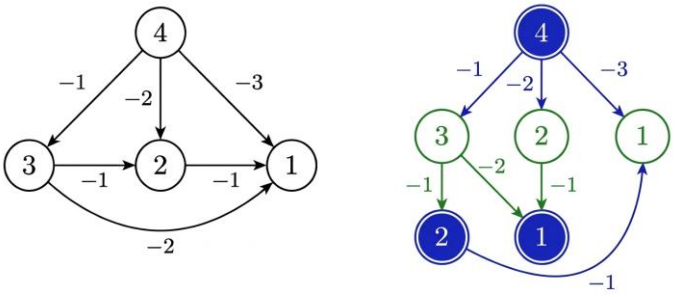
TD : JEU DE NIM

Nous allons modéliser la version classique du jeu de Nim. Il y a initialement n allumettes sur la table. Deux joueurs, J_0 et J_1, en retirent tour à tour 1, 2 ou 3. Le joueur qui enlève la dernière allumette perd la partie.

Voici avec $n = 4$, à gauche, le graphe du jeu où les sommets indiquent le nombre d'allumettes restant en jeu, et à droite le graphe biparti où les sommets doublement cerclés en bleu indiquent le tour de J_0 et les sommets en vert le tour de J_1. Ici, c'est le joueur J_0 qui commence la partie. Dans le modèle réduit adopté ici, on contracte la dernière action forcée perdante ; les sommets '1' sont donc traités comme terminaux :



I) IMPLÉMENTATION DU GRAPHE EN PYTHON

I.1. Représentation du graphe en Python

Nous allons représenter le graphe G par un dictionnaire Python où les clés sont les sommets et les valeurs sont des ensembles (set) contenant les successeurs.

Voici exactement ce qui est attendu à la sortie de la fonction pour une partie de Nim commençant avec $n = 4$ allumettes, où c'est au joueur J_0 de commencer :

```
{
    '4-J0': {'3-J1', '2-J1', '1-J1'},
    '3-J1': {'2-J0', '1-J0'},
    '2-J0': {'1-J1'},
    '2-J1': {'1-J0'},
    '1-J1': set(),
    '1-J0': set()
}
```

Les clés (les sommets du graphe) : Ce sont des chaînes de caractères qui identifient un état du jeu de manière unique. Le format est $k - J_j$, où k est le nombre d'allumettes restantes et J_j le joueur dont c'est le tour. Par exemple, `'4-J0'` signifie qu'il reste 4 allumettes et que c'est à J_0 de jouer.

Les valeurs (les successeurs) : Ce sont des ensembles Python (`set`). Ils contiennent tous les états qu'il est possible d'atteindre en jouant un seul coup légal (retirer 1, 2 ou 3 allumettes).

Les « culs-de-sac » (états finaux) : Observez les clés `'1-J1'` et `'1-J0'`. Leurs valeurs associées sont des ensembles vides `set()`. En théorie des graphes, cela correspond aux sommets dont aucune arête ne sort. Ce sont les états finaux qui marquent la fin de la partie.

I.2. Les ensembles (*set*), le piège des accolades et leurs méthodes utiles

En Python, le type `set` permet de représenter fidèlement la notion mathématique d'ensemble : il s'agit d'une collection non ordonnée d'éléments strictement uniques (sans doublon). Dans notre modélisation de graphe, utiliser des `set` est idéal pour stocker les successeurs d'un sommet, car depuis un état donné, l'ensemble des coups possibles est unique et l'ordre n'a aucune importance.

Attention cependant à un piège classique de syntaxe ! Si un ensemble contenant des éléments s'écrit bien avec des accolades (par exemple : `{'3-J1', '2-J1'}`), vous ne pouvez absolument pas utiliser `{}` pour créer un ensemble vide. En effet, les accolades étant également utilisées pour définir les dictionnaires (ex: `{'clé': 'valeur'}`), Python a fait le choix historique de réserver la syntaxe `{}` à la création d'un dictionnaire vide.

Par conséquent, pour initialiser l'ensemble vide des successeurs d'un état final (un sommet d'où plus aucune arête ne part), il est obligatoire d'utiliser la commande `set()`. Si vous utilisez `{}`, Python créera un dictionnaire, et votre programme plantera dès qu'il essaiera d'y ajouter un sommet avec la méthode `.add()`.

Une fois votre ensemble initialisé (par exemple `successeurs = set()`), **voici les outils dont vous aurez besoin pour construire votre graphe** :

- Ajouter un sommet : `.add(element)`
  Cette méthode permet d'insérer un nouvel élément dans l'ensemble.
  Exemple : `successeurs.add('2-J1')`
  Le grand avantage : si l'élément `'2-J1'` est déjà présent dans l'ensemble, Python ignorera l'instruction sans provoquer d'erreur. Cela garantit l'unicité des arêtes !

- Vérifier la présence d'un sommet : l'opérateur `in` (et `not in`)
  Comme pour les listes, vous pouvez tester très rapidement si un sommet fait déjà partie des successeurs. C'est une opération ultra-rapide (en complexité O(1)) grâce au fonctionnement interne des `set`.
  Exemple : `if '2-J1' not in successeurs :`

1. Écrire une fonction récursive `graphe_nim(n, g, j)` qui prend en paramètres le nombre `n` d'allumettes restantes, le dictionnaire `g` (initialement vide), et l'indice `j` du joueur dont c'est le tour (0 ou 1). Cette fonction doit peupler le dictionnaire `g` avec toutes les arêtes légales et le renvoyer.
Indice : Pensez à vérifier qu'il reste suffisamment d'allumettes pour en retirer.

Vérifier :

```
>>> g={}
>>> graphe_nim(4,g,0)
{'4-J0': {'3-J1', '2-J1', '1-J1'},
 '3-J1': {'2-J0', '1-J0'},
 '2-J0': {'1-J1'},
 '1-J1': set(),
 '1-J0': set(),
 '2-J1': {'1-J0'}
}
```

II) DÉFINITION DES CONDITIONS DE GAIN

À partir du dictionnaire généré pour `n = 4` :

2. Écrire une fonction `Sommets_Joueur(j,g)` pour extraire S_j, l'ensemble des sommets contrôlés par le joueur J_j.

Vérifier :
```
>>> Sommets_Joueur(0,g)
['4-J0', '2-J0', '1-J0']
```

Les deux questions suivantes ne demandent aucun code Python.

3. Un jeu de Nim est ce qu'on appelle un « jeu d'accessibilité » : la victoire se décide uniquement sur le sommet final de la partie. En observant les règles, déterminer l'ensemble F_0 contenant le (ou les) sommet(s) final(s) qui donne(nt) la victoire à J_0.

4. Déterminer l'ensemble C_0 (la condition de gain pour J_0), l'ensemble des parties $P \in \mathcal{P}$ dont le dernier sommet appartient à F_0.

Dans le cas du jeu de Nim, pour obtenir l'ensemble F_i contenant le (ou les) sommet(s) final(s) qui donne(nt) la victoire à J_i, il suffit donc de trouver l'ensemble des sommets contrôlés par l'adversaire J_{1-i} et sans successeur :

$$F_i = \{s \in S_{1-i} \mid \deg(s) = 0\}$$

5. Écrire une fonction `degres_sortants(g)` qui renvoie un dictionnaire `deg` donnant, pour chaque sommet `s`, le nombre de successeurs de `s`.

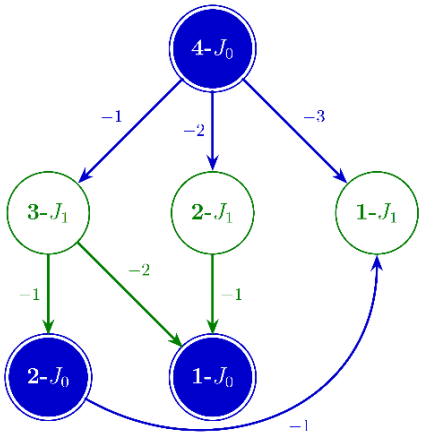
Vérifier :
```
>>> degres_sortants(g)
{'4-J0': 3, '3-J1': 2, '2-J0': 1, '1-J1': 0, '1-J0': 0,
 '2-J1': 1}
```

6. Écrire une fonction `sommets_cibles(g,i)` qui renvoie l'ensemble F_i des sommets cibles pour le joueur J_i.

Vérifier :
```
>>> g = graphe_nim(4,{},0)
>>> sommets_cibles(g,0)
['1-J1']
```

III) CALCUL DE L'ATTRACTEUR

Dans cette dernière partie, nous allons implémenter l'algorithme de calcul de l'attracteur. Cet algorithme permet à l'ordinateur de calculer l'attracteur d'un ensemble cible F pour le joueur J_i, c'est-à-dire l'ensemble des positions depuis lesquelles le joueur est certain de pouvoir forcer la victoire.



III.1. Analyse manuelle de la suite $\omega_n^{(0)}(F_0)$

Avant de coder, vérifions la théorie sur notre graphe pour n=4 allumettes. Le but de J_0 est d'atteindre l'ensemble cible $F_0 = \{'1-J1'\}$:

7. Déterminer « à la main » les ensembles $\omega_0^{(0)}(F_0)$, $\omega_1^{(0)}(F_0)$ et $\omega_2^{(0)}(F_0)$. En déduire $\omega^{(0)}(F_0)$, l'attracteur de F_0 pour le joueur J_0.

8. En déduire si la position initiale $'4-J0'$ est une position gagnante pour J_0.

III.2. Algorithme de l'attracteur

Pour information, l'algorithme de calcul de l'attracteur est rappelé à la fin du sujet. Nous allons le construire en décomposant son écriture en fonctions techniques très simples.

9. Écrire une fonction `predecesseurs(g)` qui, à partir du dictionnaire du graphe, renvoie un dictionnaire `pred` tel que `pred[s]` soit l'ensemble des prédécesseurs de `s`.

Vérifier :

```
>>> predecesseurs(g)
{'4-J0': [], '3-J1': ['4-J0'], '2-J0': ['3-J1'],
 '1-J1': ['4-J0', '2-J0'], '1-J0': ['3-J1', '2-J1'],
 '2-J1': ['4-J0']}
```

Pour la suite, on suppose donnés :

- `pred` : dictionnaire des prédécesseurs des sommets du graphe ;
- `deg` : dictionnaire donnant, pour chaque sommet, son nombre de successeurs ;
- `Si` : Liste des sommets contrôlés par J_i ;
- `omega` : Attracteur en cours de construction de l'ensemble cible F pour le joueur J_i ;
- `file` : File contenant les sommets en cours de traitement.

Pour la file `file`, on utilisera une liste Python associées à la méthode `.append()` pour y insérer un élément, et à la méthode `.pop(0)` pour extraire le premier élément de la file. De cette manière, on respecte le fonctionnement FIFO (First In First Out) de la file :

```
>>> file = [0]
>>> file.append(1)
>>> file.append(2)
>>> file
[0, 1, 2]
>>> file.pop(0)
0
>>> file
[1, 2]
>>> file.pop(0)
1
>>> file
[2]
```

10. Écrire une fonction `traiter_sommet(s,pred,deg,Si,omega,file)` qui met à jour et retourne `omega` et `file` après traitement du sommet `s`. Cette fonction doit itérer sur les prédécesseurs du sommet `s` et suivre exactement cette règle :
- Si p n'est pas dans `omega` et si $p \in S_i$, alors p entre dans `omega` et est ajouté à la file ;
- Sinon, on décrémente son compteur, et si son compteur tombe à 0, p entre dans `omega` et est ajouté à la file.

11. Écrire la fonction `Attracteur(g,Si,F)` qui calcule l'attracteur de l'ensemble `F` pour le joueur qui contrôle les sommets `Si`. Cette fonction doit :
- Initialiser l'attracteur `omega` et la file `file` avec les sommets de la cible `F` ;
- Initialiser les prédécesseurs des sommets du graphe ;
- Calculer les degrés sortants des sommets du graphe ;
- Tant que la file n'est pas vide :
  - Défiler la file (récupérer le sommet à traiter dans la file) ;
  - Traiter le sommet
- Retourner `omega`

Vérifier :
```
>>> g = graphe_nim(4,{},0)
>>> S0 = Sommets_Joueur(0,g)
>>> F0 = sommets_cibles(g,0)
>>> Attracteur(g,S0,F0)
['1-J1', '4-J0', '2-J0']
```

12. Déterminer, pour plusieurs valeurs de n, si la position initiale '`n-J0`' appartient ou non à $\omega^{(0)}(F_0)$. Conjecturer une règle générale sur les positions gagnantes.

IV) APPLICATION DU MINIMAX

Après avoir calculé qui gagne (l'attracteur), nous allons utiliser l'algorithme Minimax pour déterminer comment jouer. Pour cela, nous utilisons une fonction d'évaluation f_{nim} qui attribue des scores aux états finaux.

Dans la version du jeu de Nim étudiée (celui qui prend la dernière allumette perd), un joueur gagne s'il laisse son adversaire devant 1 seule allumette.

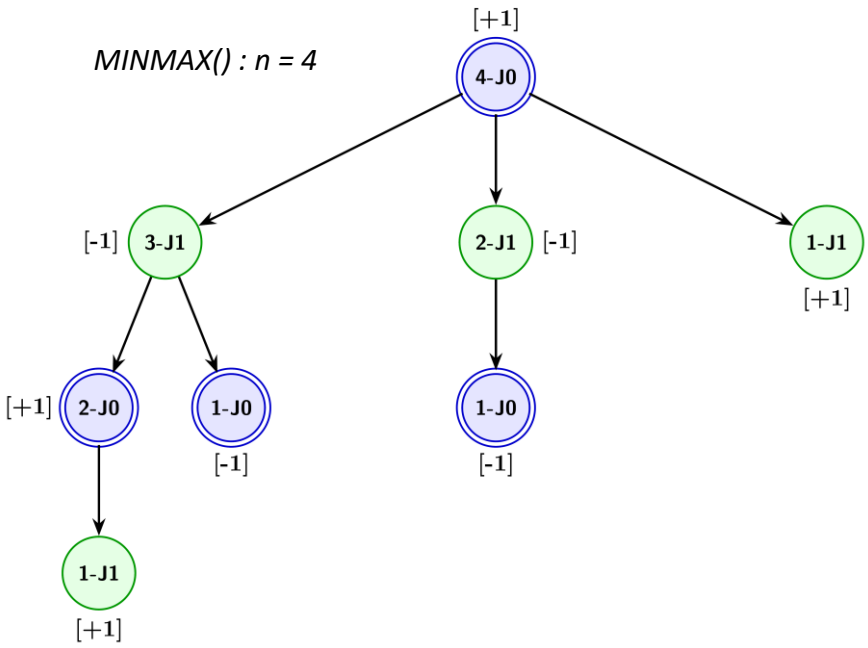
13. Écrire une fonction `f_nim(feuille,i)` qui prend en paramètre un sommet final (par exemple '`1-J1`') et l'indice du joueur J_i. Si c'est au joueur J_i de jouer, la fonction attribue la valeur 1 aux feuilles '`1-`J_{1-i}', et la valeur -1 aux feuilles '`1-`J_i'.

Vérifier :
```
>>> f_nim('1-J1',1)
-1
>>> f_nim('1-J1',0)
1
>>> f_nim('1-J0',0)
-1
>>> f_nim('1-J0',1)
1
```

L'algorithme MINIMAX(G, i, f, s) est rappelé à la fin du TD.

14. Écrire la fonction `MINIMAX(g,i,f,s)` permettant de calculer la valeur minimax d'un sommet s, du point de vue du joueur J_i.

Vérifier votre algorithme à l'aide de l'arbre MINMAX ci-dessous :



15. À l'aide de l'arbre ci-dessus, justifier pourquoi le joueur J_0 possède une stratégie gagnante à partir de 4 allumettes.

16. En comparant les résultats de `Attracteur(g,Si,F)` (question 11) et de `MINIMAX(g,i,f,s)`, vérifier que :

$$\text{sommet s} \in \omega^{(0)}(F_0) \iff \text{MINIMAX}(s) = 1$$

V) ÉCRITURE D'UNE STRATÉGIE GAGNANTE À L'AIDE DU MINIMAX

17. L'algorithme `MINIMAX` actuel renvoie la valeur du sommet (1 ou -1). Modifier cet algorithme pour qu'il renvoie également le sommet fils qui permet d'obtenir cette valeur, et `None` si aucun n'existe. Ce fils sera considéré comme le meilleur coup à jouer.

Vérifier :

```
>>> g = graphe_nim(4,{},0)
>>> strategie_minimax(g,0,f_nim,'4-J0')
(1, '1-J1')
>>> g = graphe_nim(5,{},0)
>>> strategie_minimax(g,0,f_nim,'5-J0')
(-1, None)
>>> g = graphe_nim(6,{},0)
>>> strategie_minimax(g,0,f_nim,'6-J0')
(1, '5-J1')
```

18. En utilisant la fonction `strategie_minimax`, simuler une partie complète entre deux joueurs utilisant l'algorithme. Si l'algorithme minimax renvoie `None`, le coup sera tiré au hasard par le joueur courant.

- Vérifier pour n=4 que si le jeu commence à '4-J0', le joueur J_0 gagne contre n'importe quelle défense de J_1 :

```
>> Simulation pour n=4
>> 4-J0 -> 1-J1
```

- Vérifier pour n=5 que si le jeu commence à '5-J0', le joueur J_1 gagne contre n'importe quel coup de J_0. Par exemple :

```
>> Simulation pour n=5
>> 5-J0 -> 4-J1
>> 4-J1 -> 1-J0
```

- Vérifier pour n=6 que si le jeu commence à '6-J0', le joueur J_0 gagne contre n'importe quelle défense de J_1. Par exemple :

```
>> Simulation pour n=6
>> 6-J0 -> 5-J1
>> 5-J1 -> 3-J0
>> 3-J0 -> 1-J1
```

Algorithme de calcul de l'attracteur (avec une file – propagation en largeur)

Algorithme de calcul de l'attracteur

Entrée : G : dictionnaire du graphe $G(S, A)$ (G[x] : ensemble des successeurs de x)
S_i : Ensemble des sommets contrôlés par J_i
F : Sommets cibles pour le joueur J_i
Sortie : Ω : L'attracteur de F pour le joueur J_i

ATTRACTEUR (G, S_i, F) :

```
Ω := F          # Positions gagnantes trouvées pour le joueur J_i
file := F       # File d'attente initialisée avec les sommets de F

# Initialisation : calcul des prédécesseurs et des degrés sortants de chaque sommet
Pour tout s ∈ S :
    deg[s] := 0                 # Degré sortant de s
    pred[s] := ∅                # Liste des prédécesseurs de s
Pour tout s ∈ S :
    # Pour chaque successeur s+ du sommet courant s
    Pour tout s+ ∈ G[s] :
        deg[s] := deg[s] + 1            # Incrémente le degré de s
        pred[s+] := pred[s+] ∪ {s}      # Ajoute s aux prédécesseurs du sommet s+

# Boucle principale : propagation rétrograde de l'attracteur
Tant que file est non vide :
    # Récupère le sommet dans la file
    s := défiler(file)

    # Analyse des prédécesseurs de s
    Pour tout p ∈ pred[s] :
        # Si le prédécesseur n'est pas dans l'attracteur
        Si p ∉ Ω :
            # et s'il est contrôlé par le joueur S_i
            Si p ∈ S_i :
                Ω := Ω ∪ {p}        # Alors on l'ajoute à l'attracteur
                Enfiler(p)          # et à la liste des prochains sommets à analyser
            # Sinon, si c'est un prédécesseur contrôlé par l'adversaire
            Sinon :
                # On décrémente son degré
                deg[p] := deg[p] - 1
                # Si son degré tombe à 0, on l'ajoute à l'attracteur et à la file
                Si deg[p] == 0 :
                    Ω := Ω ∪ {p}
                    Enfiler(p)
Retourner Ω
```

Algorithme MINIMAX

Algorithme MINIMAX

Entrée : G : dictionnaire du graphe $G(S, A)$ (G[x] : ensemble des successeurs de x)
i : Indice du joueur J_i
f : Fonction d'évaluation des sommets terminaux (feuilles)
s : Sommet actuel (racine de l'exploration)

Sortie : $v(s)$: Valeur minimax du sommet s, du point de vue du joueur J_i

MINIMAX (G, i, f, s) :

```
# Si s est un état final (feuille), on évalue la feuille
Si G[s] == ∅ :
    Retourner f(s)

# Si c'est au joueur J_i de jouer, il maximise
Si s ∈ S_i :
    Retourner max_{s+ ∈ G[s]} {MINIMAX(G, i, f, s+)}

# Si c'est au joueur J_{1−i} de jouer, il cherche à minimiser
Sinon :
    Retourner min_{s+ ∈ G[s]} {MINIMAX(G, i, f, s+)}
```