°26

Opposants	Premier joueur	Taille = 3	Taille = {4,5,7,9}	Taille = {6,8,10,11}
Alpha-bêta + heuristique avec Q-learning	Non	Match nul	Défaite	Victoire
Alpha-bêta + heuristique brute	Oui	Match nul	Victoire	Défaite

Analyse 13 L'issue des parties est aléatoire. Quelle que soit l'approche qui joue en premier, nous ne déduisons aucune suite logique dans les résultats. L'une des approche finit souvent par remporter la partie car lorsqu'elle se retrouve souvent dans une situation intéressante. Nous allons donc dans les prochains jeux de test augmenter le nombre de pièces à aligner pour faire une nouvelle analyse.

Catégorie 2 : nombre de pièces à aligner $n = 4$

TABLEAU 4.27 – Connect four : heuristique améliorée vs heuristique brute pour $n = 4$, test n°27

Opposants	Premier joueur	Taille = {4,5}	Taille $\in$ [6,11]
Alpha-bêta + heuristique avec Q-learning	Oui	Match nul	Victoire
Alpha-bêta + heuristique de brute	Non	Match nul	Défaite

TABLEAU 4.28 – Connect four : heuristique améliorée vs heuristique brute pour $n = 4$, test n°28

Opposants	Premier joueur	Taille = {4,5,7}	Taille = {6,10,11}	Taille = {8,9}
Alpha-bêta + heuristique avec Q-learning	Non	Match nul	Défaite	Victoire
Alpha-bêta + heuristique brute	Oui	Match nul	Victoire	Défaite

Analyse 14 En tant que premier joueur, notre approche assure la victoire pour une taille du plateau supérieure ou égale à 6. En tant que second joueur, elle empêche l'heuristique brute de parvenir aux mêmes résultats. Elle se défend bien et arrive parfois à jouer des coups menant à la victoire.

Catégorie 3 : nombre de pièces à aligner $n = 5$

TABLEAU 4.29 – Connect four : heuristique améliorée vs heuristique brute pour $n = 5$, test n°29

Opposants	Premier joueur	Taille = {5,8}	Taille = {6,9,10}	Taille = {7,11}
Alpha-bêta + heuristique avec Q-learning	Oui	Match nul	Victoire	Défaite
Alpha-bêta + heuristique brute	Non	Match nul	Défaite	Victoire

TABLEAU 4.30 – Connect four : heuristique améliorée vs heuristique brute pour $n = 5$, test n°30

Opposants	Premier joueur	Taille = {5,6,8}	Taille = {7,9,10}	Taille = {11}
Alpha-bêta + heuristique avec Q-learning	Oui	Match nul	Défaite	Victoire
Alpha-bêta + heuristique brute	Non	Match nul	Victoire	Défaite

Analyse 15 Quelle que soit l'approche qui joue en premier, nous ne déduisons aucune suite logique dans les résultats. L'une des approche finit souvent par remporter la partie car lorsqu'elle se retrouve souvent dans une condition intéressante.

4.2 Discussion

L'heuristique générique a été déterminée afin de fournir une formule commune d'évaluation des nœuds pour les jeux de type «n-alignés». Cette heuristique utilisait des paramètres fixes définis expérimentalement et a fait ses preuves en terme d'efficacité [1]. Cependant, la détermination expérimentale des paramètres n'assurent pas toujours une bonne évaluation des nœuds suivant les situations susceptibles d'être créées pendant qu'on joue aux différents jeux de notre catégorie.

Nous avons donc proposé une méthode d'apprentissage des paramètres afin de permettre à l'heuristique générique de s'adapter à chaque jeu de notre catégorie. Elle se charge, à chaque coup, de procéder à une mise à jour de paramètres associés aux menaces les plus importantes du joueur et de l'adversaire.

Pour évaluer son efficacité, nous avons utilisé l'heuristique améliorée avec la méthode Q-learning et l'algorithme alpha-bêta. Notre approche apprend les paramètres en jouant contre elle-même afin de favoriser la convergence des paramètres vers des valeurs intéressantes. Nous l'avons fait jouer aux trois (03) jeux en étude contre l'heuristique brute à une profondeur limite de 4.

Pour Gomoku, notre approche s'est montrée efficace contre l'heuristique brute en tant que premier joueur où elle a un pourcentage élevé de victoire quand on prend en compte la taille du plateau de jeu et le nombre de pièces à aligner. Elle s'est aussi bien défendue comme second joueur, obtient de meilleurs résultats que ceux de l'heuristique brute dans la même position et arrive même à remporter des parties. Nous avons réalisé une série de tests supplémentaires en couplant l'heuristique de Shevchenko [24] à alpha-bêta. Le joueur utilisant l'heuristique générique améliorée a remporté la majeure partie des matchs quels que soient le nombre de pièces à aligner et la taille du plateau.

Pour Tic Tac Toe, les résultats et les conclusions sont pareils à ceux obtenus pour Gomoku en jouant contre l'heuristique brute. Les tests supplémentaires avec l'heuristique de Chua Hock Chuan [35] ont montré l'efficacité de cette dernière pour $n = 3$ mais notre approche a été performante pour $n > 3$.

Pour Connect four, les résultats sont plutôt aléatoires et ne favorisent pas une analyse claire. Cela est peut-être dû à la configuration différente des mouvements possibles pour ce jeu. Néanmoins, nous notons parfois une bonne défense de notre approche en tant que second joueur qui la conduit vers la victoire.

Au cours du jeu, un temps de calcul supplémentaire est nécessaire pour notre approche afin d'effectuer la mise à jour des paramètres. Ceci la rend plus lente que l'approche utilisant l'heuristique brute. Toutefois, cette lenteur est comblée en moyenne par les bons résultats obtenus lors des tests.

Les connaissances que nous en tirons sont les suivantes :

- L'heuristique générique améliorée s'est révélée bien plus performante que l'heuristique de Shevchenko et assez productive face à l'heuristique de Chua Hock Chuan ;
- Les performances de l'heuristique améliorée et de l'heuristique brute varient en fonction de l'espace de recherche (taille du plateau) et du nombre de pièces à aligner n ;
- L'heuristique améliorée obtient de meilleurs résultats que l'heuristique brute lorsque l'espace de recherche est relativement grand et ceci en fonction du nombre de pièces à aligner ;
- Les paramètres fixes de l'heuristique ne sont pas les meilleurs dans toutes les situations pour chacun des jeux ;
- L'approche incluant la détermination par apprentissage automatique des paramètres de l'heuristique générique ne peut être déclarée totalement meilleure à l'heuristique générique aux paramètres fixés expérimentalement. Toutefois, elle surpasse à certains niveaux les performances de cette dernière.

Conclusion

Nous avons présenté dans ce chapitre les résultats des tests effectués sur chaque jeu. Les tests ont été effectués suivant plusieurs paramètres. Nous avons ensuite analysé les résultats afin de tirer des connaissances importantes.

Conclusion et perspectives

Dans ce travail, nous nous sommes intéressés aux jeux combinatoires de type « n-alignés » dans un environnement complètement observable, déterministe et discret. Pour ces jeux, l'espace de recherche devient énorme lorsque la taille du plateau augmente. Le choix d'une solution dans un espace géant de recherche est une tâche très compliquée. Récemment, ils ont été traités ensemble de manière à résoudre tous les jeux de cette catégorie avec la même solution. Il en est ressorti une heuristique générique qui utilise des paramètres fixés expérimentalement et qui peut être associée à des algorithmes de recherche.

Minimax est un algorithme de recherche assez réputé pour son succès dans le domaine des jeux et s'avère efficace quand il est associé à une fonction pouvant évaluer les nœuds de son arbre de recherche. L'algorithme d'élagage Alpha-Bêta qui en est une forme évoluée conduit à une économie de temps de calcul et mémoire, en renonçant à l'évaluation des sous-arbres dès que leur valeur devient inintéressante pour le calcul de la valeur associée aux nœuds père de ces sous-arbres. Il s'en sort plutôt bien avec les jeux de type « n-alignés » dont la taille du plateau peut être grande. Dans certaines situations, il faut quand même limiter la profondeur de l'arbre et utiliser une heuristique pour évaluer les nœuds.

Dans ce travail, nous avons proposé une méthode de détermination des paramètres de l'heuristique générique par apprentissage automatique pour l'obtention de meilleurs paramètres pour chaque jeu considéré. Elle se base sur le fonctionnement du Q-learning qui est un algorithme de contrôle TD « hors politique ». Ensuite, nous avons associé l'heuristique améliorée et l'heuristique brute à l'algorithme alpha-bêta pour faire une comparaison sur chaque jeu. Les résultats des tests ont montré qu'à divers niveaux l'heuristique améliorée est en moyenne plus performante même si ceci n'apparaît pas comme une évidence pour le jeu Connect four.

Une autre préoccupation de recherche consiste à déterminer les facteurs réels qui font que l'heuristique améliorée échoue parfois face à l'heuristique brute. Dans la même perspective, nous nous appesantirons aussi sur l'amélioration de l'approche proposée en abordant les traces d'éligibilité associées aux méthodes TD [7]. Aussi, pouvons nous apprendre automatiquement les stratégies gagnantes utilisées lors des parties jouées pour chaque jeu de type « n-alignés ».

Bibliographie

[1] Frédéric Don-de Dieu Gaël Roméo Aglin, *Heuristique générique pour les jeux de type «n-alignés»*, IFRI 2016, Mémoire pour l'obtention du Master en Informatique.

[2] Gauthier Picard, *Algorithme Minimax et élagage α-β*, SMA/G2I/ENS Mines Saint-Etienne, 2011, p.15,28,35

[3] Hoang Duc Viet, *Conception et Développement d'un moteur d'intelligence artificielle pour un jeu d'échecs multiplateformes*, Institut de la Francophonie pour l'Informatique 2014, p.16

[4] Tristan Cazenave, *Des Optimisations de l'Alpha-Béta*, Laboratoire d'Intelligence Artificielle Département Informatique, Université Paris, 2011 p.6,7.

[5] Stephen Simons, *Minimax theorems and their proofs*, Department of Mathematics, University of California, Santa Barbara, 1995, p.23.

[6] Samuel H. Fuller, John G. Gaschnig, *Analysis of the alpha-beta pruning algorithm*, Computer Science Department, Carnegie Mellon University, 1973, p.51.

[7] Richard S. Sutton and Andrew G. Barto, *Reinforcement Learning : An Introduction*, 2012,The MIT Press Cambridge, Massachusetts, chapter.3, 6, 7.

[8] Phillipe Preux, *Apprentissage par renforcement : Notes de cours*, GRAPPA, Version du 26 janvier 2008, p.10-13.

[9] Brit C. A. Milvang-Jensen, *Combinatorial Games, Theory and Applications*, Frebruary 2000.

[10] Yves Deville, *Adversarial search*, LINGI 2261 Artificial Intelligence, Université Catholique de Louvain, 2014.

[11] Bruno Bouzy, *Apprentissage par renforcement (3)*, 2010, p.12.

[12] Olivier Sigaud and Olivier Buffet *Markov Decision Processes in Artificial Intelligence*, 2010, chapter.2

Webographie

[13] Gomoku — Wikipédia https://fr.wikipedia.org/wiki/Gomoku consulté le 10 avril 2017.

[14] Gomoku - Jeux - Prise2tete jeux.prise2tete.fr/gomoku/gomoku.php consulté le 10 avril 2017.

[15] Morpion, renju - Le Comptoir des Jeux www.lecomptoirdesjeux.com/morpion-renju.htm consulté le 13 avril 2017.

[16] Gary Gabrel et Tom Braunlich http://jeuxstrategie1.free.fr/jeu_pente/regle.pdf consulté le 13 avril 2017.

[17] Pente et Renju, la pente fatale - Actualités - Tric Trac https://www.trictrac.net/actus/pente-renju-la-pente-fatale consulté le 26 avril 2017.

[18] Puissance 4 - Stratozor.com www.stratozor.com/puissance-4/ consulté le 27 avril 2017.

[19] Puissance 4 https://fr.wikipedia.org/wiki/Puissance_4 consulté le 27 avril 2017.

[20] Tic tac Toe : Règles, astuces et principes du Jeux et loisirs www.jeuxloisirs.net/Jeux-Papier-Crayon/Tictactoe/Tictactoe.html consulté le 28 avril 2017.

[21] Tic Tac Toe - Goobix fr.goobix.com/jeux-en-ligne/tic-tac-toe/ consulté le 30 avril 2017.

[22] Tic Tac Toe - pixabay https://pixabay.com/fr/tic-tac-toe-jeu-tick-tack-toe-jouer-150614/ consulté le 30 avril 2017.

[23] L'Atari-Go - La bibliographie du jeu de Go, bibliographie.jeudego.org/atari-go.pdf

[24] Mykola Shevchenko, *GOMOKU & Minimax-alphabeta search* https://github.com/nshevchenko/GomokuAlphabeta/blob/master/gomoku-minimax-alphabeta.pdf

[25] Chua Hock Chuan, course Java Game Programming, http://www3.ntu.edu.sg/home/ehchua/programming/java/JavaGame_TicTacToe_AI.html consulté le 2 mai 2017.

[26] What is Renju, sur http://www.renju.net/study/rules.php consulté le 26 avril 2017.

[27] Apprentissage automatique https://fr.wikipedia.org/wiki/Apprentissage_automatique consulté le 10 juin 2017.

[28] L'Apprentissage automatique https://www.univ-tlemcen.dz/~benmammar/IA2.pdf, consulté le 12 juin 2017.

[29] Temporal difference learning - Scholarpedia, www.scholarpedia.org/article/Temporal_difference_learning, consulté le 16 juin 2017.

[30] Conception informatique, www.conception-informatique.com/cycle-en-v/4-cycle-en-v, consulté le 2 juillet 2017.

[31] Cycle en V : un modèle conceptuel de gestion de projet, blog.ozitem.com/cycle-en-v/, consulté le 3 juillet 2017.

[32] Le cycle en V, https://www.focus-emploi.com/1555975/le-cycle-en-v, consulté le 7 juillet 2017.

[33] Introduction aux méthodes agiles et Scrum - L'Agiliste, www.agiliste.fr/introduction-methodes-agiles/, consulté le 11 juillet 2017.

[34] Gomoku, https://github.com/dtcristo/gomoku, consulté le 22 juin 2017.

[35] Chua Hock Chuan, course Java Game Programming, http://www3.ntu.edu.sg/home/ehchua/programming/java/JavaGame_TicTacToe_AI.html, consulté le 10 août 2017.