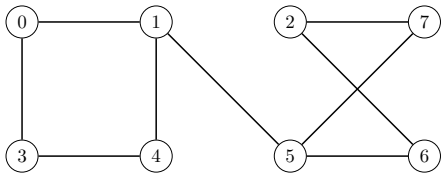
Jeux à deux joueurs

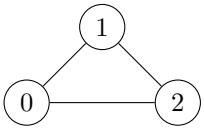
Un **puit** t dans un graphe orienté est un sommet qui n'a pas de successeur (il n'existe pas d'arête allant vers t).

Théorème : Tout graphe orienté acyclique G possède un puit.
Preuve : Soit v un sommet de G. On fait partir un chemin depuis v en se déplaçant vers un successeur à chaque étape. Comme G est acyclique, ce chemin ne peut pas revenir deux fois au même sommet. Comme le nombre de sommets est fini, ce chemin doit forcément arriver à un puit.

Un graphe $G = (V, E)$ est **biparti** s'il existe une partition $V = V_A \sqcup V_B$ telle que toute arête de E a une extrémité dans V_A et une extrémité dans V_B.



Graphe biparti, avec une partition $V_A = \{0, 4, 5, 2\}$ et $V_B = \{1, 3, 6, 7\}$



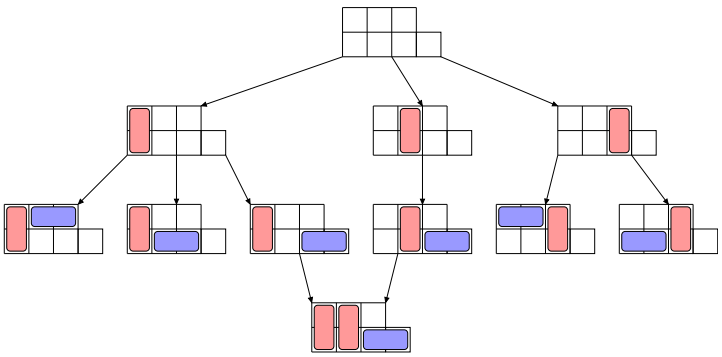
Graphe non biparti

On s'intéresse à un jeu à deux joueurs (Alice et Bob), qui se joue chacun son tour. On suppose qu'Alice commence.
Un joueur a perdu lorsqu'il n'a plus de coup possible.
Remarque : Les règles peuvent changer suivant le jeu auquel on joue.

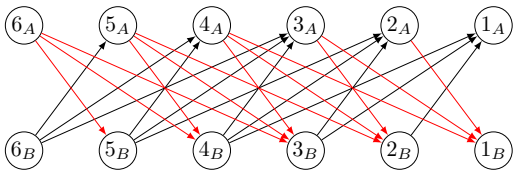
Exemples de jeux :

- Le jeu de Nim : il y a n allumettes.
  Chaque joueur peut retirer 1, 2 ou 3 allumettes.
  Le joueur qui retire la dernière allumette a perdu.
- Le jeu du domineering est un jeu de plateau où Alice place un domino vertical et Bob un domino horizontal.
  Un joueur qui ne peut plus jouer perd.

Le **graphe des configurations** d'un jeu est un graphe orienté dont les sommets sont les configurations possibles du jeu et les arêtes sont les coups possibles.



Graphe des configurations d'un jeu de domineering



Graphe des configurations d'un jeu de Nim avec, initialement $n = 6$.
Pour déterminer cmplètement une configuration, on indique le joueur dont c'est le tour.

Soit $G = (V, E)$ un graphe biparti acyclique, avec $V = V_A \sqcup V_B$.

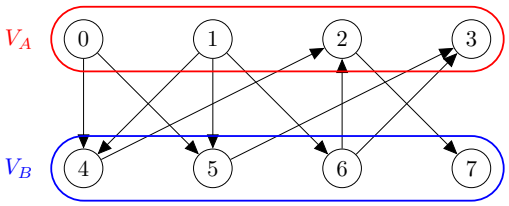
- Le jeu commence en un **sommet initial** $v \in A$.
- Une **partie** est un chemin commençant en v dont les arêtes sont choisies alternativement par Alice et Bob.
- L'ensemble P_A des puits de V_A sont les situations où Alice perd.
- Une **stratégie** pour Alice est une fonction $f : V_A \setminus P_A \to V_B$ telle que $\forall v \in V_A \setminus P_A$, $(v, f(v)) \in E$.
  Une stratégie consiste donc à choisir un coup à jouer pour chaque configuration possible.
- Une **stratégie gagnante** pour Alice est une stratégie f qui permette à Alice de gagner, quel que soit la stratégie de Bob.
- $v \in V$ est une **position gagnante** pour Alice si elle possède une stratégie gagnante pour une partie qui commence en v.
- L'**attracteur** A d'Alice est l'ensemble des position gagnantes pour Alice.

Les définitions sont similaires pour Bob en échangeant A et B.

On peut déterminer l'attracteur A d'Alice de proche en proche, en calculant l'ensemble A_k des positions gagnantes pour Alice en k coups :

- $A_0 = P_B$ (gagnants pour Alice).
- Si k est pair : $A_{k+1} = \{u \in V_A \mid \exists (u, v) \in E, v \in A_k\}$
- Si k est impair : $A_{k+1} = \{u \in V_B \mid \forall (u, v) \in E, v \in A_k\}$

S'il y a n sommets, un chemin possède au plus $n-1$ arêtes d'où : $A = \bigcup_{k=0}^{n-1} A_k$.
Exemple :



$A_0 = \{7\}$, $A_1 = \{2\}$, $A_2 = \{4\}$, $A_3 = \{0, 1\}$.
Ainsi l'attracteur A d'Alice est $\{0, 1, 2, 4, 7\}$.
On peut aussi en déduire une stratégie gagnante qui consiste à jouer sur un attracteur si possible : $f(0) = 4$, $f(1) = 4$, $f(2) = 7$.

(Formulation récursive équivalente à la précédente)
Un sommet v est un attracteur dans l'un des cas suivants :

- $v \in P_B$.
- $v \in V_A$ et il existe un attracteur w tel que $(v, w) \in E$.
- $v \in V_B$ et pour tout $(v, w) \in E$, w est un attracteur.

D'où la fonction récursive (mémoïsée pour éviter des calculs inutiles), où `G` est représenté par dictionnaire d'adjacence et `fA` est une fonction indiquant si un sommet appartient à V_A :

```
def attracteurs(G, fA):
    d = {} # d[v] = True si v est un attracteur
    def aux(v): # détermine si v est un attracteur
        if v not in d:
            succ = [aux(w) for w in G[v]]
            if len(G[v]) == 0:
                d[v] = not fA(v)
            elif fA(v):
                # test s'il existe (v, w) € E avec w attracteur
                d[v] = False
                for w in G[v]:
                    if aux(w):
                        d[v] = True
            else:
                # test si pour tout (v, w) € E, w est attracteur
                d[v] = True
                for w in G[v]:
                    if not aux(w):
                        d[v] = False
        return d[v]
    return [v for v in G if aux(v)]
```

Une **heuristique** pour un jeu est une fonction qui à une configuration associe une valeur dans $\mathbb{R}$ et qui estime à quel point la configuration v est favorable à un joueur : plus $h(v)$ est grand, plus v est favorable à Alice et inversement.

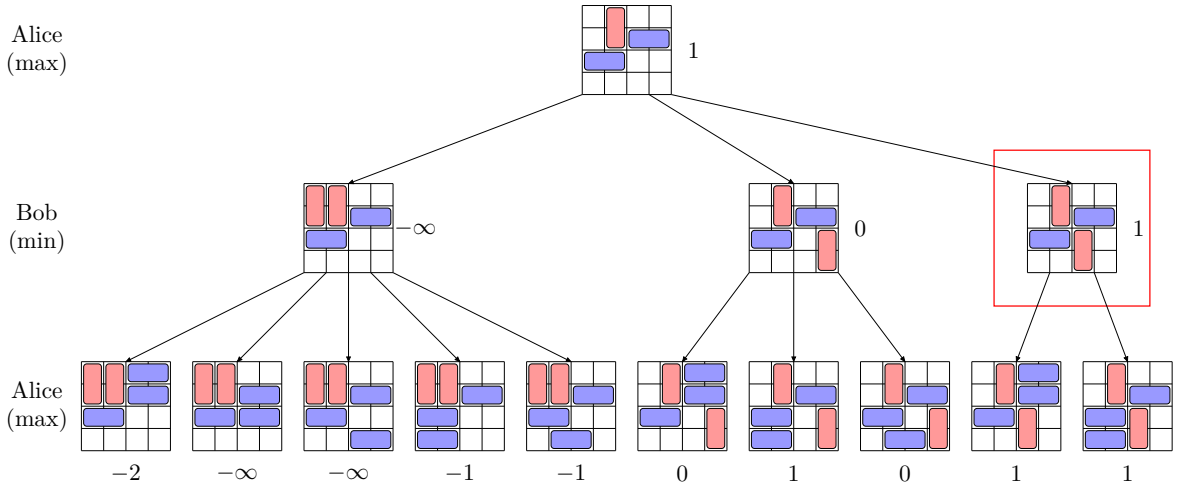
Exemple : dans le jeu du domineering, on peut utiliser $h(v) =$ nombres de possibilités pour Alice - nombres de possibilités pour Bob.
Attention : Aussi bien dans A* que min-max, utiliser une heuristique quelconque peut permettre d'accélerer la recherche mais le résultat n'est pas forcément optimal.

L'algorithme de calcul des attracteurs risque de ne pas terminer si le graphe des configurations est trop grand (échecs, go...).
L'algorithme **min-max** permet d'accélerer la recherche avec une heuristique et en ne parcourant que les configurations atteignables en au plus p coups. Il donne une valeur à chaque sommet de l'arbre de proche en proche :

- La valeur des sommets à profondeur p et ceux sans successeurs.
- La valeur des sommets à profondeur $p-1$ est le maximum (si Alice doit jouer) ou le minimum (si Bob doit jouer) des valeurs des successeurs.
- Ainsi de suite, jusqu'à calculer la valeur de la racine.

```
def minmax(s, h, v, p, j):
    succ = [minmax(s, h, w, p - 1, 1 - j) for w in s(v, j)]
    if succ == [] or p == 0:
        return h(v)
    if j == 0:
        return max(succ)
    else:
        return min(succ)
```

Une fois la valeur de chaque configuration calculée, Alice choisit le coup dont la valeur est la plus élevée (le plus favorable).



Arbre min-max rempli de bas en haut, avec le coup optimal pour Alice entouré.