CHAPITRE V
Introduction à l'intelligence artificielle

La définition d'intelligence artificielle n'a pas vraiment de rapport avec la création d'une entité douée d'intelligence supérieure. En fait, ça consiste a essayer de faire faire des tache plutôt simple et instinctive pour un humain, par une machine, comme repérer un son, une image globalement, une caricature une écriture manuscrite et de les associer aux grande capacité de mémoire et de calcul. Evidement certaine de ces taches accomplis par des humains pourront être effectuer par des machine (comme traducteur, juriste...)

1 L'algorithme des k plus proches voisins

1.1 Introduction

Pour reconnaître une plaque minéralogique, il est nécessaire de savoir reconnaître un chiffre entre 0 et 9 dans une image. Une idée possible consiste à partir d'une banque d'images représentant des chiffres (on sait quels chiffres sont représentés) puis on cherche alors les k images les plus proches (au sens d'une certaine distance) de l'image x dans la banque. On parle ici d'apprentissage surpervisé, car on part d'une banque de données où la classe d'appartenance est connue.

1.2 Algorithme

L'algorithme des k plus proches voisins est un algorithme d'apprentissage supervisé : il est nécessaire d'avoir des données labellisées, $(y_i, \vec{x_i})$, ou y_i est l'étiquette, "l'identité" et $\vec{x_i}$ les données numérique le caractérisant. À partir d'un ensemble de données labellisées, il sera possible de classer (déterminer le label) d'une nouvelle donnée.

Le principe de l'algorithme de k-plus proches voisins est le suivant :

On dispose d'un ensemble de donnée E, d'une donnée extérieure à étudier et d'une distance d.

— On calcule les distances entre la donnée u et chaque donnée appartenant à E à l'aide de la fonction d.

— On retient les k données du jeu de données E les plus proches de u.

— On attribue à u la classe qui est la plus fréquente parmi les k données les plus proches.

Par exemple on peut choisir comme distance entre deux images en noir et blanc I et J :

$$d(I, J) = \sum_{i=0}^{n} \sum_{i=0}^{j} \mid I_{i;j} - J_{i,j} \mid$$

où $I_{i,j}$ est la couleur du pixel de coordonnée (i, j) (0 ou 1).

Exercice 1.1 *Programmer une fonction python qui prend comme paramètres deux tableau numpy composés de* 0 *(noir) et de* 1 *(blancs), représentant les deux image et renvoyant la distance énoncée ci-dessus.*

1.3 Exemple

Un exemple basique :

Pour simplifier, nous allons implémenter l'algorithme dans le plan, avec la distance euclidienne. On suppose de plus qu'il n'y a que deux classes d'objets, les points rouges et les points bleus. L'ensemble d'apprentissage (la base) sera constituée d'une liste de couples (p, e) où :

— p est un point du plan euclidien, représenté comme un couple de valeurs numériques ;

— e est une étiquette, pouvant valoir 'r' ou 'b', pour rouge et bleu.

L'algorithme prendra également en entrée un point du plan dont on veut décider s'il est rouge ou bleu, et un entier k supposé impair : on attribuera la couleur associée à la couleur majoritaire parmi les k plus proches voisins, comme k est impair on n'a pas à se préoccuper des cas d'égalité.

```
import numpy as np
def d(p1,p2):
  return np.sqrt(np.vdot(p1-p2,p1-p2))
```

```
def couleur(X,p,k):
  """ calcule la couleur de p en prenant la couleur majoritaire
  parmi ses k plus proches voisins """
  L = [(d(p,c[0]),c[1]) for c in X]
  L.sort()
  nb = 0
  for i in range(k):
      if L[i][1]=='r':
          nb+=1
      else:
          nb-=1
  if nb>0:
         return 'r'
  else:
         return 'b'
```

Pour aider à choisir un bon paramètre k, il est courant de scinder en deux l'ensemble d'apprentissage. Imaginons que l'on ait encore un ensemble de points Y issus des mêmes distributions (on connaît leur couleur rouge ou bleu), que l'on avait écartés dans l'analyse précédente. À k fixé, on peut voir quelle couleur est attribuée à chacun via l'algorithme des k plus proches voisins, et construire une matrice 2×2 appelée matrice de confusion, de la forme

$$\begin{pmatrix} RR & RB \\ BR & BB \end{pmatrix}$$

où :

— RR (resp. BB) est le nombre de points de Y qui sont rouges (resp. bleus) et étiquetés comme tel par l'algorithme.

— RB (resp. BR) est le nombre de points de Y qui sont rouges (resp. bleus) et étiquetés en bleu (resp. rouge) par l'algorithme.

2 Algorithme des k-moyenne

2.1 Introduction

À l'oeil, il est facile de voir qu'un nuage de points est composé de 3 morceaux, auquel on a donné des couleurs distinctes. L'appartenance d'un point à l'amas orange ou à l'amas bleu peut se discuter pour les points situés à la frontière, l'oeil humain est très bon pour répartir instinctivement les points en trois morceaux. Le but de cette section est de décrire un algorithme prenant en entrée un tel nuage de points dans un espace euclidien, un entier k et renvoyant une partition du nuage en k sous-parties « proches ». On parle ici d'apprentissage non supervisé car contrairement à la section précédente, on n'a pas d'ensemble d'apprentissage déja étiqueté. Concrètement, le problème est le suivant :

2.2 Algorithme

Soit X un ensemble fini de points d'un espace euclidien, et k un entier inferieur au cardinal de X. L'algorithme consiste à trouver une partition de X en sous-ensembles $X_0, \ldots, X_{k-1}$ non vides qui minimise la quantité $\sum_{i=0}^{k-1} \sum_{x \in X_i} \| x - m_i \|^2$, avec m_i le barycentre des points de X_i.

Choisir k points qui représentent la position moyenne des partitions $m_1^{(1)}, \ldots, m_k^{(1)}$ initiales (au hasard par exemple) ;

Répéter jusqu'à ce qu'il y ait convergence :

— affecter chaque observation à la partition la plus proche :

$$S_i^{(t)} = \left\{ \mathbf{x}_j : \left\| \mathbf{x}_j - \mathbf{m}_i^{(t)} \right\| \leq \left\| \mathbf{x}_j - \mathbf{m}_s^{(t)} \right\| \ \forall\ s = 1, \ldots, k \right\},$$

— mettre à jour la moyenne de chaque cluster :

$$\mathbf{m}_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{\mathbf{x}_j \in S_i^{(t)}} \mathbf{x}_j.$$

2.3 Exemple

On travaille ici avec des tableaux numpy représentant des vecteurs dans R^k pour un certain k. On appellera la commande produit scalaire `np.vdot()` du module numpy.

```
import random as rd
import numpy as np
def dist(x,y):
     return np.sqrt(np.vdot(x-y,x-y))

def barycentre(X,xi):
    """ X ensemble de points, xi liste d'indices de X.
    Renvoie le barycentre des X[k] pour k dans xi """
   s=0
   for k in xi:
        s=s+X[k]
   return s/len(xi)
```

Pour l'initialisation de l'algorithme, on utilisera par exemple la fonction suivante, permettant de placer les entiers de $[|0, n-1|]$ dans k sous-ensembles disjoints, non vides (représentés comme des listes). Pour faire ceci, on mélange aléatoirement la liste constituée des entiers de $[|0, n-1|]$. Les k premiers éléments obtenus sont les indices des points qu'on ajoute à $X_1, \ldots, X_k$. Pour les éléments suivants, on choisit aléatoirement dans quel X_j on les place.

```
def initialisation(n,k):
   """ initialisation de l'algorithme :
   renvoie une partition de [0,n-1] en k morceaux non vides."""
   assert n>k
   part=[[] for i in range(k)]
   I=rd.randint(0,k,n)
   for j in range(n):
        part[I[j]].append(j)
   return part
```

Voici enfin une fonction prenant en entrée un point p et une liste d'autres points M (dans l'idée, des barycentres), et renvoyant l'indice du point le plus proche de p dans M.

```
def plus_proche(p,M):
    i, d = 0, dist(p,M[0])
    for j in range(1,len(M)):
          if dist(p,M[j])<d:
                 i, d = j, dist(p,M[j])
    return i
```

On a maintenant ce qu'il faut pour écrire l'algorithme des k-moyennes.

```
def k_moyennes(X,k):
   n=len(X)
   assert n>=k
   part=initialisation(n,k)
   while True:
            M = [barycentre(X,xi) for xi in part]
            part2 = [[] for _ in range(k)]
            for i in range(n):
                    part2[plus_proche(X[i],M)].append(i)
            if part == part2:
                    return part
           else:
                  part = part2
```

2.4 Applications

L'algorithme des k-moyennes peut être utilisé pour réduire le nombre de couleurs (en) d'une image sans que cela ne nuise trop à sa qualité. L'espace de couleur du système RGB de codage des couleurs consiste à représenter une couleur par trois nombres entiers compris entre 0 et 255 qui représentent les intensités respectives du rouge, du vert et du bleu dans la couleur à afficher. Cela donne 2563 couleurs possibles, soit environ 16 millions. L'œil humain non-entraîné peine à distinguer autant de couleurs et il est donc possible de remplacer deux couleurs proches par une seule sans grande perte de qualité ; ce peut être utile à des fins de compression ou pour permettre l'affichage optimal d'une image sur un écran

ou une imprimante n'offrant pas une grande variété de couleurs. L'algorithme des k-moyennes permet de trouver, pour chaque pixel d'une image, parmi une liste de k couleurs définies, la couleur qui est la plus proche.

L'algorithme de réduction d'image nécessite plusieurs itérations. D'abord on initialise en choisissant une liste de k couleurs, et on crée une nouvelle image en remplaçant chaque pixel de l'image d'origine par la couleur dont il est le plus proche. À chaque itération, ensuite, on récupère tous les pixels d'une même couleur pour créer une partition. Pour chacune des k partitions obtenues, on calcule la moyenne des couleurs, ce qui donne une nouvelle liste de k couleurs. On remplace alors chaque pixel de l'image d'origine par la couleur dont il est le plus proche, ce permet d'obtenir une nouvelle image.

Le partitionnement des pixels puis le remplacement de leur couleur par une couleur moyenne est itéré jusqu'à ce que l'image ne soit plus modifiée par le procédé. L'utilisation des k-moyennes pour cette application a été longtemps considérée comme peu efficace, mais il a été avancé que l'algorithme pouvait être implémenté de façon efficace.

Les paramètres qui influent sur le résultat sont :

— la valeur de k ;
— le choix d'initialisation des couleurs ;
— l'arrêt éventuel de l'exécution avant que les couleurs ne soient complètement stabilisées.

3 Algorithme Minimax

Les jeux précédents, que l'on peut résoudre intégralement à l'aide d'un parcours de graphe, ne sont pas très intéressants : il n'existe pas de championnat du monde de jeu des batonnets ou de Morpion, puisqu'une stratégie optimale est connue ! Des jeux plus compliqués sont par exemple les échecs ou le jeu de go : ces jeux ne sont pas résolus entièrement dans le sens où on ne connaît pas de stratégie optimale. En effet, le nombre de parties possibles est de l'ordre de 10^{120} pour le jeu d'échecs et 10^{600} pour le jeu de go, ce qui rend une exploration exhaustive impossible.

Malgré cette impossibilité, les humains sont aujourd'hui dépassés par la machine 3 . Cette partie vise à donner un aperçu d'un programme informatique capable de jouer à de tels jeux où l'espace des positions est trop grand pour être exploré exhaustivement.

Dans cette partie, on étudie les jeux à information complète, à deux joueurs, à somme nulle, et à coups asynchrones en nombre fini. Précisons un peu ces termes :

— un jeu à information complète est un jeu où chaque joueur connaît les actions qu'il peut entreprendre, ainsi que ses adversaires, et les gains résultants de telles actions.
— un jeu à coups asynchrones est un jeu où chaque joueur joue alternativement. En nombre fini signifie qu'une partie ne peut être infinie (aux échecs, une règle impose que si aucun pion n'a avancé ou aucune pièce n'a été capturée en 50 coups, la partie est nulle).
— un jeu à somme nulle est un jeu où le gain du premier joueur correspond à la perte de l'autre joueur. En pratique, le gain du premier joueur est souvent réduit aux trois valeurs $+\infty$ (il gagne), 0 (partie nulle), $-\infty$ (le deuxième joueur gagne), mais dans la suite ce gain pourra être toute valeur de $\mathbb{R} \cup \{\pm\infty\}$.

Dans la suite, on appelera un tel jeu un jeu Min-Max, et les deux joueurs seront appelés Max et Min : le but de Max est de maximiser son gain, le but de Min est de minimiser le gain du joueur Max.

3.1 Représentation par un arbre

Notion d'arbre. Un tel jeu se représente sous forme arborescente. Formellement, un arbre est un graphe connexe acyclique, dont on choisit un nœud particulier (la racine), qui oriente l'arbre. En informatique, les arbres «poussent» de haut en bas : la racine est donc située en haut.

La profondeur d'un nœud est sa distance à la racine (la racine est donc l'unique nœud à profondeur zéro). Pour la représentation graphique, tous les nouds à une même profondeur sont représentés sur une même ligne horizontale. Le fait que le graphe soit connexe et acyclique impose qu'il existe un unique chemin simple (sans sommet en double) de la racine r à un nœud n quelconque de l'arbre. Le long de ce chemin il y a une relation de parenté entre nœuds successifs : si p précède q, p est le père de q et q est un fils de p. Un nœud sans fils est appelé une feuille, sinon c'est un nœud interne. Sur la figure 18.5, r est la racine, n est un nœud interne, f est une feuille.

2. Si le graphe possède des circuits, il faut être plus précis. C'est possible mais il me semble que l'on s'éloigne un peu de l'introduction à la théorie des jeux !
3. Aux jeux d'échecs, Deeper Blue bat le champion du monde en titre Garry Kasparov en 1997. Il a fallu attendre 2017 pour que le champion du monde du jeu de Go Ke Jie soit battu par le programme informatique AlphaGo. Depuis, la supériorité des algorithmes sur les humains est indéniable.

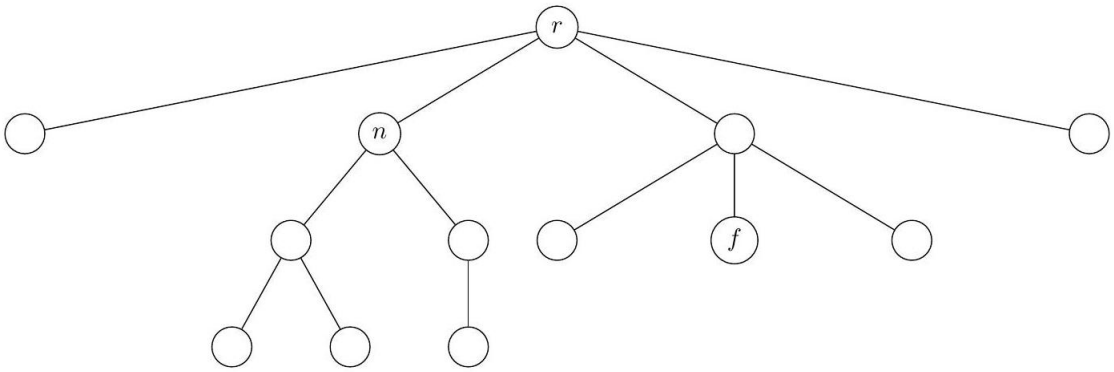


Figure 18.5 - Un arbre

Arbre pour un jeu Min-Max. Un jeu Min-Max se représente comme un arbre, où les nœuds représentent des positions du jeu. Puisque les joueurs jouent alternativement, les nœuds d'un même niveau sont alternativement contrôlés par un même joueur. Par symétrie, on peut supposer que :

— la racine est contrôlée par le joueur Max ;
— les nouds à profondeur 1 sont contrôlés par le joueur Min ;
— les nœuds à profondeur 2 sont contrôlés par le joueur Max ;
— etc...

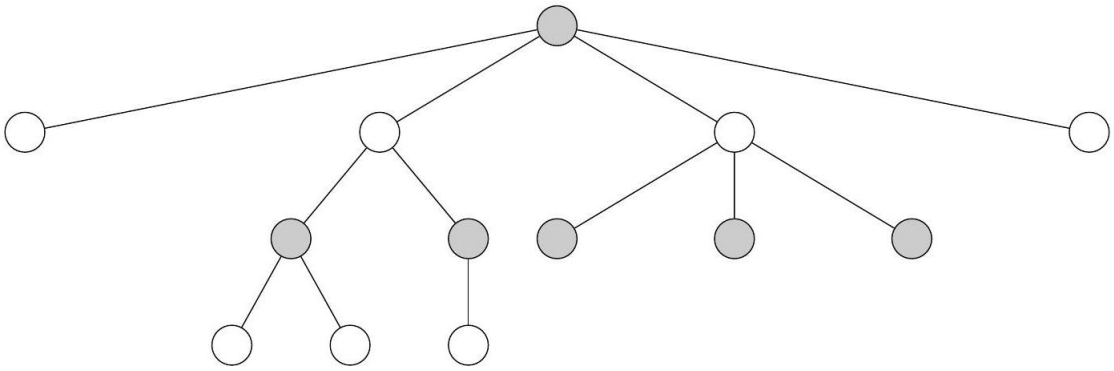


Figure 18.6 - Un arbre représentant un jeu Min-Max : les nœuds contrôlés par Max sont en gris, les autres sont contrôlés par Min.

La figure 18.6 représente un tel arbre Min-Max.

3.2 Stratégie optimale pour un jeu Min-Max

Évaluation des feuilles. Le jeu étant fini, l'arbre associé l'est aussi. Le jeu se termine lorsqu'on aboutit à une feuille f. Pour toute feuille f, on note $\mathcal{S}(f) \in \mathbb{R} \cup \{\pm\infty\}$ le score du joueur Max si cette feuille est atteinte (rappel : le but du joueur Max est de maximiser cette quantité, Min de la minimiser).

Évaluation des nœuds. On peut définir récursivement (en langage savant, on dit par induction) une extension de la fonction $\mathcal{S}$, notée $\mathcal{E}$, à tous les nœuds du graphe de la façon suivante :

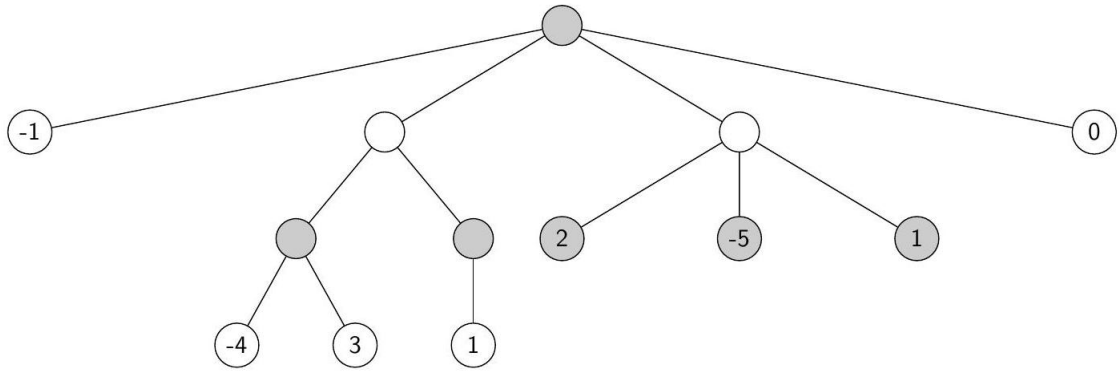
$$\mathcal{E}(n) = \begin{cases} \mathcal{S}(n) & \text{si } n \text{ est une feuille} \\ \max\{\mathcal{E}(x) \mid x \text{ fils de } n\} & \text{si } n \text{ est contrôlé par le joueur Max ;} \\ \min\{\mathcal{E}(x) \mid x \text{ fils de } n\} & \text{si } n \text{ est contrôlé par le joueur Min.} \end{cases}$$

Il est facile de montrer par récurrence que :

— si, lorsque Max doit jouer en un noud interne n, il joue un fils y de n tel que $\mathcal{E}(y) = \max\{\mathcal{E}(x) \mid x \text{ fils de } n\}$, alors son score à la fin de la partie vaut au moins $\mathcal{E}(n)$;
— si, lorsque Min doit jouer en un nœud interne n, il joue un fils y de n tel que $\mathcal{E}(y) = \min\{\mathcal{E}(x) \mid x \text{ fils de } n\}$, alors le score de Max à la fin de la partie vaut au plus $\mathcal{E}(n)$;

Stratégie optimale pour chaque joueur. Si chaque joueur joue en essayant de maximiser (resp. minimiser) la quantité $\mathcal{E}$ précédente, le score final du joueur Max à la fin de la partie sera $\mathcal{E}(r)$, avec r la racine.

Exercice 7. Dans l'arbre suivant, on a fait figurer dans chaque feuille son score $\mathcal{S}(f)$. Si chaque joueur joue optimalement, quel sera le score du joueur Max à la fin de la partie ? (Calculer la valeur de $\mathcal{E}(n)$ pour chaque nœud n).



3.3 Algorithme avec heuristique

Bien sûr, dans un jeu intéressant, l'arbre est beaucoup trop gros pour être exploré entièrement. Si on veut créer une intelligence artificielle capable de jouer assez intelligemment au jeu, il faut :

— être capable d'approximer intelligemment $\mathcal{E}(n)$ pour un nœud n qui n'est pas une feuille : on suppose connue une fonction $\mathcal{H}$ (appelée heuristique) définie sur ces nouds. Au jeux d'échecs par exemple, il est courant de compter une valeur 9 pour la dame, 5 pour une tour, 3.25 pour un fou ou un cavalier, et 1 pour un pion. En faisant la différence entre les qualités des pièces blanches et noires, on obtient une valeur raisonnable $\mathcal{H}(n)$ d'une position n.

— réduire la profondeur de recherche dans l'arbre à une certaine valeur p : dans l'évaluation $\mathcal{E}(n)$ du score d'un nœud n, on remplace l'évaluation $\mathcal{E}(n')$ des nœuds n' situés à une distance p de n par la valeur $\mathcal{H}(n')$. Ainsi, la complexité n'est pas trop élevée : Aux échecs, une profondeur de 20 est déja conséquente.

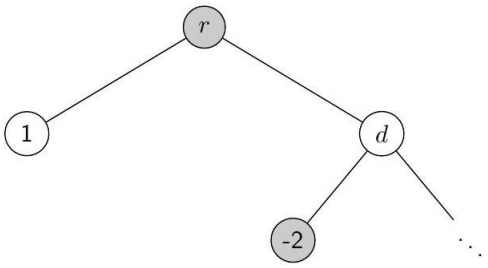
L'algorithme récursif, résume cette démarche. On renvoie ici simplement une estimation de $\mathcal{E}(n)$, pour écrire une intelligence artificielle capable de jouer, il suffirait de renvoyer également le prochain coup à jouer, dans le cas d'un nœud interne.

Algorithme 18.21 : $MiniMax(n, p, \mathcal{S}, \mathcal{H})$

Entrée : Un nœud n d'un arbre associé à un jeu Min-Max, un entier $p \geq 0$ (profondeur de recherche), une fonction $\mathcal{S}$ de score sur les feuilles, une heuristique $\mathcal{H}$ sur les autres nœuds.
Sortie : Une estimation de $\mathcal{E}(n)$, score sur le nœud n de l'arbre.
si *n est une feuille* **alors**
  Renvoyer $\mathcal{S}(n)$
si $p = 0$ **alors**
  Renvoyer $\mathcal{H}(n)$
si *n est un nœud max* **alors**
  Renvoyer $\max\{Minimax(f, p-1, \mathcal{S}, \mathcal{H}) \mid f \text{ fils de } n\}$
sinon
  Renvoyer $\min\{Minimax(f, p-1, \mathcal{S}, \mathcal{H}) \mid f \text{ fils de } n\}$

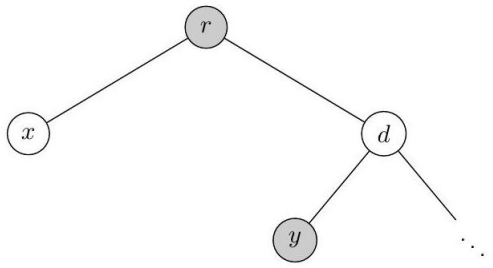
3.4 Élagage $\alpha - \beta$ (HP)

Principe. La complexité de l'algorithme 18.21 est en général exponentielle en p : en effet, si par exemple tout nœud interne de l'arbre possède a fils, la complexité est en $O(a^p)$. L'élagage $\alpha - \beta$ est une technique permettant en pratique d'éviter l'évaluation d'un bon nombre de nœuds. Prenons un exemple minimal :

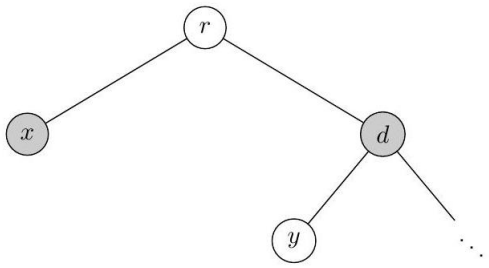


Ici, l'algorithme MiniMax a déja donné la valeur 1 au fils gauche de la racine, et est en train d'évaluer son fils droit d, qui est un nœud Min. Le fils gauche de d a déja été évalué (en -2). Puisque d est un nœud Min, on sait que la valeur qui lui sera attribuée est inférieure ou égale à -2 . Il n'est donc pas nécessaire d'évaluer la valeur de d précisément : elle n'interviendra pas dans la valeur attribuée à la racine r, puisque c'est un nœud Max qui aura donc une valeur supérieure ou égale à 1 . En particulier, l'exploration du sous-arbre droit de d peut être évitée.

Coupure α et coupure β. La figure 18.7présente les deux coupures possibles. La coupure α est la coupure présentée précédemment, qui permet de ne pas évaluer certains petits-enfants d'un nœud Max. Symétriquement, une coupure β permet de ne pas évaluer certains petits-enfants d'un nœud Min.



Coupure α : si $\mathcal{E}(x) \geq \mathcal{E}(y)$, il est inutile d'évaluer le reste de la descendance de d.



Coupure β : si $\mathcal{E}(x) \leq \mathcal{E}(y)$, il est inutile d'évaluer le reste de la descendance de d.

Figure 18.7 - Coupure α et coupure β

Modification de l'algorithme. On introduit deux nouveaux paramètres α et β dans l'algorithme, qui encadrent les valeurs intéressantes que peut prendre $\mathcal{E}(n)$ pour chaque nœud n, pour obtenir l'algorithme 18.22

- Pour l'appel initial, il suffit d'utiliser $\alpha = -\infty$ et $\beta = +\infty$;
- Pour un nœud Max n, on introduit un minorant m de la valeur $\mathcal{E}(n)$ (initialisée à $-\infty$). Cette valeur est actualisée à chaque calcul $\mathcal{E}(f)$ sur les fils de n. Si m dépasse β, on a une coupure β : la valeur $\mathcal{E}(n)$ dépassera la valeur intéressante maximale. Si m dépasse simplement α, on peut donner à α la valeur m, ce qui sera utile dans les appels récursifs sur les autres fils de n pour restreindre la plage de valeurs intéressantes.
- Symétriquement, pour un noud Min n, on introduit un majorant M de la valeur $\mathcal{E}(n)$ (initialisée à $+\infty$). Cette valeur est actualisée à chaque calcul $\mathcal{E}(f)$ sur les fils de n. Si M descend en dessous de α, on a une coupure α. Si M descend simplement en dessous de β, on peut donner à β la valeur M.

Algorithme 18.22 : $MiniMax(n, p, \alpha, \beta, \mathcal{S}, \mathcal{H})$

Entrée : Un nœud n d'un arbre associé à un jeu Min-Max, un entier $p \geq 0$ (profondeur de recherche), une fonction $\mathcal{S}$ de score sur les feuilles, une heuristique $\mathcal{H}$ sur les autres nœuds. Deux valeurs $\alpha < \beta$ qui encadrent les valeurs intéressantes de $\mathcal{E}(n)$.

Sortie : Une estimation de $\mathcal{E}(n)$, score sur le nœud n de l'arbre.

```
si n est une feuille alors
    Renvoyer S(n)
si p = 0 alors
    Renvoyer H(n)
si n est un nœud max alors
    m ← −∞;
    pour chaque fils f de n faire
        v ← Minimax(f, p − 1, α, β, S, H);
        si v > m alors
            m ← v
        si m ≥ β alors
            Renvoyer m   # coupure beta !
        α ← max(α, m);
    Renvoyer m
sinon
    M ← +∞;
    pour chaque fils f de n faire
        v ← Minimax(f, p − 1, α, β, S, H);
        si v < M alors
            M ← v
        si M ≤ α alors
            Renvoyer M   # coupure alpha !
        β ← min(β, M);
    Renvoyer M
```