



## Informatique du Tronc Commun

# Chapitre n°5 Théorie des jeux

## Introduction

Dans ce chapitre nous allons étudier la théorie de certains jeux à deux joueurs où chaque joueur joue alternativement.

La théorie des jeux a été initiée par John VON NEUMANN pendant et après la seconde guerre mondiale. Il s'intéresse aux jeux à deux joueurs qui disposent d'un nombre fini d'actions et se poursuit avec le théorème général de Nash en 1950. Même si le mot jeu a un côté plaisant, amusant, la recherche sur les jeux est un domaine sérieux et a connu un essor très important avec des applications dans de nombreux domaines comme l'économie, la politique, la physique, la biologie. La recherche informatique s'est aussi développée avec les jeux sur les graphes qui ont des applications dans plusieurs domaines d'informatique théorique et des mathématiques en permettant de modéliser certains problèmes.

Tout comme les algorithmes d'IA développés aujourd'hui, cette théorie s'inscrit dans l'objectif d'aider à la décision lorsque l'environnement est incertain, c'est-à-dire complexe et imprévisible. Elle fait intervenir des joueurs considérés comme des individus rationnels, des règles et des contextes d'évolution du jeu.

## Objectifs

Les objectifs de ce chapitre sont de :

- Assimiler la notion de jeux d'accessibilité à deux joueurs : vocabulaire, modélisation par un graphe biparti.
- Comprendre la notion de stratégie : déterminer une position gagnante, utiliser la notion d'attracteur, construire une stratégie gagnante.
- Connaître le fonctionnement de l'algorithme min-max.

## Pré-requis

- Graphes

## Plan du cours

### I Jeux d'accessibilité à deux joueurs sur un graphe

2

I.1 Définitions . . . . .	2
I.2 Modélisation par un graphe biparti . . . . .	2
I.2.a) Graphe du jeu . . . . .	2
I.2.b) Graphe biparti . . . . .	3
I.2.c) Graphe du jeu de Nim . . . . .	3
I.3 Stratégies et positions gagnantes . . . . .	3
I.4 Calcul des positions gagnantes . . . . .	4

### II Heuristique. Algorithme Min-Max

9

II.1 Heuristique . . . . .	9
II.1.a) Définition . . . . .	9
II.1.b) Exemple : puissance 4 . . . . .	9
II.2 Algorithme Min-Max avec heuristique . . . . .	10
II.2.a) Principe . . . . .	10
II.2.b) Exemples simples de mise en œuvre . . . . .	10
II.2.c) Écriture de l'algorithme . . . . .	13

I.4.a) Attracteurs . . . . .	4
I.4.b) Algorithme de calcul de l'attracteur . . . . .	6
I.5 Stratégie gagnante . . . . .	8

# I Jeux d'accessibilité à deux joueurs sur un graphe

## I.1 Définitions



### Définition : Jeux d'accessibilité

Un jeu d'accessibilité est un jeu à deux joueurs, à information complète et parfaite, séquentiel et pour lequel il n'y a pas de hasard.

- **jeux à information complète** : tous les joueurs ont une connaissance totale des données du jeu : règles, pièces, actions possibles, fonction de gain, objectifs des autres joueurs.  
Cela exclu la plupart des jeux de cartes puisqu'on ne connaît pas le jeu de l'adversaire.
- **jeux séquentiels** : les joueurs décident de leur stratégie les uns après les autres et peuvent donc tenir compte des actions des joueurs précédents.
- **jeux sans mémoire** : le joueur qui doit jouer prend une décision en fonction de la situation présente et pas de situations passées.
- **jeux sans hasard** : dans une situation donnée, une décision amène toujours à la même nouvelle situation.

**Exemple 1.** Morpion, Dames, Echecs, Puissance 4

**Exemple 2.** Jeu de Nim

Un jeu de Nim est un jeu d'accessibilité dont il existe de nombreuses variantes. Il s'agit de déplacer, de poser ou de retirer un certain nombre d'objets simples (pièces, allumettes, graines, des billes...). Le dernier à jouer gagne ou perd (variante mi-sère). Le jeu de Nim fait donc nécessairement un perdant et un gagnant.

Une des « célèbres » variantes est le jeu des bâtonnets de Fort Boyard. Dans ce duel, le candidat et le Maître se retrouvent en face d'un alignement de plusieurs bâtonnets blancs en bois. Chacun leur tour ils peuvent en retirer un, deux, ou trois. Celui qui prend le dernier bâtonnet perd le duel.



FIGURE 1 – Jeu des bâtonnets

On se basera sur ce jeu pour l'ensemble du chapitre. Les deux joueurs seront notés  $J_1$  et  $J_2$  :  $J_1$  est le premier joueur à jouer. Il ne peut pas y avoir de match nul, ni de partie infinie.

## I.2 Modélisation par un graphe biparti

### I.2.a) Graphe du jeu



### Définition : Graphe du jeu

On modélise l'ensemble du jeu par un graphe orienté  $G = (S, A)$ , tel que chaque sommet dans  $S$  représente une position du jeu (à partir de laquelle  $J_1$  ou  $J_2$  doit jouer) et chaque arrête  $a = (s_1, s_2) \in A$  signifie qu'un des deux joueurs peut passer de la position  $s_1$  à la position  $s_2$ .

Le jeu peut être dans le même état au moment où soit  $J_1$ , soit  $J_2$  joue : ce sont deux positions différentes (donc deux sommets différents).

### 1.2.b) Graphe biparti



#### Définition : Graphe biparti

Un graphe  $(S, A)$  est biparti s'il existe une partition de  $S$  en deux parties  $S_1$  et  $S_2$ , telles qu'aucune arête de  $S$  ne relie deux sommets de  $S_1$ , ou deux sommets de  $S_2$  (les joueurs jouent à tour de rôle).

Les sommets de  $S_1$  représentent les situations où le joueur 1 joue (et donc choisit son coup), on dit que ces **sommets sont contrôlés** par  $J_1$ , les sommets de  $S_2$  représentent les situations où le joueur 2 joue.

Une arête du graphe représente un coup que l'un ou l'autre des joueurs peut effectuer.

**Remarques 1.** Le programme se limite aux **jeux d'accessibilité** pour lesquels :

- les **graphes** sont supposés **acycliques**, et donc toute partie est finie ;
- les **conditions de victoire** sont déterminées par les **positions (sommets) sans successeurs** : la partie est terminée et on sait si un joueur a gagné ou s'il y a match nul.

### 1.2.c) Graphe du jeu de Nim

Pour le jeu de Nim, avec  $n$  bâtonnets au départ, on notera  $(k, i)$  la situation où le joueur  $i$  joue, avec  $k$  bâtonnets placés devant lui, et l'on aura donc :

$$S_1 = \{(k, 1) | 0 \leq k \leq n\}$$

$$S_2 = \{(k, 2) | 0 \leq k \leq n - 1\}$$

Si  $q$  est un élément de la règle (c'est-à-dire un nombre de bâtonnets que l'on peut retirer), et si  $k - q \geq 0$ , il y aura alors les arêtes suivantes (chaque joueur peut enlever  $q$  bâtonnets aux  $k$  bâtonnets présents, l'autre joueur se retrouve alors avec  $k - q$  bâtonnets) :

$$(k, 1) \longrightarrow (k - q, 2)$$

$$(k, 2) \longrightarrow (k - q, 1)$$

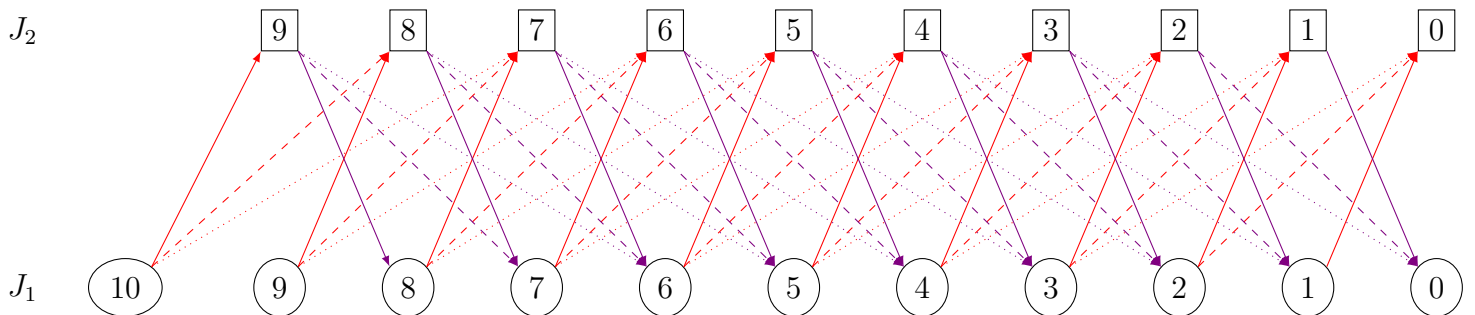


FIGURE 2 – Graphe biparti du jeu des bâtonnets, en partant de  $n = 10$  bâtonnets, et la règle autorise le retrait de 1, 2 ou 3 bâtonnets.

### 1.3 Stratégies et positions gagnantes



#### Définition : Partie

Une partie est un chemin dans le graphe, c'est-à-dire une suite  $(s_n)$  de positions telle que pour tout  $i$ ,  $(s_i, s_{i+1}) \in A$ .

On suppose qu'il existe des parties  $V_1$  et  $V_2$  de  $S$ , non vides et disjointes, qui représentent respectivement les ensembles de sommets où  $J_1$  et  $J_2$  gagnent la partie.

### Définition : Partie gagnante

Une partie est dite gagnante pour  $J_1$  si sa dernière position appartient à  $V_1$ , et si aucune de ses autres positions n'appartient à  $V_1 \cup V_2$ .  
 Une partie est dite gagnante pour  $J_2$  si sa dernière position appartient à  $V_2$ , et si aucune de ses autres positions n'appartient à  $V_1 \cup V_2$ .

Rq : Un match nul est alors une partie se terminant en une position (pas dans  $V_1 \cup V_2$ ) de laquelle ne part aucune arête.

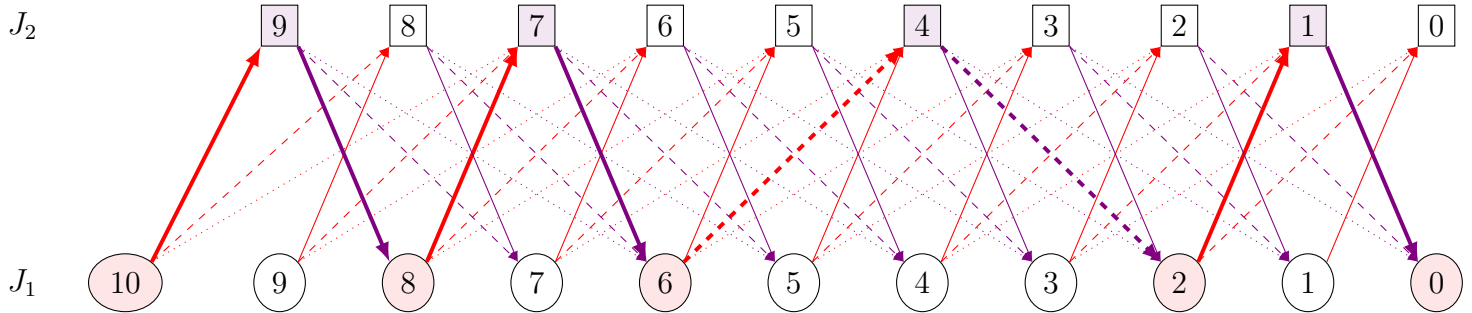


FIGURE 3 – Exemple d'une partie remportée par  $J_1$ .

### Définition : Stratégie gagnante

On dit d'une **stratégie** qu'elle est **gagnante** pour le joueur  $J$  à partir d'un état  $S$  (sommet du graphe de jeu), si elle permet à  $J$  de gagner à coup sûr en partant de  $S$ .

### Définition : Position gagnante

On dit d'un sommet  $S$  que c'est une **position gagnante** s'il existe une stratégie gagnante partant de  $S$ .

Les positions de  $V_1$  sont bien évidemment gagnantes pour  $J_1$ . Nous allons voir comment calculer les autres positions gagnantes.

**Exercice de cours A** Déterminer les positions gagnantes pour  $J_1$ .

## 1.4 Calcul des positions gagnantes

### 1.4.a) Attracteurs

Dans un jeu d'accessibilité, **résoudre le jeu consiste essentiellement à calculer les positions gagnantes de chacun des joueurs**, et une **stratégie pour chacun** (définie sur chaque position gagnante que le joueur contrôle). On se concentre d'abord sur les positions gagnantes du premier joueur, qui doit, pour gagner, atteindre un ensemble  $V_1$ . L'outil adéquat est l'attracteur.

**Ensemble définissant l'attracteur.** On se donne un graphe biparti (arène)  $G = (S, A)$ , supposé acyclique et deux ensembles  $S_1$  et  $S_2$  formant une partition de  $S$ , et  $V_1$  l'ensemble de sommets définissant la condition de victoire du premier joueur. Soit  $x$  une position du graphe.

- Si  $x \in S_1$  et n'a pas de successeur, alors  $x$  est une position gagnante pour le joueur  $J_1$ .
- Si  $x \in S_1$ , et si il existe au moins une arête  $(x, y) \in A$  telle que  $y$  est une position gagnante pour  $J_1$ , alors  $x$  est aussi une position gagnante pour  $J_1$ . Une stratégie gagnante de  $J_1$  est de jouer sur  $y$ .
- Si  $x \in S_2$ , et si pour toutes les arêtes  $(x, y) \in A$  partant de  $x$ ,  $y$  est une position gagnante pour  $J_1$ , alors  $x$  est aussi une position gagnante pour  $J_1$ , quelque soit la stratégie de  $J_2$  en  $x$ ,  $J_1$  possède une stratégie gagnante pour les coups suivants.

On peut donc calculer les positions gagnantes par récurrence, à partir des positions de victoire  $V_1$  pour  $J_1$ , et de  $V_2$  pour  $J_2$ .

### Définition : Suite des ensembles attracteurs

On note  $\mathcal{A}_m^i$  l'ensemble des positions gagnantes pour le joueur  $i$  en au plus  $m$  coups.

On définit pour le joueur 1 :

$$\begin{cases} m = 0 & \mathcal{A}_0^1 = \left\{ \text{positions gagnantes pour } J_1 \text{ en zéro coup} \right\} \\ \forall m \in \mathbb{N} & \mathcal{A}_{m+1}^1 = \mathcal{A}_m^1 \cup \left\{ \begin{array}{l} \text{sommets de } J_1 \text{ permettant d'aller vers } \mathcal{A}_m^1 \\ \cup \\ \text{sommets de } J_2 \text{ allant nécessairement vers } \mathcal{A}_m^1 \end{array} \right\} \end{cases}$$

$$\begin{cases} m = 0 & \mathcal{A}_0^1 = V_1 \\ \forall j \in \mathbb{N} & \mathcal{A}_{m+1}^1 = \mathcal{A}_m^1 \cup \left\{ \begin{array}{l} x \in S_1 \mid \exists (x, y) \in A, y \in S_2 \cap \mathcal{A}_m^1 \\ \cup \\ x \in S_2 \mid \forall (x, y) \in A \Rightarrow y \in \mathcal{A}_m^1 \end{array} \right\} \end{cases}$$

En français, l'ensemble  $\mathcal{A}_{m+1}^1$  des positions gagnantes en au plus  $m + 1$  coups est constitué de :

- l'ensemble  $\mathcal{A}_m^1$  : des positions gagnantes en au plus  $m$  coups ;
- des sommets contrôlés par  $J_1$  pour lesquels **il existe au moins** un arc permettant de rejoindre une position gagnante de  $\mathcal{A}_m^1$  contrôlée par  $J_2$  ;
- des sommets contrôlés par  $J_2$  dont **tous** les arcs aboutissent à une position gagnante de  $J_1$  des sommets de  $\mathcal{A}_m^1$  (sommets contrôlés par  $J_2$  qui font obligatoirement aboutir à une position gagnante de  $S_1$ ).

On arrive alors petit à petit à identifier toutes les positions gagnantes de  $J_1$  jusqu'au sommet initial.

### Exercice de cours B

Q1. Représenter les différentes étapes de l'évolution de l'attracteur du joueur  $J_1$  pour le graphe de la figure 2 (reproduit ci-dessous figure 4).

Q2. Que peut-on dire de la position de départ pour  $J_1$  ?

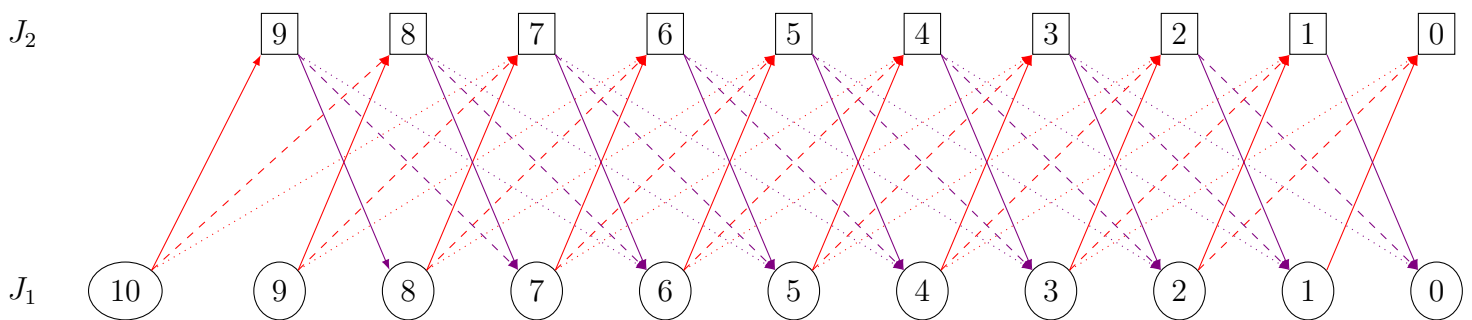


FIGURE 4 – Détermination de l'attracteur de  $J_1$

### Définition : Attracteur

L'**attracteur** du joueur  $i$  est l'ensemble des positions gagnantes pour le joueur  $i$ , soit :

$$\mathcal{A}^i = \bigcup_{j=0}^{+\infty} \mathcal{A}_j^i$$

L'attracteur  $\mathcal{A}^i$  du joueur  $J_i$  contient exactement toutes les positions gagnantes de  $J_i$ . Les autres positions sont celles conduisant a priori à un match nul, si les deux joueurs jouent de manière optimale. Le calcul d'un attracteur permet de déterminer les positions gagnantes pour un joueur puis de construire une stratégie gagnante.

#### 1.4.b) Algorithme de calcul de l'attracteur

##### i- Algorithme

#### 💡 Méthode : Calcul de l'attracteur de $J_1$

- (1) On calcule les antécédents de chaque position (ce qui revient à déterminer le graphe obtenu à partir du graphe du jeu, et dont les flèches sont « renversées »).
- (2) On calcule les degrés sortants de chaque position, que l'on affecte à une structure de données. Cette structure comptera le nombre de successeurs de chaque sommet qui ne sont pas des sommets gagnants pour  $J_1$ . Lorsque ce compteur atteindra 0 pour un sommet, on saura alors que ce sommet sera gagnant pour  $J_1$ .
- (3) On crée un attracteur vide (cela peut-être un dictionnaire).
- (4) On commence le calcul sur les positions de victoire de  $J_1$ , que l'on sait être des sommets gagnants. Pour chaque sommet que l'on sait être gagnant : si le sommet n'a pas déjà été rencontré, donc s'il n'est pas déjà dans l'attracteur que l'on est en train de construire :
  - a) on ajoute ce sommet à l'attracteur
  - b) pour chaque prédécesseur de ce sommet :
    - i. on retranche 1 à son compteur de successeurs non gagnants ;
    - ii. si ce prédécesseur est dans  $S_1$ , il est gagnant, on déclenche donc un appel récursif dessus ;
    - iii. sinon, si ce prédécesseur n'a plus que des successeurs gagnants (ce qui se lit dans le compteur associé), il est gagnant, on déclenche donc un appel récursif dessus.

##### ii- Représentation du graphe

L'arène de jeu (c'est-à-dire le graphe) est représentée par un dictionnaire dont les clés sont les sommets et les valeurs la liste des successeurs de ce sommet.

```

1 # Création du graphe du jeu de Nim étudié précédemment
2 arene= {(0,1):[], (0,2):[], (1,1):[(0,2)], (1,2):[(0,1)],
3         (2,2):[(1,1),(0,1)], (2,1):[(1,2),(0,2)], ...,
4         (10,1):[(7,2),(8,2),(9,2)]}
5 # Sommets contrôlés par J1 et J2
6 Sommets1 = {(k,1) : True for k in range(11)}
7 Sommets2 = {(k,2) : True for k in range(10)}
8 # Positions gagnantes de J1 et J2
9 Victoire1 = {(0,1) : True} # J1 ne peut plus jouer, J2 a enlevé le dernier
10 Victoire2 = {(0,2) : True}

```

##### iii- En Python

On commence par écrire la fonction d'en-tête `transpose(G;dict)->dict` (❤️) qui prend en entrée un graphe représentant l'arène de jeu, représenté sous la forme d'un dictionnaire des successeurs.

```

1 def transpose(G):
2     Gt={s:[] for s in G} # initialisation du graphe de la transposée avec
3     toutes les clés de G
4     for s in G: # pour chaque sommet de G
5         for ..... in ..... : # parcours des successeurs v de s
6             Gt[v].append(...) # s est un prédécesseur de v : ajout de s à
7             Gt[v]
8     return Gt

```

Et la fonction d'en-tête `degres_sortants(G:dict)->dict` (♥) qui prend en entrée le graphe `G` de l'arène de jeu et renvoie un dictionnaire des degrés sortants des sommets de `G` (c'est-à-dire les degrés entrants des sommets de la transposée du graphe `G`).

```
1 def degres_sortants(G):
2     deg_sort={} # dictionnaire des degrés sortants de G
3     # calcul du degré sortant de chaque sommet de G
4     .....
5     .....
6     return .....
```

Puis on écrit la fonction d'en-tête `attracteur(G:dict,S1:dict,V1:dict)->dict` qui prend en entrée le graphe `G` du jeu (sous forme d'un dictionnaire), le dictionnaire `S1` des sommets contrôlés par  $J_1$ , le dictionnaire `V1` des positions de victoire de  $J_1$  et renvoie `A` l'attracteur de  $J_1$ , sous la forme d'un dictionnaire dont les clés sont les positions gagnantes de  $J_1$ .

```
1 def attracteur(G,S1,V1):
2     """
3     G : dictionnaire des successeurs (graphe de l'arène du jeu)
4     S1 : dictionnaire des sommets contrôlés par J1
5     V1 : dictionnaire des positions gagnantes de J1
6     Renvoie : A, dictionnaire de l'attracteur, les sommets de l'attracteur
7     sont marqués à True
8     """
9     n=len(G)
10    deg_succ = ..... # degrés sortant de G
11    tG = ..... # graphe transposé
12    A={} # attracteur : dictionnaire des positions gagnantes
13    def parcours(s): # sous-fonction récursive qui cherche les positions de
14    victoire pour J1 en remontant le graphe (donc en parcourant la transposée)
15    à partir du sommet s
16        if ..... : # s n'est pas dans l'attracteur
17            ..... # on met le sommet dans l'attracteur
18            for ..... : # parcours des
19            successeurs v de s de tG (donc des prédécesseurs de s dans G)
20                ..... # s étant gagnant il y a un
21            successeur de v non gagnant en moins
22                if ..... or ..... : # si le
23            prédécesseur est dans les positions contrôlées par J1, ou s'il n'a plus
24            de prédécesseur, v est donc une position gagnante
25                ..... # on cherche les prédécesseurs de v
26            qui amènent à une position de victoire
27            for ..... : # on part d'une position de victoire de J1
28                ..... # on cherche les prédécesseurs de s
29            return A
```

Pour le graphe étudié précédemment :

```
1 >>> attracteur(arene,Sommets1,Victoire1)
2 {(0, 1): True, (1, 2): True, (2, 1): True, (3, 1): True, (4, 1): True, (5,
3     2): True, (6, 1): True, (7, 1): True, (8, 1): True, (9, 2): True, (10, 1):
4     True}
```

## I.5 Stratégie gagnante

Pour déterminer le coup à jouer sur une position gagnante, il ne suffit pas de jouer sur une autre position gagnante, il faut jouer sur une position gagnante plus proche d'une condition de victoire ( $V_1$  pour le joueur  $J_1$ ).

Pour cela, lors du calcul de l'attracteur, on numérote les sommets gagnants, en partant de la valeur 0 sur la condition de victoire, et on l'incrémente de 1 à chaque appel récursif. Si un sommet gagnant est ainsi étiqueté avec la valeur  $m \in \mathbb{N}$ , cela signifie que son rang est inférieur ou égal à  $m$ , et que le joueur pourra gagner la partie en moins de  $m$  coups.

La nouvelle fonction d'en-tête `attracteur_strat(G:dict,S1:dict,V1:dict)->dict` renvoie un dictionnaire A des attracteurs de telle sorte que si s est une position gagnante, alors A[s] est un entier m qui indique que le joueur pourra gagner la partie en moins de m coups. Par exemple :

```
>>> attracteur_strat(arene,Sommets1,Victoire1)
{(0, 1): 0, (1, 2): 1, (2, 1): 2, (3, 1): 2, (4, 1): 2, (5, 2): 3, (6, 1):
 4, (7, 1): 4, (8, 1): 4, (9, 2): 5, (10, 1): 6}
```

```
def attracteur_strat(G,S1,V1):
    n=len(G)
    deg_succ=degres_sortants(G) # degrés sortant de G
    tG=transpose(G) # graphe transposé
    A={}
    def parcours(s,m): # cherche les positions de victoire pour J1 en
remontant le graphe à partir de s
        if s not in A: # mémoire
            A[s]=m # on met le sommet dans l'attracteur
            for v in tG[s]: # pour les prédécesseurs de s dans G
                deg_succ[v]=deg_succ[v]-1 # s étant gagnant il y a un
successeur de v non gagnant en moins
                if v in S1 or deg_succ[v]==0: # si le précédesseur est dans
les positions contrôlées par J1, ou s'il n'a pas de précédesseur non
gagnant
                    parcours(v,m+1) # on cherche les prédécesseurs qui
amènent à une position de victoire en un coup supplémentaire
    for s in V1: # on part d'une position de victoire de J1
        parcours(s,0) # on cherche les prédécesseurs de s qui amènent à s
    return A
```

```
def strategie(G,attr1,attr2,s):
    """
    crée la stratégie pour le joueur J1
    attr1 : attracteur de J1 ; attr2 : attracteur de J2
    s : une position contrôlée par J1
    """
    if s in attr1: # fait partie de l'attracteur de J1
        for x in G[s]: # on parcourt les sommets accessibles depuis s
            if x in attr1 and attr1[x]<attr1[s]: # sommet de l'attracteur de
J1 qui permet d'atteindre la victoire en moins de coups
                return x
    elif s in attr2: # fait partie de l'attracteur de J2
        i=randint(0,len(G[s])-1) # choix d'un sommet quelconque de l'arène
accessible depuis s
        return G[s][i]
    else: # renvoie un successeur de s qui n'est pas dans l'attracteur de J2
        for x in G[s]:
            if x not in attr2:
                return x
```



```

1 A1=attracteur_strat(arene , S1 , V1)
2 A2=attracteur_strat(arene , S2 , V2)
3 >>> strategie(arene , A1 , A2 , (10 , 1))
4 (9 , 2)

```

## II Heuristique. Algorithme Min-Max

La méthode des attracteurs peut devenir très lente puisqu'il faut itérer à chaque fois sur tout le graphe restant pour tester les sommets n'étant pas encore dans un attracteur. Elle devient totalement inadaptée pour des jeux plus complexes. Les échecs par exemple peuvent être représentés par un graphe d'environ  $10^{50}$  nœuds. Ce nombre monte à  $10^{100}$  pour le jeu de go!

Les intelligences artificielles comme Stockfish (aux échecs) ou AlphaGo (jeu de go) fonctionnent par anticipation des prochains coups envisageables. Même si elles peuvent imaginer les issues vraisemblables de nombreux tours en avance, il leur est toujours impossible d'être exhaustif quant aux situations possibles.

### II.1 Heuristique

#### II.1.a) Définition



#### Définition : Heuristique

Pour un joueur  $J$ , il peut être utile d'évaluer l'avantage que lui donne une certaine position  $p$  par un nombre. On va donc avoir besoin d'une fonction :

$$h : \begin{cases} \text{positions sur le graphe} & \longrightarrow \mathbb{R} \\ p & \longrightarrow h(p) \end{cases}$$

de sorte que plus  $h(p)$  est grand, plus la position  $p$  est avantageuse pour  $J$ . Une telle fonction est dite **heuristique**, ou d'évaluation.

#### REMARQUES



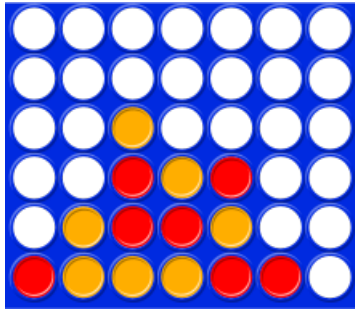
- Une position gagnante pour  $J$  devrait renvoyer  $+\infty$ , puisque ce nœud est infiniment intéressant (permet de gagner à coup sûr).
- Dans un jeu à deux joueurs, une position gagnante pour l'adversaire doit renvoyer  $-\infty$ .

La recherche d'une bonne heuristique est une tâche difficile. Pour certains jeux une heuristique est obtenue par un apprentissage supervisé d'une intelligence artificielle.

La notion d'heuristique a été vue en première année pour l'algorithme  $A^*$  (comme amélioration de l'algorithme de Dijkstra).

#### II.1.b) Exemple : puissance 4

Le but du jeu du puissance 4 est d'aligner une suite de quatre pions de même couleur sur une grille comptant six rangées et sept colonnes. Tour à tour, les deux joueurs placent un pion dans la colonne de leur choix, le pion coulisse alors jusqu'à la position la plus basse possible dans la dite colonne à la suite de quoi c'est à l'adversaire de jouer. Le vainqueur est le joueur qui réalise le premier un alignement (horizontal, vertical ou diagonal) consécutif d'au moins quatre pions de sa couleur. Si, alors que toutes les cases de la grille de jeu sont remplies, aucun des deux joueurs n'a réalisé un tel alignement, la partie est déclarée nulle.



3	4	5	7	5	4	3
4	6	8	10	8	6	4
5	8	11	13	11	8	5
5	8	11	13	11	8	5
4	6	8	10	8	6	4
3	4	5	7	5	4	3

FIGURE 6 – Les valeurs de chaque case pour l’heuristique utilisée

FIGURE 5 – Une position de puissance 4

Une heuristique simple consiste à attribuer à chaque case une valeur, par exemple le nombre d’alignements potentiels de quatre pions lorsqu’on place un pion à cet emplacement, puis à sommer les cases occupées (positivement pour les pions de  $J_1$ , négativement pour ceux de  $J_2$ ).

Par exemple, si on convient que les pions jaunes sont ceux de  $J_1$ , la valeur de l’heuristique de la position présentée figure 5 est égale à  $4 + 5 + 7 + 6 + 8 + 13 + 11 - 3 - 5 - 4 - 8 - 10 - 11 - 11 = 2$ .

L’heuristique sera égale à  $+\infty$  pour une position gagnante pour  $J_1$ , et à  $-\infty$  pour une position gagnante pour  $J_2$ .

Au moment où l’un des deux protagonistes doit jouer, plusieurs possibilités s’offrent à lui (entre une et sept pour le puissance 4). Une solution simple pour choisir le coup à jouer consiste à calculer l’heuristique correspondant à chacune des configurations atteignables et à jouer celle d’heuristique maximale (pour  $J_1$ ) ou minimale (pour  $J_2$ ).

## II.2 Algorithme Min-Max avec heuristique

### II.2.a) Principe

#### 💡 Méthode : Algorithme min-max

Lorsque  $J_1$  doit jouer, il construit un arbre d’évolution du jeu, à partir du nœud auquel il se trouve. Il itère cela jusqu’à une profondeur  $k$  (nombre de coups d’avance).

Ensuite  $J_1$  procède en partant de la dernière ligne de l’arbre et en remontant petit à petit, tout en évaluant chacun des nœuds de la manière suivante :

- une position contrôlée par  $J_2$  prend comme valeur le minimum des scores lui succédant ;
- une position contrôlée par  $J_1$  prend comme valeur le maximum des scores lui succédant ;
- s’il n’y a pas de nœud succédant, garde la valeur donnée par l’heuristique.

Une fois en haut de l’arbre,  $J_1$  joue le coup ayant le plus grand score.

### II.2.b) Exemples simples de mise en œuvre

Premier exemple Nous reprenons le jeu de Nim précédent avec un tas initial contenant cinq objets.

On construit l’arbre d’évolution de la partie pour une profondeur  $k = 4$  et on remplit les lignes une à une.

$$\forall p, h(p) = \begin{cases} +\infty & \text{si } p = \boxed{1}_{J_2} \\ -\infty & \text{si } p = \ominus_{J_1} \\ 0 & \text{sinon} \end{cases}$$

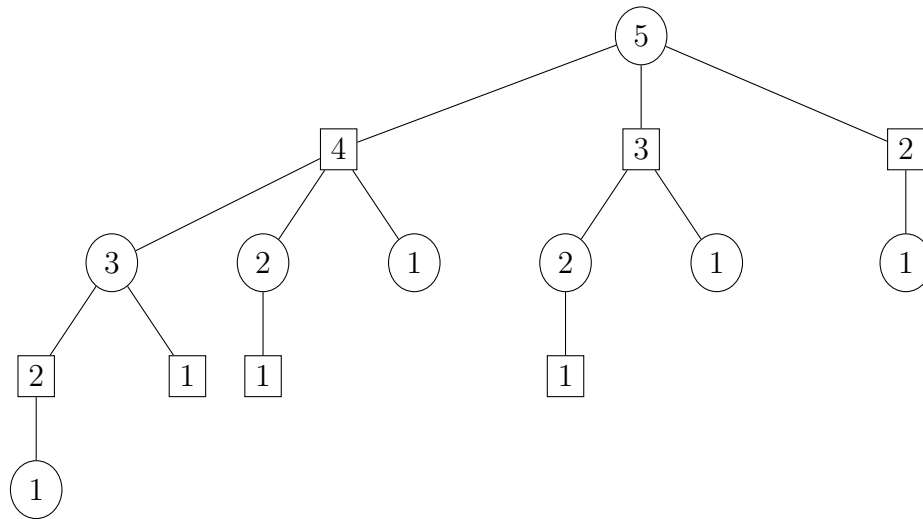


FIGURE 7 – Arbre de jeu.

Les ronds sont les positions contrôlées par  $J_1$  et les carrés sont celles contrôlées par  $J_2$ .

### Définition : Arbre

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 nœuds à une même profondeur sont représentés sur une même ligne horizontale.

Un nœud sans successeur est appelé une **feuille**, sinon c'est un **nœud interne**.

### REMARQUES

— L'exemple ci-dessus est extrêmement simplifié.

L'utilisation de la méthode minimax par rapport à celle des attracteurs n'est techniquement pas justifiée. Cet algorithme manque d'intérêt dans ce jeu, où chaque nœud est soit gagnant, soit perdant.

— On voit bien que l'utilisation des fonctions Min et Max permet de simuler une partie où chaque joueur · euse joue toujours son coup optimal.

— On remarque une certaine redondance dans les racines de l'arbre. L'utilisation de la programmation dynamique est tout à fait bienvenue ici !

Traisons un autre exemple

- Dans l'exemple de la figure 8,  $J_1$  choisira le coup le plus à gauche, correspondant à une évaluation égale à 10.

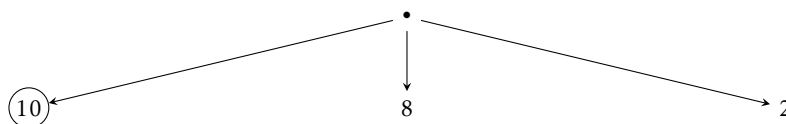


FIGURE 8 –  $J_1$  doit jouer, 3 coups sont possibles, il joue sur l'heuristique maximale.

- Mais  $J_1$  peut aussi tenir compte du coup que va jouer  $J_2$  ensuite, et donc calculer l'heuristique de chacune des positions que  $J_2$  pourra atteindre. Si on observe la figure 9, on constate qu'il vaut mieux pour  $J_1$  de jouer le coup central, en partant du principe que  $J_2$  joue au mieux son coup.

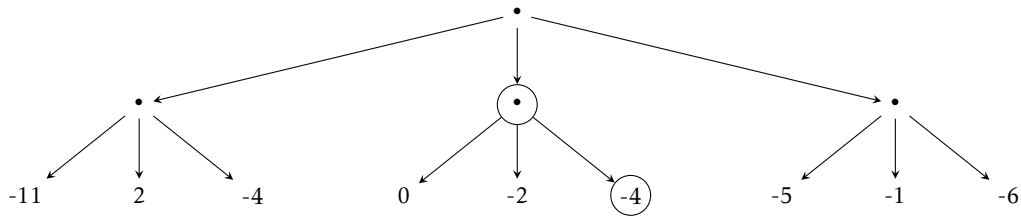


FIGURE 9 – C'est à  $J_1$  de jouer, en tenant compte du coup suivant de  $J_2$ .

- Bien évidemment, on peut réitérer ce raisonnement et tenir compte du coup suivant, joué cette fois par  $J_1$ . La figure 11 montre qu'en tenant compte des deux coups suivants,  $J_1$  a en fait intérêt à jouer le coup le plus à droite.

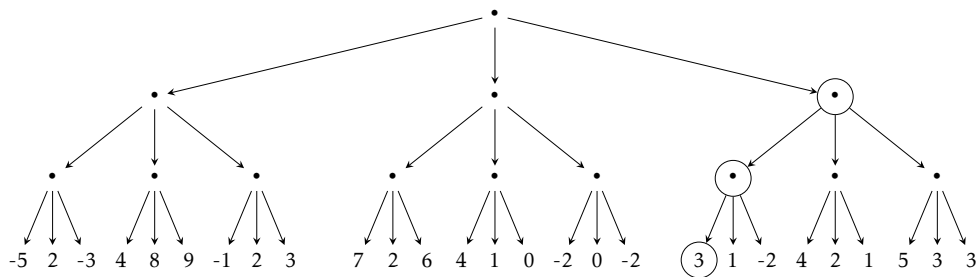


FIGURE 10 – Le meilleur coup de  $J_1$ , en tenant compte des deux coups suivants.

- On peut répéter ce raisonnement, mais le nombre de configurations à examiner ayant tendance à croître exponentiellement, il est nécessaire de limiter la profondeur de la recherche

Pour calculer le meilleur coup de  $J_1$ , il faut donc :

- Commencer par calculer la valeur de l'heuristique de toutes les positions atteignables en  $n$  coups : les feuilles de l'arbre.
- Si  $n$  est pair,  $J_2$  aura joué le dernier coup : le père de chacune de ces feuilles se verra donc attribuer le minimum des valeurs de ses fils.
- À l'inverse, si  $n$  est impair le père de chacune de ces feuilles se verra attribuer la valeur maximale de ses fils (car  $J_1$  aura joué en dernier). Ainsi, de proche en proche chaque position de l'arbre se verra attribuer une valeur.

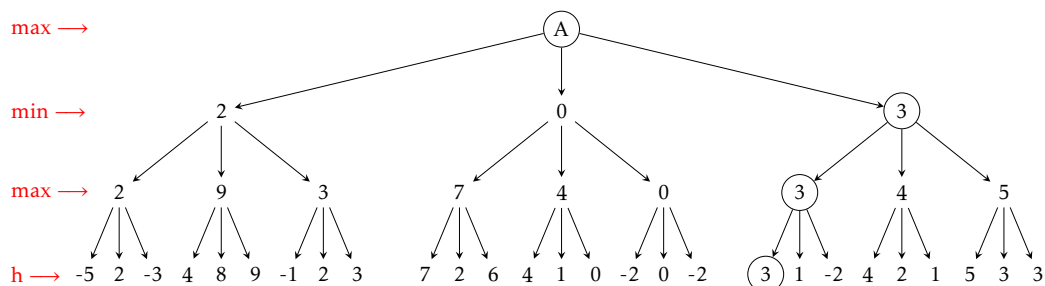
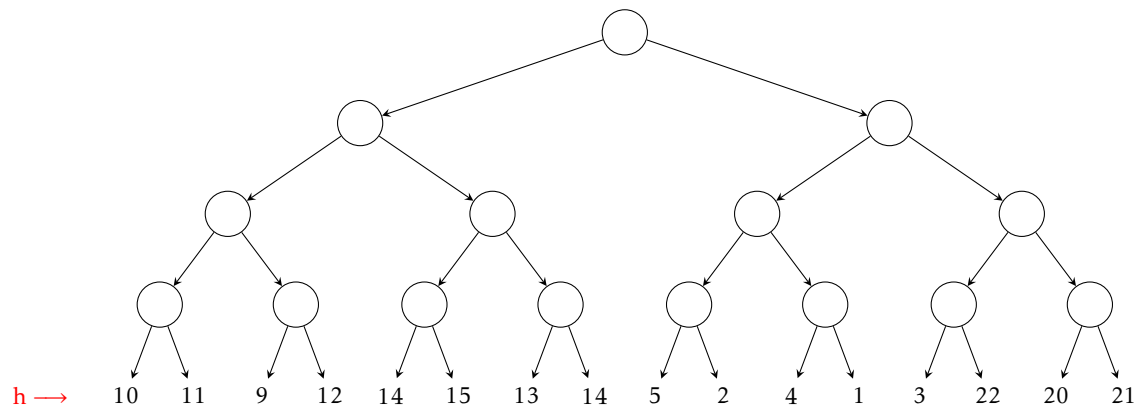


FIGURE 11 – Le résultat de l'algorithme min-max à une profondeur 3 où  $J_1$  doit jouer

### Exercice de cours C

Calculer la valeur de la position associée à l'arbre ci-dessous, dans le cas où c'est le joueur qui cherche à maximiser l'heuristique qui doit jouer.



### II.2.c) Écriture de l'algorithme

On écrit la fonction `minimax(G:dict,p:tuple,h:funct,prof:int)` qui à partir du graphe de `G`, d'une position `p` du graphe (valeur de la position, numéro du joueur qui contrôle la position), de la fonction `h` de l'heuristique, et d'une profondeur `prof` renvoie le maximum de l'heuristique en considérant `prof` coups.

```

1 def minimax(G,p,h,prof):
2     """
3     Arguments :
4     G : dictionnaire du graphe (comme précédemment)
5     p : position de départ (un noeud de l'arbre)
6     h : fonction heuristique : h(p) donne la valeur associée à la position p
7     selon la fonction h
8     prof : profondeur
9     """
10    if prof==0 or G[p]==[]: #
11        return h(p)
12    elif p[1]==1: # J1 contrôle le noeud
13        # on cherche le maximum des sous-arbres
14        return max([minimax(G,n,h,prof-1) for n in G[p]])
15    else : # J2 contrôle le noeud
16        # on cherche le minimum des sous-arbres
17        return min([minimax(G,n,h,prof-1) for n in G[p]])

```

Remarque. Pour établir la stratégie d'un joueur il suffit de retenir le mouvement qui a permis de minimiser ou maximiser la valeur à la première étape.

**Remarques 2.** Une technique courante pour construire cette heuristique (évaluation d'une position) est une méthode de Monte-Carlo : pour chaque position, on effectue un grand nombre de parties en faisant jouer les deux protagonistes au hasard (en supposant les parties tout de même assez courtes) et on attribue une valeur qui est la moyenne des gains des parties ainsi engendrées (en n'utilisant plus  $\pm\infty$ ). Le programme AlphaGo qui a battu un grand maître de Go en 2016 a dans ses premières versions utilisé cette méthode.