PROGRAMMER UN JEU

B. Landelle

Table des matières

I Programmation récursive

1 Définitions

Définition 1. *Une fonction* récursive *est une fonction qui fait appel à elle-même. Cet appel est dénommé* appel récursif *ou* récursion.

Remarque : On définit sur le même principe la notion d'algorithme *récursif*.

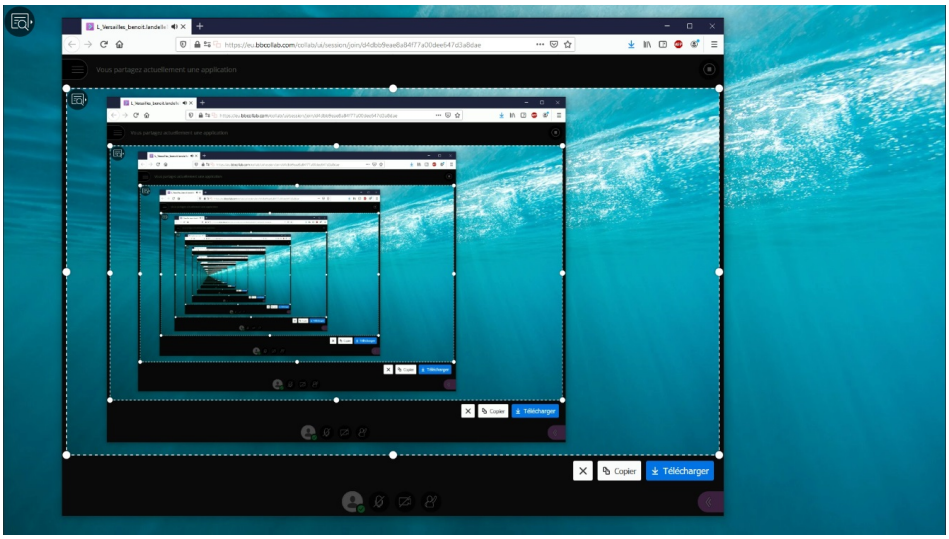

FIGURE 1 – Écran qui partage l'écran qui partage l'écran ...

Exemple : Un exemple très classique et très simple est celui de l'application factorielle. On a

$$\forall n \in \mathbb{N} \qquad n! = \begin{cases} n \times (n-1)! & \text{si } n \geqslant 1 \\ 1 & \text{si } n = 0 \end{cases}$$

```
def fact(n):
    """fact(n:int)->int
    renvoie n!"""
    if n==0:
        return 1
    else:
        return n*fact(n-1)
```

Proposition 1. *Une fonction récursive est constituée de deux alternatives fondamentales :*
- *un (ou plusieurs)* cas de base *sans appel récursif ;*
- *le* cas de propagation *avec un (ou plusieurs) appel récursif.*

Remarque : Le parallèle avec une suite définie $(u_n)_n$ définie par une relation de récurrence est flagrant : le cas de base correspond à la valeur initiale de la suite et le cas de propagation correspond à la relation de récurrence de la suite.

Exemple : Considérons la fonction `fact` de calcul de factorielle présentée précédemment.

```
def fact(n):
    if n==0:
        return 1                # cas de base
    else:
        return n*fact(n-1)      # cas de propagation avec appel récursif
```

La définition de 0! est le cas de base qui garantit la terminaison de la fonction et la relation de $n \times (n-1)!$ est le cas de propagation avec l'appel récursif à l'application factorielle.

Remarque : Le cas de base correspond à l'arrêt des appels récursifs. Si on omet ce cas, la fonction va s'appeler indéfiniment. De même, si le cas de propagation ne correspond pas à la monotonie du cas courant vers le cas de base (décroissante en général), la fonction s'appellera indéfiniment.

```
def fact_inf(n):
    return n*fact_inf(n-1)          # factorielle sans cas de base
```

```
def fact_cr(n):
    if n==0:
        return 1
    else:
        return fact_cr(n+1)//(n+1)   # propagation croissante
                                     # incohérente avec cas de base
```

Sur ces exemples, la fonction s'appelle indéfiniment. Le langage python ne détecte pas d'erreurs de syntaxe dans l'écriture de ces fonctions sans cas de base. En revanche, leurs appels provoquent des erreurs :

- `RuntimeError: maximum recursion depth exceeded` ;
- `RuntimeError: maximum recursion depth exceeded in comparison` ;

Exemple : On définit la suite $(u_n)_n$ par $u_0 = 1$ et $u_{n+1} = \sum_{i=0}^{n} u_i u_{n-i}$ (nombres de *Catalan*). La recherche d'une formule explicite de u_n ou même plus simplement l'écriture d'un algorithme itératif n'est pas immédiate. Tandis que l'écriture récursive découle naturellement de la définition de la suite.

```
def suite(n):
    """suite(n:int)->int
    Calcule le nombre de Catalan d'indice n"""
    if n==0:
        return 1
    else:
        s=0
        for i in range(n):
             s+=suite(i)*suite(n-1-i)
        return s
```

Devant l'apparente simplicité de certains codes récursifs, on peut légitimement se demander : comment sont gérés les appels successifs de la fonction par elle-même ?

La réponse est simple et élégante : le programme utilise une structure appelée *pile*, qui peut se voir comme une pile d'assiettes en remplaçant les assiettes par des données. Les appels récursifs sont empilés jusqu'à atteindre le cas de base puis sont dépilés pour réaliser le calcul de proche en proche.

2 Récursions multiples

La suite de Fibonacci $(u_n)_n$ est définie par $u_0 = u_1 = 1$ et $u_{n+2} = u_{n+1} + u_n$. Son codage récursif se fait très naturellement :

```
def fibo(n):
    """fibo(n:int)->int
    Renvoie le terme d'indice n de la suite de Fibonacci"""
    if n<=1:                          # cas de base pour n=0 et n=1
        return 1
    else:
        return fibo(n-1)+fibo(n-2)  # propagation avec récursion double
```

Notons $\mathrm{R}(n)$ le nombre de récursions de `fibo`. Il vérifie la relation

$$\mathrm{R}(0) = \mathrm{R}(1) = 1 \quad \text{et} \quad \forall n \geqslant 2 \qquad \mathrm{R}(n) = \mathrm{R}(n-1) + \mathrm{R}(n-2) + 1$$

Par une récurrence immédiate, on montre

$$\forall n \in \mathbb{N} \qquad \mathrm{R}(n) = 2u_n - 1$$

Notons $\varphi = \dfrac{1+\sqrt{5}}{2}$ (le nombre d'or). D'après le théorème sur les suites récurrentes linéaires d'ordre deux, on obtient

$$\forall n \in \mathbb{N} \qquad u_n = \frac{1}{\sqrt{5}} \left[\varphi^{n+1} - \frac{(-1)^{n+1}}{\varphi^{n+1}} \right]$$

La complexité temporelle est du même ordre que le nombre de récursions et on a donc une complexité temporelle exponentielle en $\mathrm{O}(\varphi^n)$.

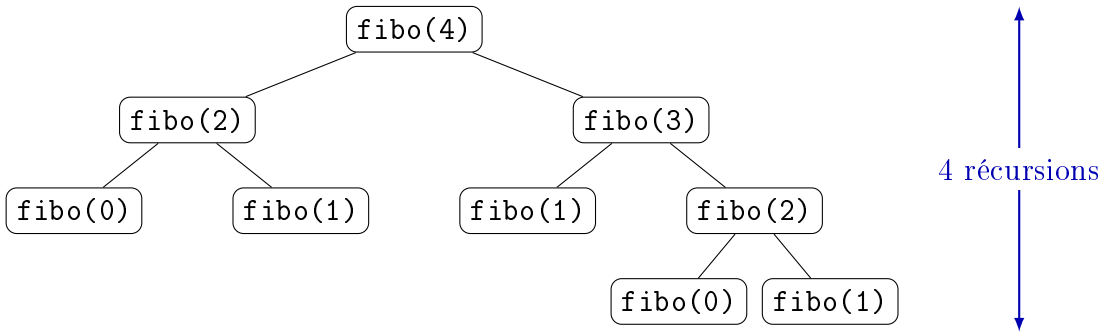

FIGURE 2 – Arbre des appels de `fibo(4)`

Ce graphique est vu comme un *arbre* (avec une arborescence vers le bas, comme dans le système racinaire d'une plante), les étiquettes `fibo(n)` étant appelées des *noeuds* de l'arbre. Cet arbre

est parcouru « en profondeur ». Ainsi, la branche la plus longue de l'arbre de `fibo(4)` à `fibo(1)` donne lieu à autant d'empilement que de noeuds dans l'arbres. Puis on dépile en remontant cette branche jusqu'à `fibo(2)`, puis on empile pour parcourir l'autre branche jusqu'à `fibo(0)`, puis on dépile en remontant cette branche jusqu'à `fibo(3)`, etc. ...

II Un premier exemple

1 Le jeu de Nim

On considère le *jeu des bâtonnets* aussi appelé *jeu de Nim*, popularisé par le jeu télévisé *Fort Boyard*. On dispose d'un tas de 20 bâtonnets et deux joueurs s'affrontent à tour de rôle en retirant 1, 2 ou 3 bâtonnets à chaque coup. Le joueur qui enlève le dernier bâtonnet perd la partie.

$$20 \underset{J_0}{\longrightarrow} 17 \underset{J_1}{\longrightarrow} 14 \underset{J_0}{\longrightarrow} 11 \underset{J_1}{\longrightarrow} 9 \underset{J_0}{\longrightarrow} 6 \underset{J_1}{\longrightarrow} 5 \underset{J_0}{\longrightarrow} 4 \underset{J_1}{\longrightarrow} 1 \underset{J_0}{\longrightarrow} 0$$

FIGURE 3 – Exemple de partie opposant les joueurs J_0 et J_1

Dans l'exemple ci-avant, le joueur J_0 démarre et retire 3 bâtonnets, il en reste 17, le joueur J_1 retire 3 bâtonnets, il en reste 14, etc ..., il en reste 5, le joueur J_0 retire 1 bâtonnet, il en reste 4, le joueur J_1 retire 3 bâtonnets, il en reste 1 et le joueur J_0 perd en retirant le dernier bâtonnet.

On peut observer qu'une fois arrivé à 5 bâtonnets, le joueur J_0 peut anticiper son échec puisque, quoi qu'il fasse, le joueur J_1 dispose à coup sûr d'un choix qui lui garantit la victoire : si le joueur J_0 retire 1 bâtonnet, alors le joueur J_1 en retire 3, si le joueur J_0 retire 2 bâtonnets, alors le joueur J_1 en retire 2 et si le joueur J_0 en retire 3, alors le joueur J_1 en retire 1 et il gagne la partie.

On peut modéliser l'ensemble des parties possibles par un graphe orienté. Pour simplifier, on considère la configuration d'un tas de 9 bâtonnets.

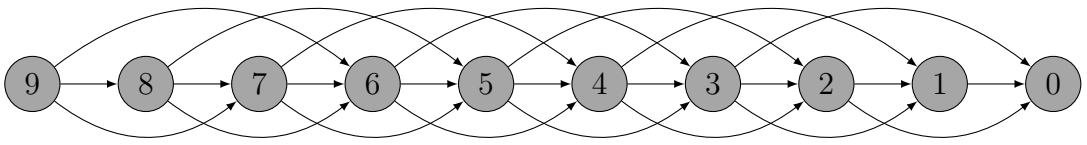

FIGURE 4 – Jeu de Nim

On peut alléger cette représentation en considérant qu'un joueur ne se « suicide » pas : il ne décide pas de perdre, une défaite est toujours subie. Par conséquent, l'état à 0 bâtonnet n'est plus pertinent dans la représentation et la partie se termine lorsqu'il ne reste plus qu'un seul bâtonnet à l'un des joueurs.

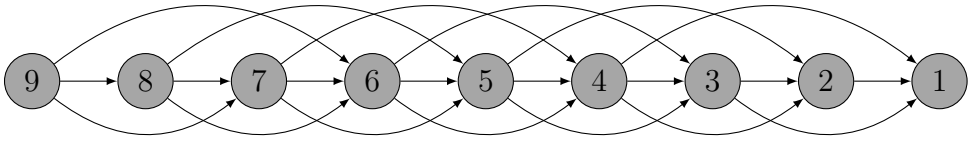

FIGURE 5 – Jeu de Nim, représentation allégée

Pour suivre le déroulé d'une partie, on peut *dédoubler* les états afin de savoir qui joue. Avec le dédoublement des états, une partie se lit complètement dans le choix d'un chemin de ce nouveau graphe orienté *biparti* : un sommet est contrôlé par un joueur et un seul et tout arc passe d'un sommet contrôlé par un joueur à un sommet contrôlé par l'autre.

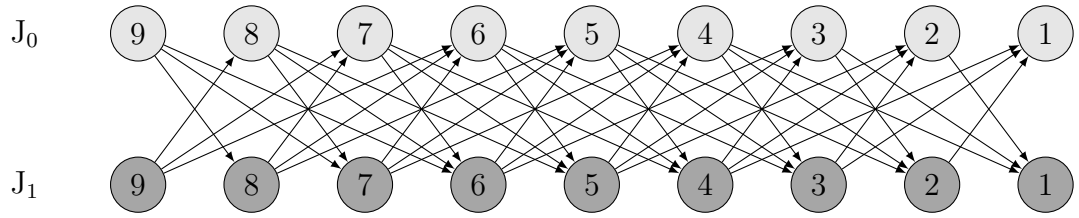

FIGURE 6 – Jeu de Nim

Dans l'exemple qui suit, on peut lire une partie démarrée et gagnée par le joueur J_0. On verra plus loin que le joueur J_1 aurait pu tirer partie de cette configuration initiale particulière.

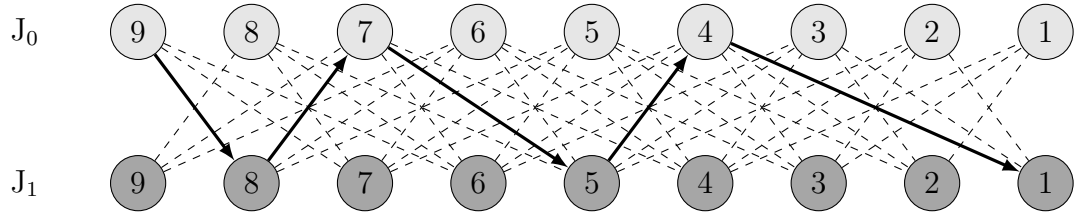

FIGURE 7 – Partie gagnée par J_0

Comme on l'avait mentionné précédemment, si un joueur atteint l'état de 5 bâtonnets, alors la défaite lui est assurée : il laissera au joueur suivant 4, 3 ou 2 bâtonnets et celui-ci n'aura plus qu'à « compléter » le retrait pour ne laisser qu'un seul bâtonnet à son adversaire.

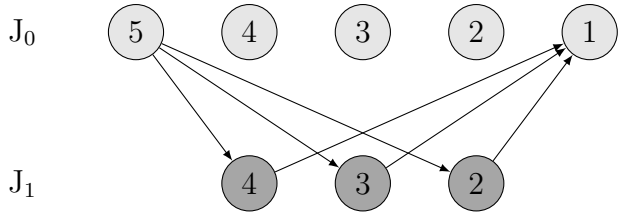

FIGURE 8 – Jeu de Nim

Il s'agit d'un fait plus général : si un joueur atteint l'état de $4n+1$ bâtonnets (n entier non nul), alors son adversaire est assuré de pouvoir le maintenir sur des états de $4k+1$ bâtonnets avec $k \in [\![0\,;\, n-1]\!]$ jusqu'à atteindre 1 bâtonnet et donc gagner. En effet, le joueur avec $4n+1$ bâtonnets en retire ℓ avec $\ell \in [\![1\,;\, 3]\!]$ et son adversaire n'a plus qu'à retirer $4-\ell \in [\![1\,;\, 3]\!]$. Le tas passe ainsi de $4n+1$ à $4n+1-\ell$ puis de $4n+1-\ell$ à $4(n-1)+1$ bâtonnets, etc

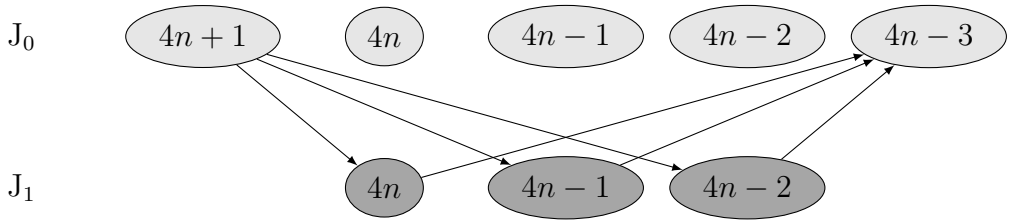

FIGURE 9 – Jeu de Nim

Une stratégie gagnante consiste donc à laisser à son adversaire un tas de $4k+1$ bâtonnets avec k entier. Si le tas initial ne contient pas $4n+1$ bâtonnets, alors le joueur qui commence et suit la règle de laisser $4k+1$ bâtonnets à son adversaire gagne. En revanche, si le tas initial contient $4n+1$ bâtonnets et que le deuxième joueur suit la règle des $4k+1$ bâtonnets, c'est lui qui gagne. Cette configuration est celle illustrée dans l'exemple 10.

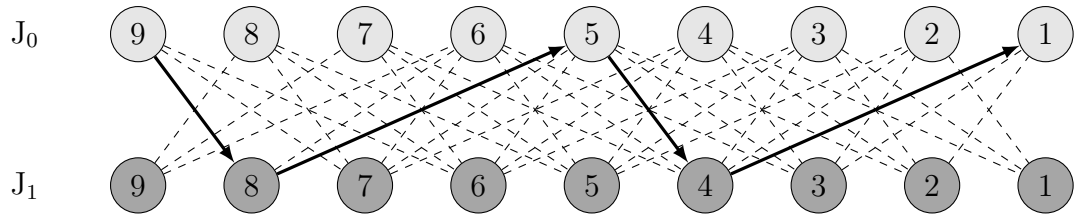

FIGURE 10 – Partie gagnée par J_1

2 Arbre du jeu

On peut représenter le jeu de Nim par un *arbre de jeu*. Par exemple, avec un tas initial de 6 bâtonnets et le joueur J_0 qui commence, le graphe biparti est décrit par

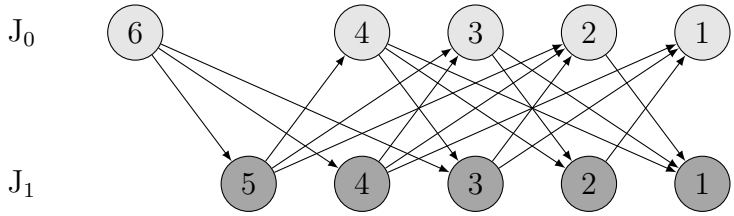

FIGURE 11 – Graphe biparti du jeu de Nim

et l'arbre de jeu correspondant est

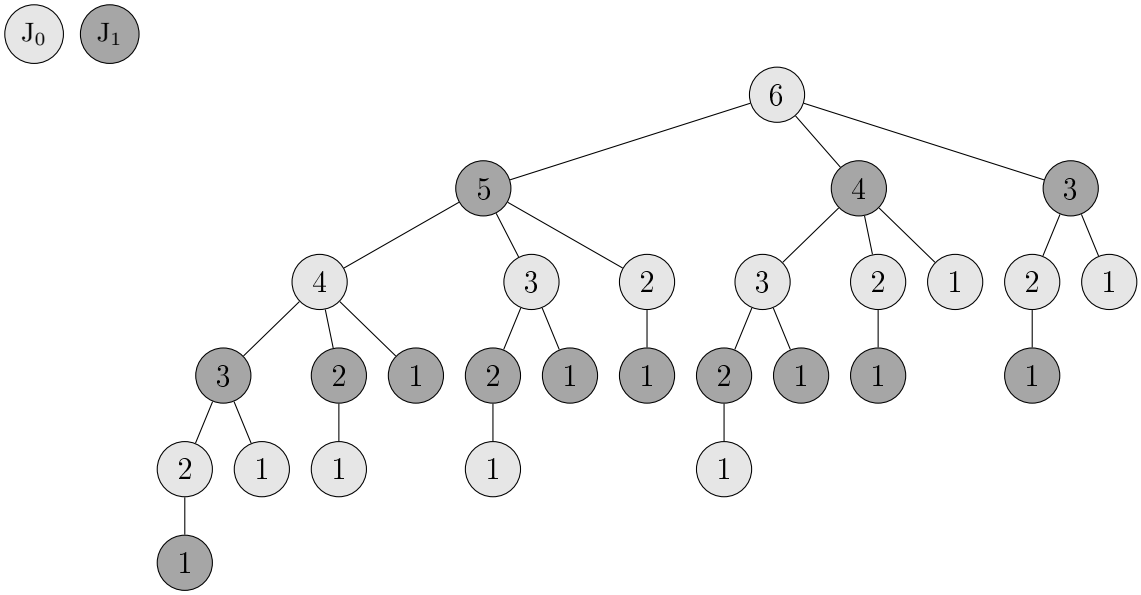

FIGURE 12 – Arbre du jeu de Nim

Définition 2. *Un* arbre *est un graphe acyclique et connexe.*

Définition 3. *Une* feuille *d'un arbre est un sommet de degré égal à* 1.

L'arbre décrit en figure 12 possède différentes feuilles qui correspondent toutes à une configuration d'un tas à 1 seul bâtonnet, soit contrôlées par J_0, soit par J_1. L'arbre se lit donc de haut en bas, de la *racine* qu'est le sommet 6, jusqu'aux feuilles qui sont les états finals du jeu.

Pour caractériser ces particularités, on considère l'*arbre enraciné* décrit par le graphe orienté :

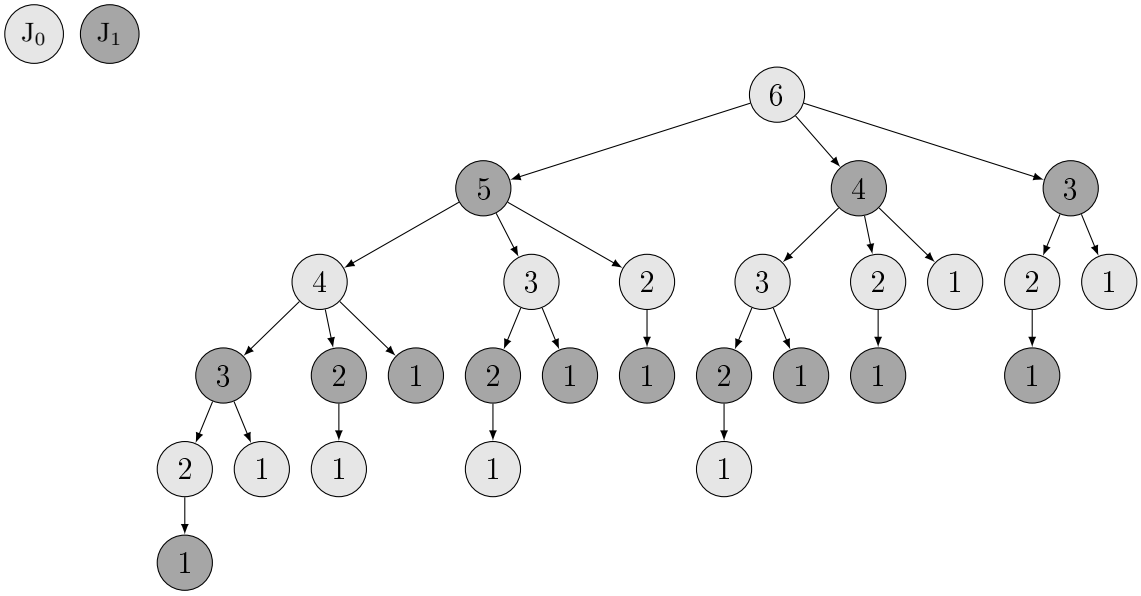

FIGURE 13 – Arborescence du jeu de Nim

Définition 4. *Une* arborescence *aussi appelée* arbre enraciné *est un graphe orienté vérifiant :*
- *le graphe sous-jacent (celui obtenu en désorientant les arcs) est un arbre ;*
- *il existe un sommet de degré entrant nul et tout autre sommet a un degré entrant égal à* 1.

Le sommet de degré entrant nul est appelé racine *de l'arbre enraciné.*

Définition 5. *Soit* (S, A) *un arbre enraciné et* $(x, y) \in A$. *Le sommet* x *est dit* père *de* y *et le sommet* y *est dit* fils *de* x. *Deux sommets ayant même père sont dits* frères. *Un sommet ayant au moins un fils est appelé* nœud interne.

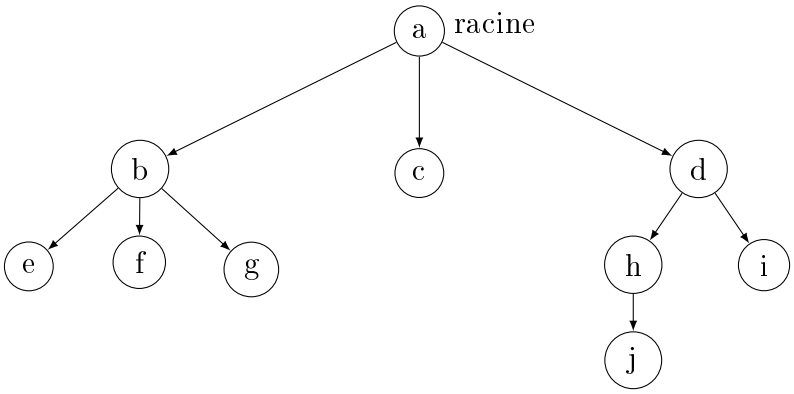

FIGURE 14 – Arbre enraciné

Le sommet a est la racine de cet arbre enraciné, les sommets b, c et d sont ses fils et sont donc frères. Les sommets e, f, g, j et i sont les feuilles de l'arbre.

Définition 6. *Soit* (S, A) *un arbre enraciné et* $x \in \mathrm{S}$. *La* profondeur *du sommet* x *est la longueur de l'unique chemin de la racine à* x. *Un* niveau *dans un graphe orienté est l'ensemble des sommets ayant une profondeur donnée.*

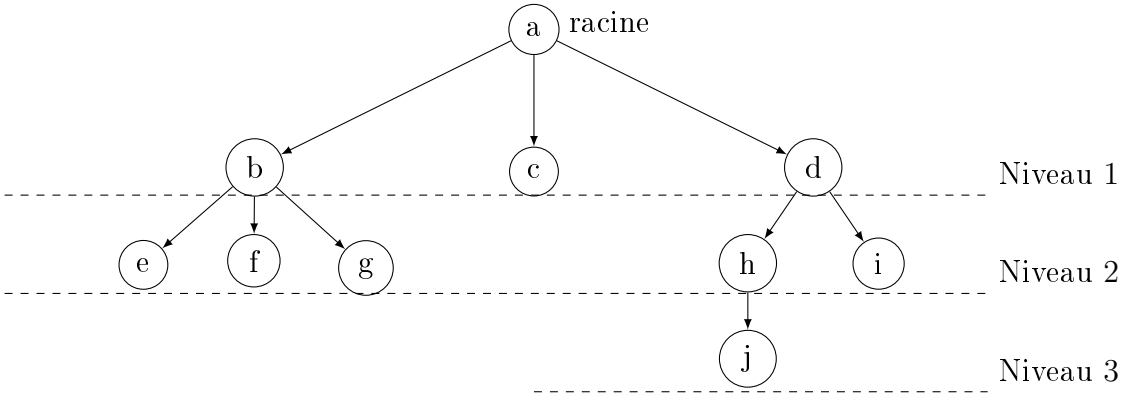

FIGURE 15 – Niveaux de profondeur

Sur l'arbre enraciné de la figure 15, le sommet j est à une profondeur égal à 3. Les sommets e, f, g, h et i sont sur un même niveau de profondeur égal à 2.

Définition 7. *Soit* (S, A) *un arbre enraciné et* $x \in \mathrm{S}$. *Une branche de* x *est un chemin partant de* x *et allant jusqu'à une feuille.*

Pour la plupart des jeux intéressants, le graphe de jeu est tellement énorme qu'il est inenvisageable de le construire. Considérer le point de vue d'une arborescence permet, sans chercher à la construire intégralement, d'envisager une méthode pour faire joueur la machine contre un autre joueur (ou contre elle-même si on souhaite se divertir).

III Algorithme du minimax

1 Principe

Lors d'une partie où chaque joueur joue de manière optimale, le joueur dont c'est le tour cherche la meilleure configuration sachant que son adversaire fera de même quand ce sera son tour.

En considérant l'arbre de jeu enraciné, on définit une fonction score $\mathcal{S}$ sur l'ensemble de ses feuilles par

$$\mathcal{S}(f) = \begin{cases} +\infty & \text{si } f \text{ feuille gagnante pour } \mathrm{J}_0 \\ -\infty & \text{si } f \text{ feuille gagnante pour } \mathrm{J}_1 \\ 0 & \text{si } f \text{ feuille de match nul} \end{cases}$$

Dans le cas du jeu de Nim avec un tas initial de 5 bâtonnets et le joueur J_0 qui commence, on obtient l'arbre de jeu :

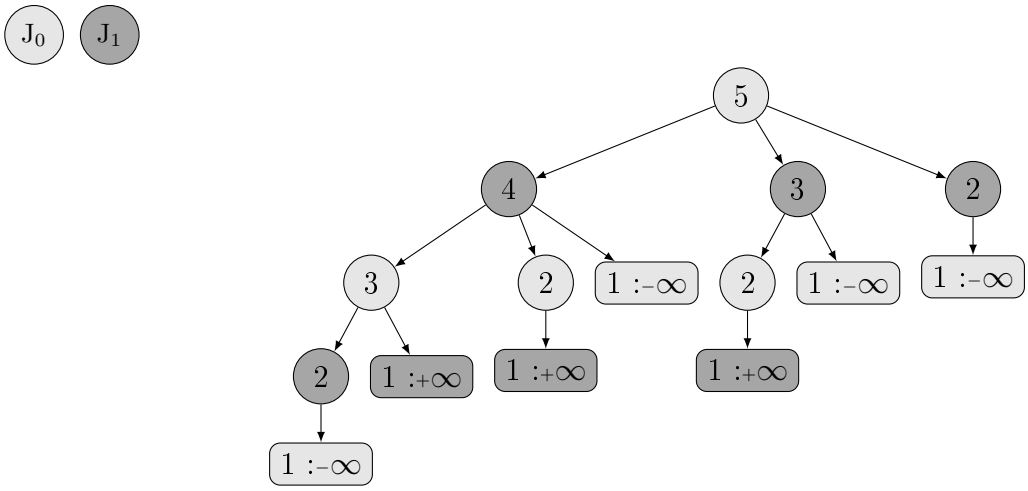

FIGURE 16 – Score des feuilles

Ensuite, on généralise cette fonction score aux autres nœuds en une fonction dite *fonction d'évaluation* notée $\mathcal{E}$ selon le principe énoncé plus haut qui amène naturellement à une définition récursive :

$$\mathcal{E}(s) = \begin{cases} \mathcal{S}(s) & \text{si } s \text{ est une feuille} \\ \max\{\mathcal{E}(u), u \text{ fils de } s\} & \text{si } s \text{ noeud interne contrôlé par } J_0 \\ \min\{\mathcal{E}(u), u \text{ fils de } s\} & \text{si } s \text{ noeud interne contrôlé par } J_1 \end{cases}$$

En suivant ce schéma récursif, on détermine, en allant de gauche à droite dans l'arborescence du jeu de Nim, l'ensemble des valeurs de la fonction d'évaluation. Après avoir complètement parcouru les branches issues du nœud 4 à gauche, on obtient :

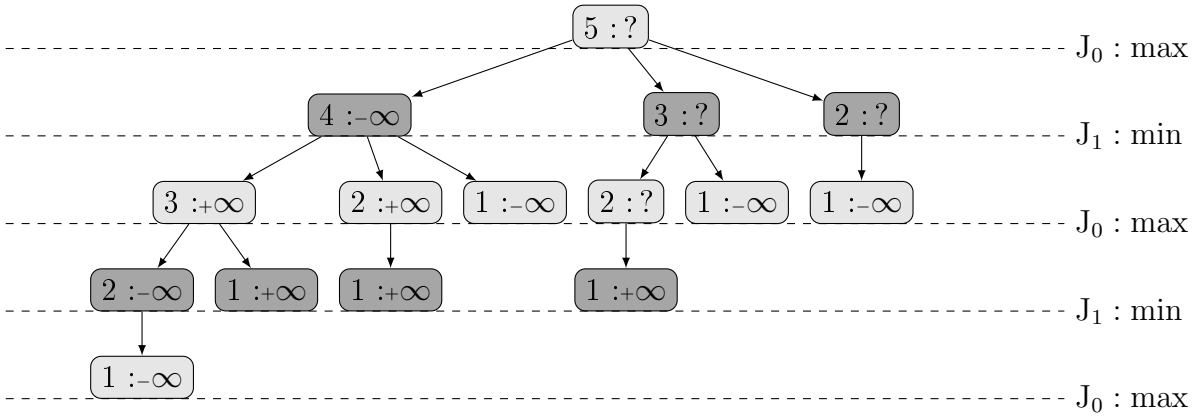

FIGURE 17 – Construction de la fonction d'évaluation

Après détermination complète de la fonction d'évaluation, on a :

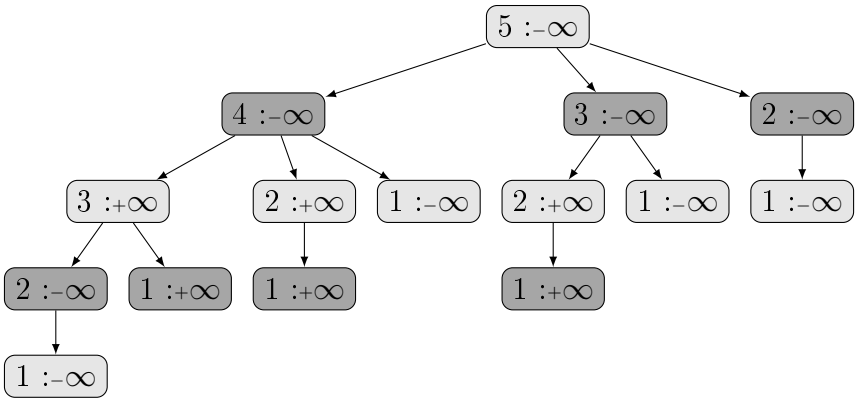

FIGURE 18 – Fonction d'évaluation

On retrouve le résultat déjà connu à savoir que pour un tas de $5 = 4+1$ bâtonnets, le joueur qui ne commence pas a une stratégie gagnante. Mais la différence réside dans le fait qu'on n'a pas instruit la machine d'une telle stratégie : le meilleur coup à jouer est déterminé par la valeur de la fonction d'évaluation, sans expertise *a priori* sur le jeu.

Définition 8. *L'algorithme* minimax *est l'algorithme de calcul récursif de la fonction d'évaluation d'un arbre de jeu.*

Algorithme 1 : Minimax

Entrées : s un nœud de l'arbre, J_i le joueur qui contrôle le nœud s
Résultat : $\mathcal{E}(s)$
fonction `Minimax(`s, J_i`)` **:**
    si s *est une feuille* **alors**
        retourner $\mathcal{S}(s)$
    sinon
        $aux \leftarrow [\,]$;
        pour u *fils de* s **faire**
            $aux \leftarrow aux+ [$ Minimax(u, adversaire(J_i)) $]$;
        si $J_i = 0$ **alors**
            retourner max(aux)
        sinon
            retourner min(aux)

On peut critiquer la construction de la liste aux puisqu'à terme, il s'agit uniquement de déterminer son maximum ou son minimum. On pourrait, selon le joueur, actualiser la valeur du maximum ou minimum au cours de la boucle sans stocker la totalité des résultats pour l'ensemble des fils de s.

Pour le jeu de Nim, on propose l'implémentation suivante :

```
def score(x):
    """score(x:int)->int
    Si J_0 a un seul batonnet, il perd   -> score=-infini
    Si J_1 a un seul batonnet, J_0 gagne -> score =+infini"""
    return (2*x-1)*float('inf')

def adversaire(x):
    return 1-x

def minimax(tas,J_i):
    """minimax(tas:int,J_i:int)->float
    Renvoie la fonction d'évaluation du jeu pour une configuration donnée.
    tas : nombre de batonnets, J_i : joueur dont c'est le tour"""
    if tas==1:
        return score(J_i)
    else:
        aux=[]
        for k in range(1,4):
            reste=tas-k
            if reste>0:
                aux.append(minimax(reste,adversaire(J_i)))
        if J_i==0:
            return max(aux)
        else:
            return min(aux)
```

Dans le cas d'une partie opposant un joueur J_0 contre la machine J_1, on programme la machine pour que celle-ci calcule la fonction d'évaluation sur les nœuds « jouables » qui correspondent à l'ensemble des coups que la machine peut jouer. Il lui suffit ensuite de choisir un nœud pour lequel le minimum de la fonction d'évaluation sur l'ensemble des nœuds jouables est atteint.

2 Heuristique

L'algorithme du minimax réalise un parcours en profondeur de l'arbre de jeu. Si celui-ci est très grand, le temps de parcours devient rédhibitoire. Une stratégie simple consiste alors à limiter la profondeur des récursions. Mais dans ce cas, quand on atteint la profondeur limite, il faut pouvoir proposer une alternative au calcul exact de la fonction d'évaluation. On utilise pour cela une *heuristique* qu'on note $\mathcal{H}$ qui pour, un nœud s donné, renvoie une valeur $\mathcal{H}(s)$ qui correspond à une estimation de $\mathcal{E}(s)$ basée sur une certaine expertise du jeu.

Algorithme 2 : Minimax_depth

Entrées : s un nœud de l'arbre, J_i le joueur qui contrôle le nœud s, p un niveau de profondeur

Résultat : Estimation de $\mathcal{E}(s)$

fonction `Minimax_depth(`s, J_i, p`)` **:**

```
si s est une feuille alors
    retourner S(s)
sinon si p = 0 alors
    retourner H(s)
sinon
    aux ← [ ];
    pour u fils de s faire
        aux ← aux+ [ Minimax_depth(u, adversaire(J_i), p − 1) ];
    si J_i = 0 alors
        retourner max(aux)
    sinon
        retourner min(aux)
```

Références

[1] `https://gfredericks.com/blog/767`

[2] `https://fr.wikipedia.org/wiki/Jeu_de_Wythoff`

[3] Laraki R., Renault J. et T. Tomala, *Théorie des Jeux*, X-UPS 2006, Éditions de l'École Polytechnique

[4] Hans Peters, *Game Theory, a Multi-Leveled Approach*, Springer, 2015

[5] Giacomo Bonamo, *Game Theory*, CreateSpace, 2018

[6] A. Artasanchez, P. Joshi, *Artificial Intelligence with Python*, Packt, 2020