LYCÉE LECONTE DE LISLE

Intelligence artificielle

Vincent Picard

1
Apprentissage supervisé

Principe de l'apprentissage supervisé

- On dispose d'un ensemble de n **données étiquetées** :

$$S = \{(\overrightarrow{x_1}, y_1), \dots, (\overrightarrow{x_n}, y_n)\}$$

- Les **données** $\overrightarrow{x_i} \in \mathcal{X}$ sont des m-uplets $(u_1, \dots, u_m) \in A_1 \times \dots \times A_m$, dont les composantes sont appelées **attributs**.
- Chaque donnée $\overrightarrow{x_i}$ est **étiquetée** par $y_i \in \mathcal{Y}$. On considère que l'ensemble des étiquettes $\mathcal{Y}$ possible est fini.
- Exemple :
  - $\mathcal{Y} = \{MP, PSI, K\}$
  - Attributs : genre (F/G), moyenne de maths, moyenne de français.
  - exemples : $\overrightarrow{x_1} = (G, 12, 16)$, $\overrightarrow{x_2} = (F, 14, 14)$, $\overrightarrow{x_3} = (F, 7, 16)$, ...
  - avec $y_1 = PSI$, $y_2 = MP$, $y_3 = K$, ...
- **But** : si on me donne une nouvelle donnée $\vec{x}$ **non étiquetée** puis-je **prédire** son étiquette ?
  - $\vec{x} = (G, 3, 18)$, $y =$???

Exemples d'application

- Reconnaissance de panneaux routiers
- Robotique
- Reconnaissance de titres musicaux
- Dispositifs biométriques
- Médecine
- ...

Les applications sont vastes et ont un fort impact économique et sociétal.

Distance euclidenne

- Dans ce cours, on se place exclusivement dans le cadre où tous les attributs des données sont numériques, c'est-à-dire que $\mathcal{X}$ est l'espace vectoriel $\mathbb{R}^m$
- On peut donc parler de **norme euclidienne** d'une donnée :

$$\|\vec{x}\|_2 = \sqrt{\sum_{i=1}^{m} x_i^2}$$

- et définir la **distance euclidienne** entre deux données :

$$d(\vec{u}, \vec{v}) = \|\vec{u} - \vec{v}\|_2 = \sqrt{\sum_{i=1}^{m} (u_i - v_i)^2}$$

- Ainsi il est possible de savoir si deux données sont proches ou pas.

Programmation

- Si on utilise la bibliothèque `Numpy` pour représenter les vecteurs :

```
import numpy as np

def dist(u, v):
    d = np.linalg.norm(u - v)
    return d
```

- Si on préfère utiliser des listes (ou des tuples) :

```
def dist(u, v):
    m = len(u)
    s = 0
    for i in range(m):
        s += (u[i] - v[i]) ** 2
    return s ** 0.5
```

- On remarque que `Numpy` gère directement le calcul dans l'espace vectoriel $\mathbb{R}^m$ (somme de vecteurs, etc...). Attention ce n'est pas le cas avec des listes ou des tuples !

Algorithme du plus proche voisin

- Une manière très simple de prédire l'étiquette d'une donnée et de trouver l'étiquette de la donnée la plus proche de l'ensemble d'apprentissage.

- En Python :

```
""" Entrée :
- S une liste de couples (donnee, etiquette)
- u une nouvelle donnée dont on doit prédire l'étiquette
Sortie : l'étiquette prédite pour u """
def PPV(S, u):
    n = len(S)
    assert(n > 0)
    pp = 0 # indice du plus proche
    dmin = dist(S[0][0], u) # distance du plus proche
    for i in range(1, n):
        if dist(S[i][0], u) < dmin:
            dmin = dist(S[i][0], u)
            pp = i
    return S[i][1]
```

Problème du sur-apprentissage

- Le problème de l'algorithme du plus proche voisin est qu'il produit du **sur-apprentissage** :
  - Tendance à vouloir *copier* les données d'apprentissage
  - Difficulté à généraliser à partir de l'ensemble de données
- Pour éviter ce problème on ne va pas seulement regarder le plus proche voisin mais les K plus proches voisins.

Algorithme des K plus proches voisins

- Principe
  - On se fixe une valeur $K \in \mathbb{N}^*$
  - Pour prédire l'étiquette de $\vec{u}$, on cherche dans l'ensemble de données S les K données les plus proches de $\vec{u}$.
  - Parmi ces K données on retourne l'étiquette qui apparaît le plus souvent.
- On aura donc besoin d'implémenter :
  - Le calcul des K plus proches voisins
  - Le calcul de la valeur majoritaire d'une liste
- Remarque : l'algorithme du plus proche voisin correspond au cas $K = 1$.

Calcul des K plus proches voisins

- Méthode simple : on trie le jeu de données par ordre croissante de distance à $\vec{u}$.
- Pour cela, on peut utiliser l'argument `key` de la méthode `sort` de tri d'une liste :

```
""" Retourne les K couples (donnees, etiquette) de S qui sont les plus
proches de u"""
def plusproches(K, S, u):
    # distance à u d'un couple (x, y)
    def distu(c):
        return dist(u, c[0])

    S.sort(key = distu)
    return S[0:K]
```

- Algorithmiquement, ce n'est pas le plus efficace, mais ça a le mérite de s'écrire simplement.

Valeur majoritaire

- On va compter le nombre d'occurrences de chaque valeur de la liste à l'aide d'un **dictionnaire**.
- Code Python :

```
def majoritaire(L):
    assert (len(L) > 0)
    occ = {} # nombre d'occurrences
    f = None # valeur la plus frequente
    occf = 0 # nombre d'occurrences de f

    for x in L:
        if x in occ: # si on a deja rencontré x
            occ[x] = occ[x] + 1
        else: # si c'est la premiere fois
            occ[x] = 1
        if occ[x] > occf:
            f = x
            occf = occ[x]
    return f
```

Implémentation des K plus proches voisins

On peut maintenant programmer la fonction qui prédit l'étiquette d'une donnée $\vec{u}$:

```
"""
K : le nombre de plus proches voisins
S : données d'apprentissage : une liste de couples (donnee, etiquette)
u : donnée à étiqueter
"""
def(K, S, u):
    pp = plusproches(K, S, u)
    etiquettes = [ c[1] for c in pp] # liste des étiquettes des K-ppv
    return majoritaire(etiquettes)
```

L'apprentissage supervisé en pratique

- Pour mettre en œuvre un algorithme d'apprentissage supervisé on partitionne l'ensemble S des données étiquetées en 3 sous-ensembles :

  1. **Un ensemble d'entraînement** : il sert de données d'apprentissage à l'algorithme d'apprentissage supervisé
  2. **Un ensemble de validation** : il sert à étudier l'impact des hyper-paramètres tels que la valeur K sur la qualité de l'apprentissage
  3. **Un ensemble de tests** : il sert à étudier la qualité des résultats obtenus avec l'algorithme calibré.

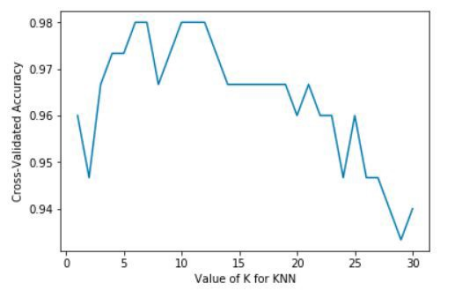
  Ces trois ensembles sont disjoints afin de ne pas introduire de biais dans le processus.

Phase de validation

- Dans l'exemple du programme : l'algorithme des K plus proches voisins, on peut étudier la qualité des résultats obtenus sur l'ensemble de validation, en fonction de la valeur K choisie.
- On s'intéresse, en fonction de K à la valeur de **précision** :

$$\text{précision} = \frac{|\text{données correctement classées}|}{|S|}$$

- Un K trop petit conduit au sur-apprentissage, un K trop grand conduit à un sous-apprentissage.



Phase de tests

- On utilise cette fois les **données de tests**, pour évaluer l'algorithme utilisé avec l'ensemble d'entrainement et les bons paramètres.
- Pour évaluer la qualité des résultats on peut construire la **matrice de confusion** :
  - Les lignes représentent les étiquettes réelles
  - Les colonnes représentent les étiquettes prédites
  - Dans la case $M_{i,j}$ on inscrit le nombre de données d'étiquette numéro i qui ont été classifiées en tant que donnée d'étiquette numéro j.
- Dans le cas **idéal** la matrice doit être diagonale : aucune erreur de prédiction.
- On peut utiliser d'autres métriques pour étudier la qualité des résultats.

2
Apprentissage non supervisé

Principe de l'apprentissage non supervisé

- Dans l'apprentissage non supervisé on dispose d'un ensemble S de n données à valeurs dans $\mathcal{X}$.
- **Ces données sont non étiquetées**.
- On cherche à anlyser les données pour en faire émerger une **structure** ou une **organisation**.
- Par exemple, on peut souhaiter pouvoir regrouper les données en **paquets** de données semblables: on appelle cela le **clustering**.
- **Exemples**

  1. Agrégations d'articles de presse : on veut regrouper les articles récents en paquets d'articles traitant de la même actualité
  2. Détection d'anomalies : un groupe de données présentant des caractéristiques différentes
  3. Identification de profils types de clients
  4. etc

L'algorithme des K moyennes

- A ne pas confondre avec l'algorithme des K plus proches voisins.
- On se fixe K un entier ≥ 2.
- On choisit initialement K points de l'ensemble, comme des centres $C_1, \ldots, C_K$.
- On regroupe trie les données de S en clusters $S_1, \ldots, S_K$ en fonction du centre le plus proche d'elle.
- On calcule pour chaque cluster S_i une nouvelle valeur de centre C_i correspondant à son barycentre.
- On réitère le processus autant de fois que voulu (en pratique : jusqu'à ce que les centres ne bougent plus beaucoup).

Implémentation en Python : barycentre

```
import numpy as np

A=np.array([1, 3])
B=np.array([1, 5])
C=np.array([2, 2])
D=np.array([3, 5])
E=np.array([1, 7])

#calcule la moyenne d'un ensemble de points
def barycentre(l):
    assert(l != [])
    d = np.zeros(len(l[0]))
    for x in l:
        d += x
    return d / len(l)
```

Implémentation en Python : cluster le plus proche

```
#calcule l'indice du mu le plus proche x
def plusproche(mu, x):
    res = -1
    dist = float('inf')
    for i in range(len(mu)):
        if eucl_dist(mu[i], x) < dist:
            dist = eucl_dist(mu[i], x)
            res = i
    return res
```

Implémentation en Python : K-moyennes

```
def kmeans(K, exemples):
    assert(len(exemples) >= K)

    mu = [0] * K # tableau pour les centres
    for i in range(K):
        mu[i] = exemples[i]
    for p in range(1000): ## 100 itérations par exemple
        # Affectation des points a leur centre le plus proche
        classe = [ 0 ] * K
        for i in range(K):
            classe[i] = []
        for x in exemples:
            j = plusproche(mu, x)
            classe[j].append(x)
        # Recalcul des centres
        for i in range(K):
            if classe[i] != []:
                mu[i] = barycentre(classe[i])
    return classe
```

Comment évaluer les résultats ?

- On cherche à obtenir des **clusters** contenant des données proches, c'est-à-dire que les points d'un cluster doivent être proche du centre du cluster.
- On considère pour cela le score de qualité

$$Q = \sum_{i=1}^{K} \sum_{\vec{x} \in S_i} \left\| \vec{x} - \vec{C_i} \right\|^2$$

- Mesure la variance autour du centre de chaque cluster.
- En pratique on peut calculer ce score et arrêter les itérations de l'algorithme quand Q ne diminue presque plus.

3
Théorie des jeux

Jeux d'accessibilité à deux joueurs

Un **jeu d'accessibilité** est un jeu qui se joue à deux joueurs notés J_1 et J_2 tels que :

- Les joueurs jouent à tour de rôle.
- L'information est **complète** c'est-à-dire que les deux joueurs connaissent toutes les règles et les conditions de victoire.
- Le jeu ne comporte pas de hasard.
- Le but pour chaque joueur est d'atteindre un certain état du jeu.

L'exemple typique de jeu d'accessibilité est le **morpion** :

- Les états sont les grilles de jeu + le numéro de joueur qui doit jouer
- Les joueurs jouent bien à tour de rôle.
- Lors de son tour, un joueur effectue une action qui peut changer l'état du jeu.
- Il n'y a pas de hasard (pas de lancé de dé par exemple)
- Chaque joueur essaie de former une grille qui contient 3 de ses symboles alignés

Autres exemples : puissance 4, les dames, les échecs, ...

Exemple : Le jeu de Nim

On dispose $N \geq 1$ bâtonnets sur une table. Le joueur J_1 commence. Lorsque c'est son tour, un joueur retire de la table 1, 2, ou 3 batonnets de la table. Le jeu est perdu par un joueur s'il enlève le dernier bâtonnet.

Décrire :

- Quels sont les états du jeu ?
- Quel(s) état(s) J_1 cherche-t-il à atteindre ?

Modélisation par un graphe

Un **jeu à d'accessibilité à deux joueurs** est un 6-uplet $G = (S_1, S_2, i, A, \Omega_1, \Omega_2)$ où :

- S_1 est l'ensemble des états **contrôlés/possédés** par J_1
- S_2 est l'ensemble des états **contrôlés/possédés** par J_2
- $i \in S_1 \cup S_2$ est l'**état initial** du jeu
- A est un ensemble d'**arcs** (x, y) (avec $x, y \in S_1 \cup S_2$ et $x \neq y$)
- $\Omega_1 \subset S_1 \cup S_2$ est l'ensemble des états finaux où J_1 gagne
- $\Omega_2 \subset S_1 \cup S_2$ est l'ensemble des états finaux où J_2 gagne

Dans cette définition on suppose aussi que S_1 et S_2 forment une partition de l'ensemble des états du jeu noté $S = S_1 \cup S_2$ (avec $S_1 \cap S_2 = \varnothing$).

Ainsi, on a modélisé le jeu par un **graphe** orienté $G = (S, A)$ où les sommets sont les états du jeu et les arcs correspondent aux possibilités d'action des joueurs.

Exemple : jeu de Nim avec $N = 8$

Graphe biparti

De plus, si on considère comme on l'a fait que les joueurs jouent à tour de rôle alors cela signifie :

$$\forall (x, y) \in S^2, \qquad ((x, y) \in A \text{ et } x \in S_1) \Rightarrow y \in S_2$$

et

$$\forall (x, y) \in S^2 \qquad ((x, y) \in A \text{ et } x \in S_2) \Rightarrow y \in S_1$$

Autrement dit les arcs relient des sommets entre S_1 et S_2 mais il n'y a pas d'arcs dans S_1 ni dans S_2 : **on dit dans ce cas que le graphe $G = (S, A)$ est biparti**.

Ce graphe est parfois appelé **arène du jeu**.

Etats finaux

La définition du jeu parle d'états finaux. Un état $x \in S$ sera dit **final** s'il n'existe aucune action à effectuer dans cet état c'est-à-dire :

$$\nexists y \in S, \qquad (x, y) \in A$$

Une autre manière de dire est qu'il n'y a pas d'arcs sortant de x.

Quand on atteint un état final le jeu s'arrête donc forécement.

Parmi les états finaux il y a ceux de Ω_1 où J_1 a gagné, ceux de Ω_2 où J_2 a gagné, mais il peut en exister qui ne sont ni dans Ω_1 ni dans Ω_2, dans ce cas on dira que la partie est **nulle**.

Exemple : jeu du morpion

Parties

Une **partie** dans un jeu d'accessibilité $G = (S_1, S_2, i, A, \Omega_1, \Omega_2)$ est une suite d'états :

$$p = (x_0, x_1, \dots, x_m)$$

dans laquelle :

- $\forall i \in [|0, m-1|], \qquad (x_i, x_{i+1}) \in A$
- $x_0 = i$

Autrement dit, une partie est un chemin dans le graphe $G = (S, A)$ qui commence par le sommet initial i. Ce chemin décrit les états du jeu atteint au cours d'une partie entre les deux joueurs.

Si de plus, x_m est final on dira que c'est une **partie terminée**. Si $x_m \in \Omega_1$ la partie est gagnée pour J_1, si $x_m \in \Omega_2$ la partie est gagnée pour J_2 sinon il y a partie nulle.

Stratégies

Une **stratégie** pour le joueur J_1 est une application

$$\begin{aligned} \sigma : S_1^* &\to S \\ x &\mapsto \sigma(x) \end{aligned}$$

qui associe à chaque état non final contrôlé par J_1 (notation S_1^*) un état $\sigma(x)$ tel que $(x, \sigma(x)) \in A$.

Autrement dit, une stratégie spécifie comment un joueur est supposé jouer dans chaque état non final possible du jeu où c'est à lui de jouer.

On peut définir de même une stratégie pour J_2.

Exemples :

- jeu de Nim : la stratégie où on enlève le maximum de batonnets possible (sauf le dernier)
- jeu du Morpion : stratégie consistant à jouer dans la première case vide disponible

Stratégies gagnantes

Une **stratégie gagnante** pour le joueur J_1, est une stratégie telle que **toute partie** où J_1 suit cette stratégie se termine par la victoire de J_1.

On définit de même une stratégie gagnante pour J_2.

Exemple : jeu de Nim $(N = 8)$.

Il est évidemment très intéressant de déterminer, s'il en existe, des stratégies gagnantes. Mais comment faire ?

Attracteurs

On va déterminer les états **à partir desquels** un joueur possède une stratégie gagnante (c'est-à-dire si on suppose que le jeu commence à cet état). On appelle ces états **attracteurs**.

Les **attracteurs** pour le joueur J_1 sont définis récursivement par :

$$Attr_0^1 = \Omega_1$$

$$Attr_{k+1}^1 = Attr_k^1 \cup \{x \in S_1 \text{ tels que } \exists y \in Attr_k^1 : (x,y) \in A\}$$
$$\cup \{x \in S_2 \text{ tels que } x \text{ non final et } \forall y \in S_1, (x,y) \in A \Rightarrow y \in Attr_k^1\}$$

Ainsi $Attr_k^1$ est l'ensemble des états pour lesquels le joueur 1 possède une stratégie pour gagner à coup sur en moins de k coups.

Explication : initialement, on place dans les attracteurs les états gagnants. Ensuite on ajoute à chaque itération :

- Les états pour lesquels le joueur 1 doit jouer et peut rejoindre un attracteur
- Les états où le joueur 2 doit jouer et n'a pas d'autre choix que de rejoindre un attracteur du joueur 1

Remarques sur les attracteurs

- $(Attr_k^1)_{k\in\mathbb{N}}$ est une suite croissante d'ensemble d'états au sens de l'inclusion :

$$Attr_0^1 \subset Attr_1^1 \subset Attr_2^1 \subset ...$$

- Si le jeu est fini (S est fini), alors la suite $(Attr_k)_{k\in\mathbb{N}}$ est stationnaire et on peut arrêter le calcul au bout d'un nombre fini d'étapes.

- La construction des attracteurs donne également la construction d'une stratégie gagnante : il suffit pour chaque état contrôlé par J_1 qui est dans l'ensemble d'attracteurs $Attr_{k+1}^1$ de choisir d'aller dans un état de $Attr_k^1$ (ce qui est possible par construction des attracteurs).

- On construit de même les attracteurs pour J_2 :

$$\begin{aligned} Attr_0^2 &= \Omega_2 \\ Attr_{k+1}^2 &= Attr_k^2 \cup \{x \in S_2 \text{ tels que } \exists y \in Attr_k^2 : (x,y) \in A\} \\ &\quad \cup \{x \in S_1 \text{ tels que } x \text{ non final et } \forall y \in S_2, (x,y) \in A \Rightarrow y \in Attr_k^2\} \end{aligned}$$

Explosion combinatoire

Ainsi, le calcul des attracteurs consiste à "résoudre" un jeu de manière exhaustive, malheureusement le nombre d'états à calculer devient très vite prohibitif :

Jeu	Nombre d'états estimés
Morpion	765
Puissance 4	4.5×10^{12}
Échecs	4.5×10^{46}
Go (19×19)	1.7×10^{172}

Le calcul des attracteurs est donc rarement possible en pratique sauf pour des très petits jeux.

Nous avons donc besoin d'un algorithme pour savoir comment jouer au mieux lorsqu'il est trop coûteux de calculer les stratégies gagnantes.

Minimax : Heuristique

Lorsqu'il est impossible de *tout calculer* on utilise une **fonction heuristique** :

$$h : S \to \mathbb{R} \cup \{+\infty, -\infty\}$$

telle que :

- $h(s) = 0$: signifie que la configuration est équilibrée
- $h(s) > 0$: signifie que J_1 a l'avantage
- $h(s) < 0$: signifie que J_2 a l'avantage
- plus $h(s)$ est grand plus J_1 a l'avantage
- $h(s) = +\infty$ signifie que la partie est gagnée pour J_1
- $h(s) = -\infty$ signifie que la partie est gagnée pour J_2

Attention, cette évaluation de la position est **statique** (elle ne considère pas les coups qui vont être joués dans la suite).

Exemple d'heuristique

Au jeu du morpion pour une configuration s donnée on peut noter $a_1(s)$ et $a_2(s)$ le nombre d'alignements disponibles restants pour J_1 et J_2 et poser $h(s) = a_1(s) - a_2(s)$.

Exemple

Principe de l'algorithme Minimax

Le principe est de tranformer l'évaluation **statique** h de l'état du jeu en une évaluation **dynamique** qui tienne compte des k prochains coups qui vont être joués. Le nombre K s'appelle la **profondeur de reflexion**.

On suppose que c'est à J_1 de jouer dans la configuration s:

- Si $k = 0$, l'évaluation de s est $h(s)$ (pas de profondeur de reflexion)
- Si $k = 1$, l'évaluation de s est le maximum des $h(s')$ où $s \to s'$ sont les actions possible de J_1.
- Si $k = 2$, l'évaluation de s est le maximum des $eval(s')$ où $s \to s'$ sont les actions possible de J_1 et $eval(s')$ est l'évaluation de s' (pour J_2) avec une profondeur de $k = 1$. C'est donc le maximum d'un minimum.
- ...

On obtient ainsi un arbre de réflexion de hauteur k où les nœuds correspondent au maximum (pour J_1) ou au minimum (pour J_2) de leurs fils.

Approche récursive

De manière plus rigoureuse, on définit l'**évaluation** $eval(s, j, k)$ de l'état s où c'est au joueur j de jouer avec une profondeur k par :

$$
\begin{aligned}
& eval(s, j, 0) = h(s) \\
& eval(s, j, k) = h(s) \text{ si s est final} \\
\forall k > 0, \quad & eval(s, 1, k) = \max\{eval(s', 2, k-1), \quad s \to s'\} \\
\forall k > 0, \quad & eval(s, 2, k) = \min\{eval(s', 1, k-1), \quad s \to s'\}
\end{aligned}
$$

Cela conduit évidemment à une implémentation récursive de l'évaluation.

Algorithme Minimax

```
def eval(config, joueur, profondeur, h):
    coups = coups_possibles(config)
    if profondeur == 0 or coups == []:
        return h(config)
    if joueur == 1: # veut maximiser
        max_eval = -float('inf')
        for c in coups:
            new_config = config.copy()
            jouer(new_config, joueur, c)
            e = eval(new_config, 2, profondeur-1, h)
            if e > max_eval:
                max_eval = e
        return max_eval
    if joueur == 2: # veut minimiser
        min_eval = +float('inf')
        for c in coups:
            new_config = config.copy()
            jouer(new_config, joueur, c)
            e = eval(new_config, 1, profondeur-1, h)
            if e < min_eval:
                min_eval = e
        return min_eval
```

Minimax : coup à jouer

On peut adapter `eval` pour qu'elle retourne un couple (evaluation, coup à jouer).

```
def minmax(config, joueur, profondeur, h):
    coups = coups_possibles(config)
    if profondeur == 0 or coups == []:
        return (h(config), None)
    if joueur == 1: # veut maximiser
        max_eval = -float('inf')
        best = None
        for c in coups:
            new_config = config.copy()
            jouer(new_config, joueur, c)
            (e, z) = minmax(new_config, 2, profondeur-1, h)
            if e > max_eval:
                max_eval = e
                best = c
        return (max_eval, best)
    if joueur == 2: # veut minimiser
        # idem adapter pour J2
```

Nous avons donc obtenu une IA capable de jouer et évaluer en réfléchissant k coups à l'avance !

Élagage α-β : principe

Hors programme en MP PC PSI !

Pour éviter d'évaluer tout l'arbre de recherche à profondeur fixée : on introduit une fenêtre $[\alpha, \beta]$ dans laquelle on est sûr que l'évaluation des nœuds ancêtres vont se situer.

- Si quand on évalue les fils d'un nœud min, on trouve une valeur $v < \alpha$, alors on sait que le résultat de ce nœud ne sera pas pris en compte par ses ancêtres. On **interrompt** l'exploration des fils : c'est la **coupure α**.
- Si quand on évalue les fils d'un nœud max, on trouve une valeur $v > \beta$, alors on sait que le résultat de ce nœud ne sera pas pris en compte par ses ancêtres. On **interrompt** l'exploration des fils : c'est la **coupure β**.

Cette technique permet d'**élaguer** l'arbre d'exploration et fait donc gagner en temps de calcul. Cela permet aussi d'envisager des plus grandes profonfeurs de réflexion.

Élagage α-β : programme

```
def eval(config, joueur, profondeur, h, alpha, beta):
    coups = coups_possibles(config)
    if profondeur == 0 or coups == []:
        return h(config)
    if joueur == 1: # veut maximiser
        max_eval = -float('inf')
        for c in coups:
            new_config = config.copy()
            jouer(new_config, joueur, c)
            e = eval(new_config, 2, profondeur-1, h)
            if e > max_eval:
                max_eval = e
            if max_eval > beta: # coupure beta
                return max_eval
            alpha = max(alpha, max_eval)
        return max_eval
    if joueur == 2: # veut minimiser
        min_eval = +float('inf')
        for c in coups:
            new_config = config.copy()
            jouer(new_config, joueur, c)
            e = eval(new_config, 1, profondeur-1, h, alpha, beta)
```

```
        if e < min_eval:
            min_eval = e
        if min_eval < alpha: # coupure alpha
            return min_eval
        beta = min(beta, min_eval)
    return min_eval
```