Projet IA41 : Teeko

Geoffrey NORO

Xavier BRUNI

Romain LALLEMAND

Robin JESSON



Table des matières

[Introduction 3](#_Toc516519418)

[Analyse du problème 4](#_Toc516519419)

[Min-Max 4](#_Toc516519420)

[Alpha-Bêta 5](#_Toc516519421)

[Méthode proposée 6](#_Toc516519422)

[Intelligence artificielle 6](#_Toc516519423)

[Interface graphique 7](#_Toc516519424)

[Descriptions de situations 8](#_Toc516519425)

[Résultats 9](#_Toc516519426)

[Difficultés rencontrées 10](#_Toc516519427)

[Amélioration et perspective 11](#_Toc516519428)

[Références 12](#_Toc516519429)

[Annexe 13](#_Toc516519430)

# Introduction

Dans le cadre de notre UV d’intelligence artificielle, nous avons eu à réaliser un jeu comprenant une IA. Notre choix s’est porté sur le jeu Teeko bien qu’il nous était d’abord inconnu.

# Analyse du problème

Notre problème consistait à réaliser une intelligence artificielle capable de jouer au jeu Teeko.

Pour rappel, les règles de ce jeu sont :

* Les joueurs un par un posent les quatre pions sur le plateau d'un carré de cinq cases par cinq.
* Ensuite chacun leur tour ils peuvent déplacer un pion sur une case adjacentes si elle est libre.
* Le but du jeu est d'aligner ses quatre pions, ou de forme un carré avant l'autre joueur.

Pour cela, il fallait pouvoir donner à notre intelligence artificielle la possibilité de détecter une combinaison gagnante, de lui implémenter les règles de déplacements, et particulièrement de la faire recherche une position gagnante.

Le jouer doit permettre à un humain de joueur contre une IA, ou à deux IA de joueur l’une contre l’autre.

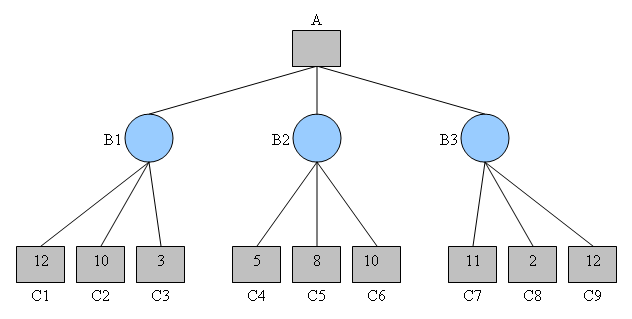
## Min-Max

L’algorithme est utile dans les cas ou il s’agit d’un jeu entre deux adversaires. Il permet de maximiser les coups d’une intelligence artificielle en prenant également les coups de l’adversaires.

L’algorithme Min-Max génère un arbre de nœuds, où chacun représente un état du plateau de jeu. La racine de cet arbre correspond à l’état du plateau juste avant que l’intelligence artificielle joue un coup. Elle devra choisir parmi les plateaux possibles successeurs du plateau-racine le déplacement qui maximiserait son coup. Pour pouvoir choisir le meilleur coup, l’algorithme a besoin d’une fonction d’évaluation qui permet de chiffrer un plateau selon le joueur.

Afin que l’intelligence artificielle soit plus précise dans son choix, un arbre plus grand peut-être généré. Mais à chaque fois, l’IA va regarder ses possibilités de coups, et la fois d’après elle regardera les possibilités qu’aura le joueur adverse, et ainsi de suite jusqu’à ce que ce soit un nœud gagnant. Elle choisira finalement le successeur de la racine le plus amène de l’amener à la victoire.

Mais étant donné que chaque nœud génère plusieurs successeurs, l’arbre peut devenir très grand et ainsi impacter le temps de calcul pour un coup. Ainsi l’arbre peut être limité jusqu’à une profondeur limitée pour pouvoir mieux contrôler le temps de calcul.



Sur l’exemple du dessous, la racine A va devoir choisir le maximum entre B1, B2 et B3. Pour cela, chacun des B va devoir instancier sa valeur à l’aide de ses propres successeurs en prenant le minimum des C1.

Ainsi B1 vaudra 3, B2 5 et B3 2. A devra après choisira par la suite B2 car c’est celui qui maximise son score en ayant pris en compte les coups adversaires.

Mais il y a toujours la possibilité d’avoir des temps de calcul long, donc l’élagage alpha-bêta permet de réduire encore plus les temps de calcul.

## Alpha-Bêta

# Méthode proposée

Afin d'implémenter l'IA, nous avons décidé au début de la faire en utilisant le langage Prolog. Concernant l'interface graphique, nous souhaitions la faire en Python en utilisant la librairie Pygame. Ainsi pour faire la jonction entre le code Python et le code Prolog, nous avons trouvé la librairie Python "Pyswip" qui permet de faire des appels de prédicats d'un fichier écrit en Prolog et de récupérer les résultats sous forme de Generator.

## Intelligence artificielle

L’intelligence artificielle est principalement écrite en langage Prolog. Elle est décomposée en plusieurs parties, dont la gestion des déplacements, le calcul de la valeur d’un plateau, et l’algorithme Min-Max.

Pour modéliser, nous avons choisi d’avoir une liste de vingt-cinq éléments. La valeur d’une case est soit 1 si la case est occupée par le joueur un, soit 2 pour le deuxième joueurs ou 0 si elle est vide.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 |

Exemple d’un plateau :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 0 | 0 |
| 0 | 0 | 2 | 0 | 0 |
| 0 | 2 | 0 | 1 | 0 |
| 0 | 0 | 2 | 0 | 0 |
| 0 | 2 | 0 | 0 | 0 |

La détection d’une combinaison est faite par le prédicat gagnant(+P1, +P2,+P3,+P4)qui vérifie si les quatre pions donnés en paramètres sont dans une combinaison gagnante. Pour ce faire, tous les cas sont testés :

* Ligne horizontale
* Ligne verticale
* Diagonale montante
* Diagonale descendante
* Carré

Dans chaque cas, il faut faire attention au premier pion. En effet, il suffit par exemple d’ajouter 1 entre chaque position pour voir si les quatre pions forment une ligne horizontale. Mais il faut faire attention au premier pion, afin que tout soit sur une même ligne et ne pas obtenir une combinaison du type {4,5,6,7} qui n’est pas gagnante tandis que {1,2,3,4} l’est.

Concernant les déplacements, il s’agissait de la partie la plus difficile à mettre en place car il fallait s’accorder avec règles du jeu qui sont d’avoir me droit de déplacer un pion sur une case adjacente si celle-ci est libre. Le travail a été décomposé ainsi en plusieurs prédicats :

* possibiliteDeplacerPion(+P, ?Res)
* deplacementsPossibles(+Joueur,+PlateauActuel,

-ListeDeplacementsPossibles)

* effectuerTousLesDeplacementsJoueur(+Joueur, +PlateauActuel, -PLateauxApres)

Les prédicats de la forme possibiliteDeplacerPion(+P, ?Res) permettent de retourner la liste des cases adjacentes d’une case donnée.

Le prédicat deplacementsPossibles prend dans ses paramètres le numéro du joueur et la plateau actuel obigatoirement. Il retorune la liste de toutes les cases sur lesquels un joueur aura la possibilité de se déplacer.

Pour finir, effectuerTousLesDeplacementsJoueur a en paramètres le numéro du joueur et le plateau actuel également, et une liste de plateaux est générée en fonction des déplacements possibles.

Concernant l’évaluation d’un plateau, il y a le prédicat evaluationJoueur. Il calcule la distance moyenne entre les quatre pions d’un joueur. Ainsi une grande distance moyenne signifie qu’un joueur ne pourra pas gagnant immédiatement.

Cette liste de plateaux ci-dessus sera donc utilisé dans l'algorithme Min-Max.

Par simplification, l’algorithme Min-Max a été écrit avec le langage Python, avec une classe IA et une classe Resultat. Cette dernière a pour attributs la valeur d’un plateau et le plateau qui lui est associé. La classe IA n’a qu’un numéro correspondant au numéro de joueur de l’intelligence artificielle. La méthode jouer(plateau)permet de retourner ce meilleur coup à jouer sous forme d’instance de Resultat. Elle effectue un appel à la méthode MaxValue()qui choisit le meilleur plateau parmi ceux qui seront retournés par la méthode MinValue().

Etant donné que dans notre cas la valeur la plus avantageuse doit être petite, la fonction MaxValue() prend le plateau ayant la plus petite valeur, et inversement pour MinValue().

## Interface graphique

# Descriptions de situations

Description détaillée de plusieurs situations du programme

-placement des pions au début :

-liste des déplacements :

Concernant le déplacements des pions, les prédicats utilisés sont regroupés dans le fichier *ia/deplacement.pl*. Tout d’abord, une liste de faits a été établie pour avoir les cases adjacentes d’une case.

Un prédicat permet de transférer le contenu d’une case vers une autre voulue : effectuerUnDeplacement. Pour cela deux indices de cases sont donnés, celui de la case de départ et celui de la case d’arrivée. A l’aide d’une variable initialisé à 1, on compare celle-ci avec les deux indices données, et si l’un des deux correspond, on effectue le changement adequat, c’est-à-dire libérer la case ou importer la valeur de la case à vider dans la nouvelle.

L’avant-dernière étape est de récupérer les plateaux après avoir effectuer les déplacements possibles d’une seule case passée en paramètre dans le prédicat selon un plateau originel passé également en paramètre.

Enfin, afin de récupérer la liste des déplacements des quatre cases d’un joueur, le prédicat précédent est appelé sur ces quatre cases-là.

Cette liste sera par la suite utilisé dans Min-Max. En effet c’est elle qui permet de générer les successeurs d’un état.

# Résultats

# Difficultés rencontrées

Pyswip

Puisque nous avons dès le départ souhaité faire l’intelligence artificielle en Prolog, et d’un autre côté réaliser une interface à l’aide de Python, il nous fallait utilisé une interface qui puisse à partir d’un programme écrit en Python pouvoir effectuer des appels de prédicat dans un autre fichier Prolog.

Dès le début nous avons trouvé la libraire Pylog. Son inconvénient est qu’il n’utilise pas de fichier Prolog. C’est plutôt une utilisation du moteur Prolog sur des objets Python.

Ensuite nous avons trouvé la librairie PySwip qui permet de faire des appels de prédicats sur un fichier Prolog :

prolog=Prolog()

prolog.consult(‘gagnant.pl’)

res=list(prolog.query(‘father(michael,X) ‘))

A l’aide d’un cast, on peut retourner les résulats des variables Prolog sous forme de liste.

L’inconvénient avec notre code, c’est que quand nous faisions un appel sur certains de nos prédicat, le code entrait dans une boucle infinie.

Nous avons ensuite trouvé la source de l’erreur. Cela provenait du code Prolog. Sur certains prédicats, une fois le résultat affiché, nous n’avions pas fait attention qu’il pouvait afficher plusieurs fois le même résultat en appuyant sur « ; ».

Nous avons finalement placé un cut au bon endroit afin que le prédicat ne soit plus appelé une fois le résultat trouvé. Cela a permis de régler ce problème de boucle infinie.

MinMax en prolog

Nous avons rencontré des difficultés à implémenter l’algorithme Min-Max en Prolog.

En effet, l’algorithme demande d’abord de boucler sur chacun des successeurs d’un nœud. Cela nous posait problème car nous ne savions pas vraiment comment faire, nous étions focalisés sur l’utilisation des prédicat foreach ou forall. Nous nous sommes par la suite rendu-compte que ce serait peut-être plus simple d’utiliser les propriétés des listes en Prolog, c’est-à-dire travailler sur la tête d’une liste, et rappeler le prédicat voulu sur le reste de la liste.

Cette façon résolvait alors notre problème de boucle « for » de l’algorithme vu en cours.

Mais un autre problèmes s’est soulevé : comment comparer la valeur de la tête de la liste après son évaluation avec les autres valeurs des autres successeurs ?

Après plusieurs essais, notamment à avoir essayé de créer une variable « temporaire » pour contenir la valeur la plus haute ou petite des successeurs précédents, nous n’arrivions pas à remplacer cette variable par une autre valeur et la passer au successeurs suivant.

Nous avons alors abandonné l’idée de faire l’algorithme en Prolog, et de plutôt le faire en Python car nous venions tout juste de réparer notre problème sur Pyswip.

# Amélioration et perspective

# [Références](http://www.citethisforme.com/harvard/source-type)

Pour les références si vous en avez : <http://www.citethisforme.com/harvard/source-type>

Code.google.com. (n.d.). *pyswip*. [online] Available at: https://code.google.com/archive/p/pyswip/ [Accessed 11 Jun. 2018].

Wikipédia. (2018). *Algorithme minimax*. [online] Available at: https://fr.wikipedia.org/wiki/Algorithme\_minimax [Accessed 11 Jun. 2018].

# Annexe