Ecole Nationale Supérieure d'Ingénieur pour l'Industrie et l'Entreprise

Rapport du projet d'IRP :

Minimax et ses variantes

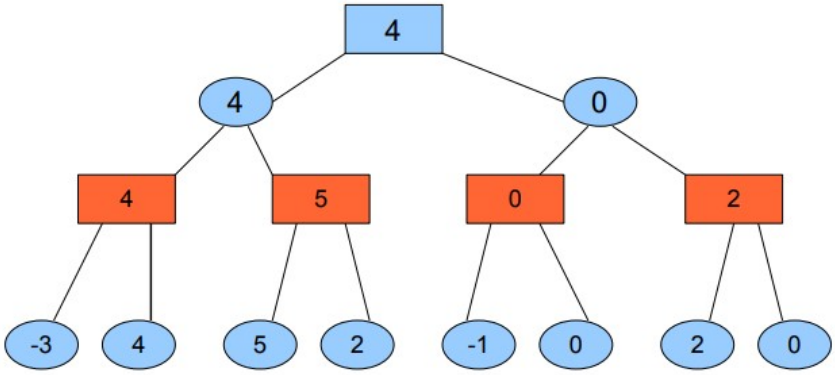
Élèves : **Jules LINARD & Maximilien SCHOLLAERT**
Professeurs à l'ENSIIE : **Catherine DUBOIS & Stefania DUMBRAVA**

Table des matières

I Echauffement

A Question 1 :

Voici la figure complétée avec les valeurs MiniMax :

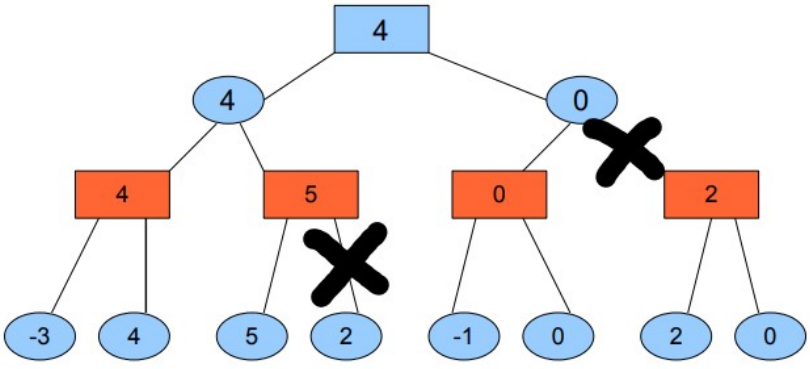


B Question 2 :

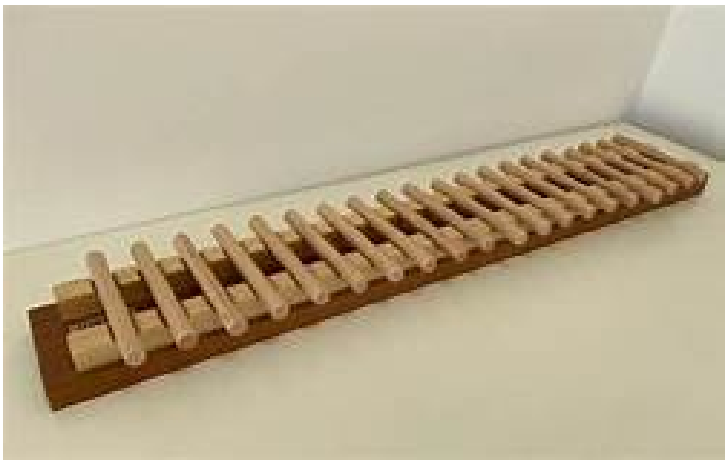
Ainsi, le meilleur choix pour le joueur Max est la valeur 4. Si on l'applique l'élaguage AlphaBeta, ce meilleur choix ne sera jamais remis en question.

C Question 3 :

Après élaguage des branches, on obtient le graphe suivant :



II Jeu de Nim



A Présentation des règles du jeu

Le jeu de Nim est un jeu à 2 joueurs se jouant avec un nombre N d'allumettes. Chacun leur tour, les deux joueurs doivent retirer 1, 2 ou 3 allumettes parmi les N allumettes. Le perdant est le joueur qui retire la dernière allumette.

B Implémentation

Afin d'implémenter le jeu de Nim, nous avons tout d'abord détaillé la liste des actions possibles à chaque état en suivant les règles énoncées précedemment :

- S'il ne reste qu'une allumette à tirer, le joueur est obligé de la tirer.
- S'il reste 2 allumettes à tirer, le joueur peut en tirer une, ou deux.
- S'il reste 3 allumettes ou plus, le joueur peut tirer 1, 2 ou 3 allumettes.

Pour calculer le nouvel état après qu'un joueur ait joué, il suffit de soustraire au nombre d'allumettes restantes le nombre d'allumettes que retire le joueur. Un état est considéré comme terminal s'il ne reste plus aucune allumette à retirer.

C Fonction d'évaluation

Pour calculer l'utilité de chaque noeud du graphe, nous avons choisi la fonction d'évaluation suivante :
Soit S l'état associé au noeud.

- Si S n'est pas un état terminal, la fonction d'évaluation renverra 0

- Si S est un état terminal, la fonction d'évaluation renverra 1 en cas de victoire du joueur humain, et -1 si la victoire est remportée par l'IA.

L'IA va alors chercher à minimiser la valeur de la fonction d'évaluation, et ainsi trouver les chemins dans le graphe lui permettant de gagner.

III Puissance 4

A Présentation des règles du jeu

Le puissance 4 est un jeu à 2 joueurs se jouant sur une grille de 6 lignes et 7 colonnes. Chacun leur tour, les deux joueurs placent un de leurs pions (jaunes ou rouges, représentés dans notre programme par des cercles et des croix) dans la grille. L'objectif est d'aligner 4 de ses pions verticalement, horizontalement ou en diagonale afin de remporter la partie.

B Implémentation

Pour implémenter le jeu Puissance 4, nous avons créé une classe éponyme qui implémente l'interface `Game`. Chaque état du jeu sera représenté par un couple (grille, joueur) qui représente la grille et le joueur dont c'est le tour. Les actions seront quant à elles représentées par des couples d'entiers symbolisant les coordonnées du pion à ajouter à la grille.

La classe aura pour membres de classe le nombre de lignes, de colonnes, la profondeur pour les algorithmes de recherche, ainsi que l'état initial correspondant au couple formé par la grille vide et le joueur commençant la partie.

La liste des actions possibles sera implémentée par une liste de couples d'entiers, qui représenteront le sommet d'une colonne ainsi que son numéro à condition que cette colonne ne soit pas remplie.

Pour obtenir le nouvel état du jeu après une action, il suffira d'ajouter à la grille le pion représenté par l'action choisie et à changer le joueur devant jouer.

Le jeu s'arrête lorsque la grille est remplie ou qu'un joueur a aligné 4 pions dans une direction. Pour vérifier l'alignement, nous avons implémenté la fonctions auxiliaire *victoire* qui à l'aide un point donné et d'un joueur donné, indique si le joueur a gagné. Afin de faciliter l'écriture de cette fonction et d'améliorer ses performances, nous ne regarderons que si le pion à trois pions au dessus de lui, ou à droite de lui, ou en haut à droite de lui, pour indiquer si le joueur a gagné ou non. Cette façon de penser s'explique par le fait qu'à chaque fois qu'un pion sera posé, la grille sera entièrement parcourue pour vérifier s'il y a un alignement de 4 pions, ce qui signifie qu'on parcourra toujours le sommet vérifiant les conditions d'alignement précédentes.

C Fonction d'évaluation

1 Choix de la fonction d'évaluation

Pour la fonction d'évaluation nous avons implémenté les fonctions auxiliaires *Utility_aux* et *Utility_aux_ajustee*.

La première fonction auxiliaire permet d'associer à chaque case de la grille une valeur "de base". Cette valeur correspond au nombre de façon possible d'aligner 4 pions en ayant un pion sur la case choisie, ce qui permettra d'orienter l'ordinateur vers les cases lui offrant le plus de possibilités d'alignement de points et donc de victoire. Pour calculer cette valeur nous avons quatre autres fonctions auxiliaires qui calculent le nombre de possibilités d'aligner quatre pions dans une direction donnée avec un pion sur la case voulue. Par exemple, pour la direction horizontale, cette valeur vaut : $nb\_cases\_a\_gauche + 1 + nb\_cases\_a\_droite - 4 + 1$, ce qui correspond au nombre de cases à gauche de la case, plus la case choisie, plus le nombre de cases à droite, moins le nombre de pions à aligner et plus 1. La fonction *Utility_aux* renvoie donc la somme de ces quatre sous-fonctions. Voici un tableau donnant la valeur "de base" de chaque case de la grille.

3	4	5	7	5	4	3
4	6	8	10	8	6	4
5	8	11	13	11	8	5
5	8	11	13	11	8	5
4	6	8	10	8	6	4
3	4	5	7	5	4	3

Valeurs de base des cases de la grille

La deuxième fonction auxiliaire *Utility_aux_ajustee* permet d'ajuster la valeur des cases en fonction des choix des joueurs. Pour cela, nous reprendrons la valeur donnée par *Utility_aux* multipliée par un petit coefficient (0,2 ici, nous le noterons α) dont nous expliquerons l'intérêt par la suite. Pour une case donnée, nous regarderons le nombre de pions "alignés" (c'est-à-dire le nombre de pion distant d'au plus trois cases avec la case initiale et n'ayant aucun pion adverse entre lui et la case initiale) pour chaque direction. Ce nombre permettra d'augmenter la valeur d'une case en fonction du nombre de point autour d'elle traduisant les chances de former un alignement de pions en posant le pion sur cette case. Cependant, ce nombre de pions "alignés" dans une direction sera mis à 0 si jamais les pions adverses ne permettent pas de faire un alignement de 4 pions afin de

ne pas orienter l'ordinateur dans des choix inutiles. La fonction *Utility_ aux_ ajustee* renverra donc $\alpha * Utility\_aux + \sum_{d \in Direction} nb\_voisins\_aligne(d)^2$. Le coefficient α permet de chosir la case qui offre le plus de possibilités si jamais plusieurs cases ont le même nombre de voisins. Les nombres de voisins sont mis au carrés afin d'augmenter plus rapidement l'utilité d'une case avec son nombre de voisins.

Enfin, la grille étanr constituée de -1 pour les pions de l'ordinateur, 1 pour les pions du joueur, et de 0 pour les cases vides, l'utilité est donc la somme des cases de la grille multipliée par leur valeur donnée par *Utility_ aux_ ajustee*.

2 Perspectives d'amélioration de la fonction d'évaluation

Nous avons choisi d'implémenter une heuristique "défensive" afin de forcer l'ordinateur a défendre les coups gagnant du joueur et à ne pas privilégier ses coups gagnants sans prendre en considération les coups de joueur, même si ces derniers sont gagnants. Le problème de notre heuristique est que l'ordinateur va en permanence chercher à faire dans le pire des cas une égalité, sans pour autant chercher à gagner, ce qui implique que l'ordinateur ne va pas forcément choisir un coup qui entraînera sa victoire lorsqu'il peut gagner. Il faudrait donc améliorer l'heuristique sur ce point.

IV Algorithmes de recherche

Nous noterons que les deux jeux implémentés dans ce projet utilisent l'algorithme Alpha-Beta avec une profondeur limitée afin de gagner en efficacité.

A Minimax avec profondeur limitée

L'algorithme Minimax avec profondeur limitée reprend l'algorithme Minimax en y rajoutant un paramètre. Il s'agit d'un entier représentant la profondeur jusqu'à laquelle l'IA va parcourir le graphe lui permettant de déterminer son prochain coup. Plus cet entier est petit, moins l'IA a de chances de prendre la bonne décision concernant le reste de la partie. Si cet entier est suffisament grand et est supérieur à la taille du plus long chemin du graphe, on retrouve l'algorithme Minimax "classique".

B AlphaBeta

L'algorithme AlphaBeta fonctionne comme l'algorithme Minimax en principe : il cherche a maximiser et minimiser successivement. Cependant, le nombre de noeud explorés est nettement inférieur à celui de Minimax puisque pour chaque sommet l'algorithme AlphaBeta va calculer des bornes pour la valeur d'utilité et élaguer les branches dont on sait qu'elles n'amélioreront pas la solution (voir **Readme.md** pour accéder aux comparaisons des algorithmes).

C AlphaBeta avec profondeur limitée

De même, l'algorithme AlphaBeta avec profondeur limitée est identique à l'algorithme AlphaBeta "classique", mais rajoute en paramètre un entier indiquant la profondeur jusqu'à laquelle l'IA va parcourir le graphe lui permettant de déterminer son prochain coup.

D AlphaBeta avec profondeur limitée et mémoire

Nous avons tenté d'implémenter cet algorithme mais il ne fonctionne pas. Comme il était demandé, nous avons utilisé la fonction de hachage de Zobrist comme clé pour stocker les valeurs d'alpha et de beta. Cependant nous n'avons pas su corriger les problèmes que nous avons rencontré.

1 Modification de la classe `Game`

Nous avons rapidement modifié l'interface Game afin qu'elle puisse prendre en compte le hashage de Zobrist. Les fonctions rajoutées ne seront cependant pas implémentées dans le jeu de Nim, mais uniquement utilisées lors du puissance 4.

2 Modification de la classe `Puissance4`

Afin de prendre en compte ce nouvel hashage, nous avons donc implémenté les 3 nouvelles méthodes de l'interface Game. Initialement, on génère un hashcode de manière aléatoire pour notre jeu.
La première méthode set_hashcode_joueurMax est une méthode permettant de générer une "grille de hashcode" pour le joueur Max. Cela signifie que l'on va associer à chaque case un hashcode que l'on utilisera dans la méthode ZobristHashState pour réaliser le Ou-Exclusif. De même, la méthode set_hashcode_joueurMin génère une "grille de hashcode" pour le joueur Min.
La méthode ZobristHashState permet de calculer le nouvel hashcode de la partie. Lorsqu'un joueur va jouer dans une case, la fonction va réaliser un Ou-Exclusif entre l'ancien hashcode de la partie et le hashcode correspondant à la case jouée (en adéquation avec le joueur) afin de trouver le nouvel hashcode de la partie.

V Classes annexes

A Paire

La classe Paire représente simplement un couple de deux valeurs, quelque soit leurs types. Elle possède notamment des getters et des setters. Elle est notamment utilisée pour le jeu puissance 4.

B Triple

La classe Triple représente simplement un triplet de trois valeurs, quelque soit leurs types. Elle possède notamment des getters et des setters. Elle est notamment utilisée pour stocker les triplets

(alpha, beta, profondeur) dans les tables de hachage.

VI Conclusion

En conclusion, nous avons implémenté le jeu de Nim et le jeu Puissance 4, ainsi que les algorithmes de recherche Minimax et AlphaBeta et leur variante avec profondeur limitée, mais nous n'avons pas cependant pas réussi à implémenter l'algorithme AlphaBeta avec mémoire. Nous avons pour chaque jeu une heuristique qui permet à l'ordinateur de jouer correctement aux jeu, même si celle du jeu puissance 4 semble avoir des difficultés à choisir les actions lui permettant de gagner.