MinMax et α-β

```
1 function ALPHA_BETA(s) returns an action
2     v ← MAX_VALUE(s,-∞,+∞)
3     return a ∈ actions(s) with value v
```

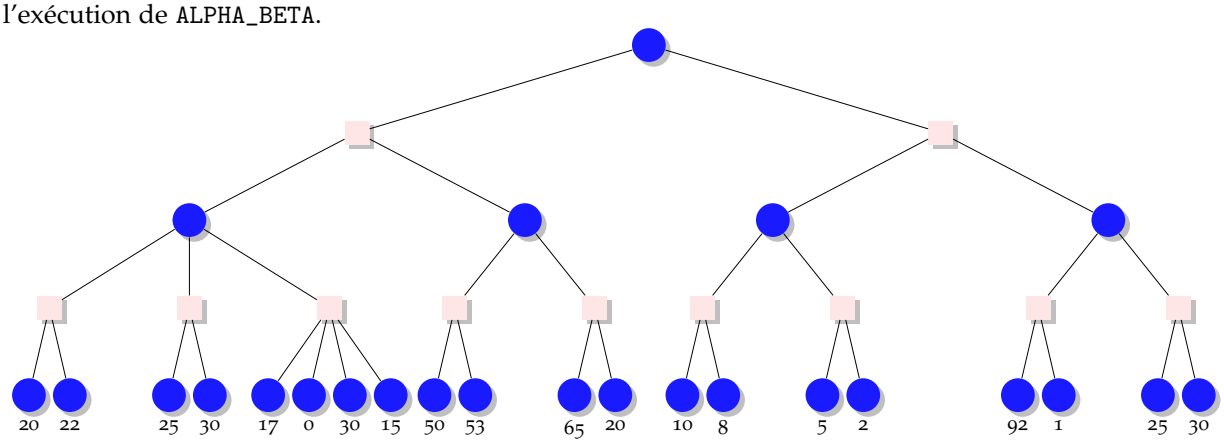
- `result(s,a)` est l'état atteint lorsqu'on prend l'action `a` dans l'état `s`
- α valeur de la meilleure (plus haute) valeur jusqu'ici pour `max`
- β valeur de la meilleure (plus basse) valeur jusqu'ici pour `min`

```
1 function MIN_VALUE(s,α,β) returns a utility value
2     if terminal?(s) then return utility(s)
3     v ← +∞
4     for each a ∈ actions(s) do
5         v ← min{v,MAX_VALUE(result(a,s),α,β)}
6         if v ≤ α then return v
7         β ← min{β,v}
8     return v
```

```
1 function MAX_VALUE(s,α,β) returns a utility value
2     if terminal?(s) then return utility(s)
3     v ← −∞
4     for each a ∈ actions(s) do
5         v ← max{v, MIN_VALUE(result(a,s),α,β)}
6         if v ≥ β then return v
7         α ← max{α,v}
8     return v
```

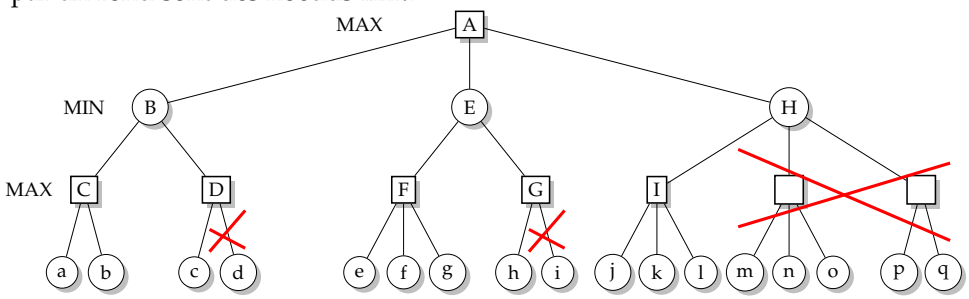
Exercice 1

Appliquez les algorithmes `MINIMAX` et `ALPHA_BETA` à l'arbre de jeu ci-dessous où les cercles sont des noeuds `MAX`, les carrés sont des noeuds `MIN`. Montrez les valeurs de α, β et v pour chaque noeud lors de l'exécution de `ALPHA_BETA`.



Exercice 2

Considérons l'arbre de jeu suivant où les noeuds représentés par un carré sont des noeuds MAX et ceux représentés par un rond sont des noeuds MIN.



- Donnez des valeurs aux feuilles de sorte que l'algorithme α-β coupe *exactement* les branches indiquées.

• Appliquez l'algorithme sur l'arbre avec vos valeurs.

Exercice Implémentation

Le but de ce TD est d'implémenter un algorithme pour jouer un jeu contre un adversaire à l'aide des algorithmes de `MinMax` et d'élagage α-β.

On considère le jeu suivant à deux joueurs qui se joue sur une grille $n \times n$. Un joueur est appelé `ligne` et l'autre est appelé `colonne`. Chaque joueur possède un tas de dominos qui mesurent 2×1 unité. Chacun son tour, `ligne` et `colonne` déposent *un* domino sur la grille :

— `ligne` peut déposer un domino horizontalement, couvrant deux cases de la grille.

— `colonne` peut quant à lui déposer un domino verticalement, couvrant deux cases de la grille.

Evidemment, on ne peut placer un domino que sur deux cases vides. Au fur et à mesure du jeu, la grille se remplit et le perdant est le premier joueur qui ne peut pas poser son domino. Dans l'exemple de la Figure 1, on a $n = 8$, `ligne` a commencé à jouer et la partie n'est pas terminée.

Dans ce jeu, le joueur qui ouvre la partie possède peut-être un avantage sur le joueur qui joue en second. Il faudra donc faire quelques tests en changeant de rôle.

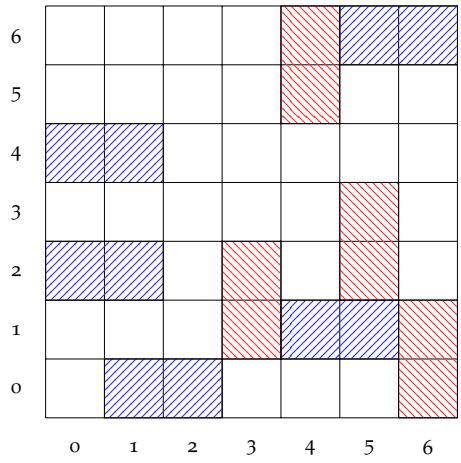


FIGURE 1 – Exemple du jeu pour $n = 7$, la partie est en cours.

Code fourni

On fournit un code `Java` pour jouer une partie. Le code contient l'implémentation d'un serveur va gérer le déroulement de la partie en demandant au joueur de placer un jeton chacun leur tour. Chaque case de la grille est identifiée par ses coordonnées (i, j) où $i \in \{0, n-1\}$ et $j \in \{0, n-1\}$. On utilisera la classe `Case` pour représentée un case. Un domino sera représenté par une paire de `Case` (qui en principe doivent être consécutives). Vous devez implémenter des joueurs qui vont suivre l'interface `Joueur`. Cette interface contient les méthodes suivantes :

— `setRole(int direction)` indique le rôle du joueur : direction prendra soit la valeur `Jeu.LIGNE` (respectivement `Jeu.COLONNE`) pour indiquer que le joueur jouera en tant que joueur `ligne` (respectivement `colonne`).

— `String getName()` retourne le nom du joueur.

— **void** `reset()` : cette méthode sera appelée avant le début de chaque nouvelle partie.

— `Domino joue()` : cette méthode sera appelée pour demander de placer un domino. Avant cet appel, l'interface vérifiera que vous avez au moins un coup possible à jouer. Pour jouer, il vous

suffira de retourner un objet `Domino`.

— **void** `update(Domino l)` : Après le dépôt d'un domino par l'un des deux joueur, le serveur communiquera aux *deux* joueurs le domino qui a été placé grâce à un appel à la méthode `update`.

On vous fournit également le code d'un agent qui joue de manière aléatoire (`JoueurAleatoire`) et le code d'un agent qui vous permet de jouer en entrant les coordonnées au clavier (`JoueurClavier`). Pour ce joueur, entrez les coordonnées d'une case en suivant le format suivant : vous entrerez sur une même ligne les coordonnées d'une case, les coordonnées étant séparées par un espace `i j`. Si vous êtes un joueur `ligne`, le domino placé occupera les cases `(i,j)` et `(i+1,j)`. Si vous êtes le joueur `colonne`, le domino occupera les cases `(i,j)` et `(i,j+1)`.

Travail

Le serveur de jeu n'est là que pour réaliser une partie et vérifier que les coups sont légaux. Pour coder vos joueurs, c'est à vous de choisir votre représentation du jeu et des coups. Evidemment, vous pouvez vous inspirer de ce qui est fait pour la classe `Jeu` (où le plateau est simplement représenté à l'aider d'un tableau `boolean[][] libre`), voir utiliser une nouvelle instance du jeu. Vous pouvez aussi utiliser votre propre représentation si cela vous arrange.

Pour les tests, vous pouvez jouer avec un jeu de petite taille. Pour effectuer les tests finaux, on se concentrera sur un jeu de taille 8×8.

1. Implémentez une fonction d'évaluation qui retourne pour chaque état du jeu un nombre. Vous êtes libres de choisir le type du nombre (`int`, `float`, `double`, etc...). Utilisez votre intuition pour évaluer l'état du jeu. Pour cela, vous pouvez jouer quelques parties à la main contre le joueur aléatoire.
2. Etant donné la taille de l'arbre de jeu (sur un jeu de taille 8×8 ou plus), vous risquez de ne pas avoir les ressources suffisantes pour exécuter l'algorithme `MinMax` sur l'arbre de jeu en entier. Vous pouvez donc remplacer le test pour savoir si vous avez atteint un état final du jeu par un test (`cut-off tes`) qui arrête une recherche plus profonde. Vous utiliserez alors votre fonction d'évaluation pour estimer la qualité de l'état dans lequel vous vous êtes arrêtés.
   Implémenter un joueur qui utilise une variante de l'algorithme `MinMax` avec votre fonction d'évaluation.
   Testez votre algorithme en jouant à la main contre lui. Est-ce que la stratégie de votre algorithme correspond à votre intuition pour jouer ce jeu ?
3. Vous pouvez utiliser la même stratégie pour l'algorithme α-β.
   Implémentez un joueur qui utilise l'algorithme `élagage` α-β et votre fonction d'évaluation.
4. Faite une étude pour évaluer le gain de l'algorithme α-β par rapport à `MinMax` en utilisant le même `cut-off test` et la même fonction d'évaluation. Par exemple, vous pouvez évaluer combien de noeuds ne sont pas visités par α-β par rapport à `Minimax`. Dites aussi comment cette étude peut vous aider à modifier le `cut-off test` de α-β par rapport à celui de `MinMax`.