TP/Projet IRP - MiniMax et ses variantes - S4

L'objectif est ici d'implémenter MiniMax, l'élagage AlphaBeta et d'autres variantes pour quelques jeux simples. Le code de base est fourni en Java et en OCaml. Vous choisirez un de ces 2 langages.

Le sujet à la section 2 est à travailler en binôme et à rendre le 22 mars 2022 au plus tard sur le site exam.ensiie.fr (nom du dépôt étant irp_jeux2022. Vous devez déposer une archive (au format tar ou zip impérativement) dont le nom contiendra les noms des étudiants composant le binôme, contenant un rapport au format pdf expliquant en particulier vos choix d'implementation (en particulier les fonctions d'évaluation) et les tests réalisés, le code, un makefile et un *readme*.
Tout rendu ne contenant pas de makefile permettant d'installer et exécuter votre code facilement ne sera pas examiné. Idem pour le readme qui doit clairement indiquer comment utiliser le code et comment jouer contre l'ordinateur.

1 Echauffement

1. Donnez la valeur MiniMax de chacun des noeuds de l'arbre de décision de la figure 1, sachant que les valeurs aux feuilles sont les valeurs d'utilité.
2. Quel est le meilleur choix pour le joueur Max ? Si on applique l'élagage AlphaBeta, le meilleur choix peut-il être remis en question ?
3. Quelles branches seront-elles élaguées si on applique l'applique l'algorithme AlphaBeta ?

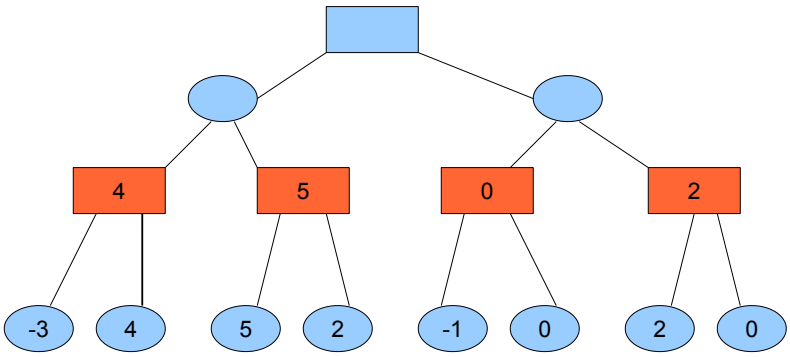

FIGURE 1 – Quizz

2 Implémentons et jouons

On commence par définir un jeu comme indiqué en cours. On propose l'interface générique `Game` définie dans le fichier `Game.java` ainsi que le type de module `Game` défini dans le fichier `tpjeux.ml`

Le joueur Max est représenté par le booléen true, le joueur Min par false.

— `getInitialState` renvoie l'état initial du jeu,

— `getActions` renvoie la liste des actions possibles pour un état donné,
— `getResult` renvoie l'état résultant de l'exécution d'une action donnée sur un état donné,
— `isTerminal` vérifie si un état est un état terminal,
— `getUtility` retourne la valeur d'utilité d'un état donné
— `getDepth` retourne le profondeur maximale dans le cas d'une recherche à profondeur limitée.
Les algorithmes de recherche implémenteront l'interface `Search`.

L'algorithme MiniMax vu en cours est éclaté en deux fonctions mutuellement récursives `MinValue` et `MaxValue`. La fonction `MakeDecision` en plus de calculer les valeurs MinMax de chaque noeud, calcule l'action à choisir pour le noeud Max qui maximise sa valeur MinMax. Ces 3 fonctions sont implémentées de manière générique dans le fichier `MinimaxSearch.java` ou dans le foncteur `MinimaxSearch` en OCaml. Elles gardent trace du nombre de noeuds explorés.

Description du jeu de Nim

. On dispose d'un tas d'allumettes (le nombre d'allumettes sera un paramètre du jeu). Deux joueurs choisissent, chacun à leur tour (Max commence) 1, 2 ou 3 allumettes dans la réserve. Le joueur qui retire la dernière allumette de la réserve perd la partie.

Ici, l'état du jeu est constitué d'un entier qui représente le nombre d'allumettes restantes. Les actions sont représentées par des valeurs entières, c'est-à-dire le nombre d'allumettes retirées par le joueur (1, 2 ou 3).

Travail à faire Représenter le jeu par une classe Java ou un module OCaml qui implémente l'interface `Game`. En Java, vous fournirez un constructeur à deux paramètres (nombre initial d'allumettes et profondeur de l'arbre de jeux, non utilisé dans cette première approche).
Vous écrirez un programme de test qui appelle la fonction `MakeDecision` sur l'état initial permattent d'obtenir la meilleure action pour Max ainsi que le nombre de noeuds explorés.

Exécutez ce programme sur différents états initiaux. Retrouvez en particulier les résultats du jeu de Nim utilisé en cours.

Implémenter une deuxième classe de test ou fonction de test qui permet de jouer avec l'ordinateur qui, lorsqu'il doit jouer, appelle la fonction de décision MiniMax pour choisir la meilleure action. Le choix de l'Humain sera lu. Le squelette de `NimTest` est donné dans l'archive.

MiniMax avec profondeur limitée

. **Travail à faire** Implémenter dans une nouvelle classe générique ou un nouveau foncteur, l'algorithme MiniMax avec profondeur limitée. Dans ce cas on supposera que la fonction utility a été fournie de manière à calculer une valeur heuristique si elle est appelée sur un état qui n'est pas un état de fin de jeu.

Travail à faire sur le jeu de Nim. Ajouter dans la classe de Test `NimTest` un appel à la nouvelle fonction de décision de manière à pouvoir comparer le nombre de noeuds explorés. Modifier le jeu contre l'ordinateur pour qu'il appelle aussi la nouvelle fonction de décision (toujours à des fins de comparaison).

Élagage AlphaBeta

. **Travail à faire**. Implémenter dans une nouvelle classe générique ou un nouveau foncteur, l'algorithme AlphaBeta. Ajouter dans `NimTest` un appel à la nouvelle fonction de décision de manière à pouvoir comparer le nombre de noeuds explorés. Modifier le jeu contre l'ordinateur pour qu'il appelle aussi la nouvelle fonction de décision (toujours à des fins de comparaison).

Élagage AlphaBeta avec profondeur limitée

. **Travail à faire**. Implémenter dans une nouvelle classe générique ou un nouveau foncteur, l'algorithme AlphaBeta ave profondeur limitée. Ajouter dans l `NimTest` un appel à la nouvelle fonction de décision de manière à pouvoir comparer le nombre de noeuds explorés. Modifier le jeu contre l'ordinateur pour qu'il appelle aussi la nouvelle fonction de décision (toujours à des fins de comparaison).

Puissance 4

Sur une grille de 7 colonnes de 6 cases, chaque joueur met à tour de rôle un pion dans une colonne, celui-ci descend en bas de la colonne. Celui qui en aligne 4 a gagné.

Travail à faire. Implémenter le jeu comme une implémentation de l'interface Game. Puis réaliser un jeu contre l'ordinateur, comme précédemment en utilisant les différentes fonctions de décision pour l'ordinateur (comparer les nombres de noeuds explorés). Vous essayerez de trouver une bonne fonction d'évaluation pour les feuilles de l'arbre.

Tables de transposition et alphabeta

Le principe des tables de transposition est le suivant : pour chaque position rencontrée dans la recherche, l'algorithme minimax retourne une évaluation. Il s'agit de conserver dans une table la valeur de chacune des positions rencontrées de façon à pouvoir réutiliser directement cette valeur lors des recherches suivantes. Les informations stockées pour une position sont son évaluation, la meilleure action et la distance à la position terminale. Par exemple, sur l'arbre de décision de la figure 2, les informations pour la position P1 sont 20 (pour la valeur MiniMax), Gauche (pour la meilleure action) et 2 pour la distance. On gardera plutôt pour le noeud les valeurs de alpha et beta.

Supposons que dans une recherche future l'ordinateur rencontre à nouveau la position P1. Si la distance à la racine de P1 est plus grande dans la table de transposition que dans la recherche actuelle, l'ordinateur ne poursuivra pas l'évaluation : il se contentera de recopier la valeur stockée dans la table de transposition. En revanche, si la distance à la racine est plus petite dans la table de transposition que dans l'évaluation courante, le programme terminera normalement l'évaluation de la position et remplacera l'ancienne évaluation de la table de transposition par celle qu'il aura calculée, puis stockera la nouvelle valeur de la distance à la racine.

La table de transposition est représentée par une table de hash. Il est classique dans le contexte des jeux d'utiliser la fonction de hachage appelée *Zobrist hashing* (voir `https://en.wikipedia.org/wiki/Zobrist_hashing`)

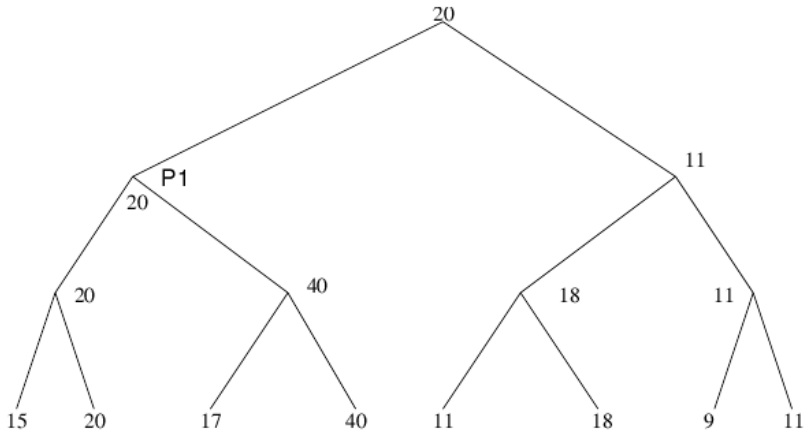

FIGURE 2 – Informations conservées dans la table de transposition

On va maintenant améliorer l'élagage AlphaBeta en utilisant une table de transposition (avec la fonction de hachage de Zobrist) pour accélérer la recherche. On commence par vérifier si le noeud est dans la table.

Travail à faire. Implémenter l'algorithme AlphaBeta avec mémoire (table de transposition). Appliquer ce dernier algorithme pour écrire une fonction de jeu contre l'ordinateur avec le jeu de votre choix.