Cours - Théorie des jeux

Dans ce cours nous allons nous intéresser à l'étude théorique et algorithmique de jeux à deux joueurs antagonistes jouant alternativement, joueurs que l'on dénomme traditionnellement Adam et Ève. Les jeux qui nous intéressent sont à information totale : **à tout instant d'une partie chacun des joueurs a une vision complète de l'état du jeu.** Ceci exclus la plus-part des jeux de cartes (on ne connait pas le jeu de l'adversaire) mais inclus des jeux tels les échecs, les dames, le go, etc. Dans un premier temps, nous allons nous intéresser à des jeux « simples » pour lesquels il est possible de déterminer (au moins pour de petites configurations) une stratégie gagnante, puis nous verrons, pour les jeux les plus complexes, comment bâtir une stratégie à l'aide d'une heuristique.

I Jeux sur un graphe

I.1 Le jeu de Chomp

Le premier jeu auquel nous allons nous intéresser se joue à l'aide d'une tablette de chocolat rectangulaire dont le coin supérieur gauche est empoisonné : Chaque joueur choisit à tour de rôle un carré et le mange, ainsi que tous les morceaux situés à la droite et en dessous du carré choisi. Bien évidemment, le joueur qui n'a plus d'autre choix que de manger le carré empoisonné a perdu. On trouvera sur la figure suivante un exemple de partie perdue par Adam, qui a commencé à jouer en premier.

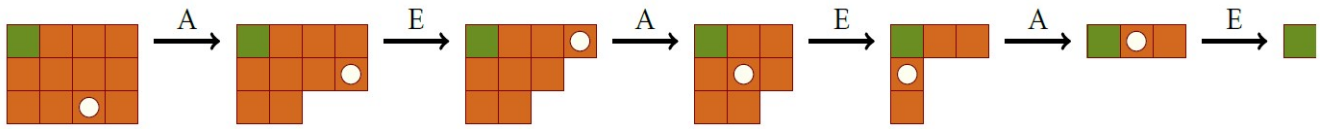


FIGURE 1 – Exemple de jeu de Chomp

À ce type de jeu est associé un graphe orienté (S, A) appelé *arène*. Chaque sommet de S représente une configuration du jeu (une *position*), l'une d'entre elles étant la *position de départ*. Une arête $a = (s1; s2) \in A$ reliant deux sommets indique la possibilité pour un joueur de passer de la position $s1$ à la position $s2$ en un coup. Par exemple, l'arène associée au jeu de Chomp pour une tablette (2, 3) est représentée figure 2. Pour visualiser une partie de Chomp, il suffit d'imaginer un jeton initialement posé sur la position initiale s_0. À tour de rôle, chaque joueur le déplace le long d'une arête issue de la position courante s et le pose sur un successeur de s. Une partie est donc un chemin d'origine s_0 dans l'arène. Ce jeu fait partie des jeux *d'accessibilité* : le graphe associé ne comporte pas de cycle (ce qui assure que toute partie est finie), et il est déterminé par un ensemble de positions (les *cibles*) qui sont sans successeurs. Ainsi, dans le jeu de Chomp le noeud du graphe associé au seul carré empoisonné est la seule cible du jeu, et l'atteindre signifie la fin de la partie (et la victoire pour celui qui l'atteint).

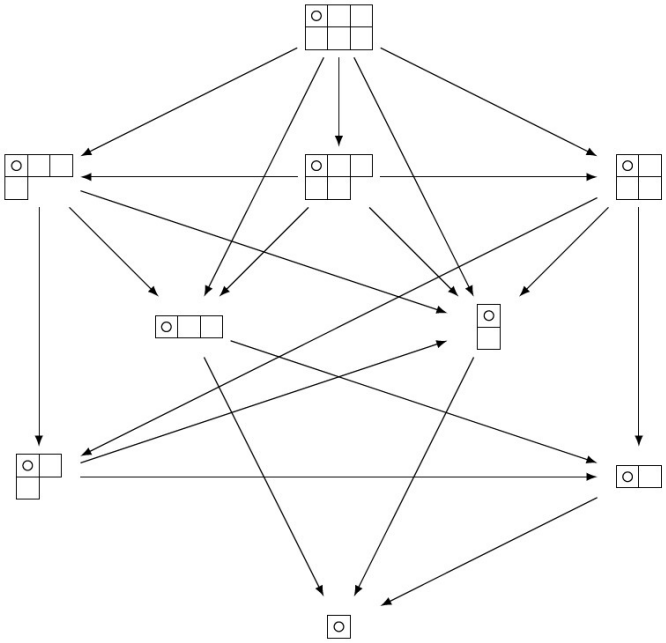


FIGURE 2 – Arène associée au jeu de Chomp (2,3)

Une fois fixé le joueur qui commence (Adam par exemple), on peut, en doublant chacun des sommets qu'on indexera par le nom du joueur, créer un nouveau graphe dont les sommets seront partitionnés en deux sous-ensembles $S = S_a \cup S_e$ avec $S_a \cap S_e = \emptyset$ où S_i est l'ensemble des positions à partir desquelles le joueur i jouera. Un tel graphe est dit *biparti* (illustration figure 3).

Dans un graphe biparti, les arêtes ne peuvent relier qu'un noeud de S_a à un noeud de S_e, et réciproquement.

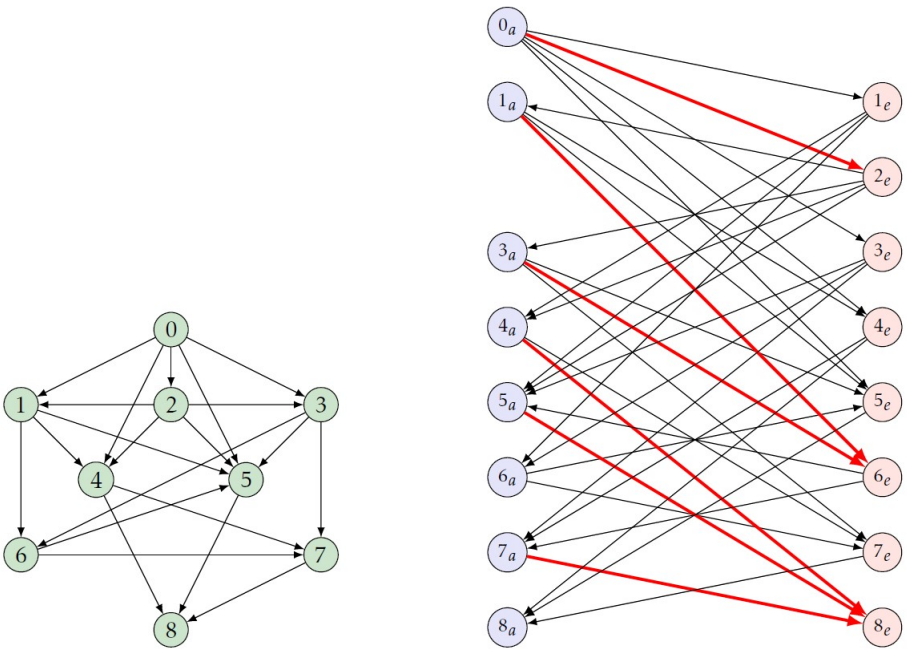


FIGURE 3 – Graphe et graphe biparti associés au jeu de Chomp(2,3)

Stratégies gagnantes :
Une stratégie pour Adam est définit par une application $f : S'_a \in S_a \to S_e$ tel que pour $s \in S'_a, (s, f(s))$ est une arête du graphe biparti.

Ainsi, de manière informelle, une stratégie consiste à déterminer, pour chaque position de S'_a, le mouvement suivant à jouer. Une partie $s_0 \to s_1 \to ... \to s_n$ est dite jouée suivant f lorsque pour tout $k \in [0, n-1]$, si $s_k \in S'_a$ alors $s_{k+1} = f(s_k)$. Une stratégie est dite *gagnante* pour Adam si toute partie jouée en suivant cette stratégie est gagnante pour Adam. On définit bien évidemment de manière analogue les stratégies gagnantes pour Ève.
Sur le graphe biparti de la figure 3 une stratégie gagnante pour Adam est mise en gras. Il doit commencer par poser le jeton sur le sommet 2, puis :

— si Ève joue son coup suivant sur 1 ou 3, poursuivre sur 6. Ève ne peut alors que suivre sur 5 ou 7, et Adam peut alors conclure en jouant sur 8 ;

— si Ève joue son coup suivant sur 4 ou 5, poursuivre (et terminer) sur 8.

Ève possède elle aussi une stratégie gagnante définie par $f(1_e) = f(3_e) = 6_a$ et $f(4_e) = f(5_e) = f(7_e) = 8_a$, mais comme elle ne joue pas en premier il est nécessaire qu'Adam ne joue pas sur le sommet 2 au début de la partie pour qu'Ève puisse suivre cette stratégie.

Positions gagnantes :
Une position s est dite *gagnante* lorsqu'il existe une stratégie gagnante pour une partie débutant au sommet s. Par exemple, pour le jeu de Chomp (2, 3) les positions 0, 1, 3, 4, 5, 7 sont gagnantes, les positions 2, 6 et 8 sont perdantes.

On remarque alors que dans le jeu de Chomp(p, q), il existe une stratégie gagnante pour le premier joueur dès lors que $(p, q) \neq (1, 1)$.

Si $p = 1$ ou $q = 1$, un coup suffit à Adam pour atteindre la position finale. Supposons maintenant $p > 1$ et $q > 1$, et faisons choisir à Adam le carré en bas à droite. Supposons que ce coup soit perdant ; dans ce cas Ève possède un coup gagnant. Mais tout mouvement choisi par Ève à cette étape du jeu aurait pu être choisi par Adam pour entamer la partie. Cela signifie qu'ou bien ce premier mouvement est gagnant, ou bien il en existe un autre qui soit gagnant. Dans tous les cas, Adam possède donc un coup gagnant.

Ce type de preuve est appelé un *vol de stratégie*. C'est malheureusement une preuve non constructive : savoir qu'une stratégie gagnante existe ne suffit pas à nous apprendre comment la trouver !

I.2 Positions gagnantes

Considérons un graphe orienté acyclique (S, A) associé à un jeu d'accessibilité. Il est possible de démontrer que dans un graphe acyclique il existe des sommets sans successeur.

Définition : Un sous-ensemble S' de sommets est dit *stable* si tout sommet de S' n'a aucun successeur dans S' . Un sous-ensemble S' de sommets est dit *absorbant* si tout sommet n'appartenant pas à S' possède au moins un successeur dans S' . Un sous-ensemble S' de sommets est un *noyau* s'il est à la fois stable et absorbant.

Par exemple, dans le jeu de Chomp (2, 3), les sommets (2,6,8) forment un noyau du graphe.

On peut démontrer que **tout graphe orienté et acyclique possède un unique noyau**. Raisonnons par récurrence sur le nombre n de sommets :

— si $n = 1$, l'unique sommet est aussi l'unique noyau du graphe ;

— si $n > 1$, supposons le résultat acquis jusqu'au rang $n - 1$. Par hypothèse, il existe au moins un sommet s sans successeur. Nécessairement, ce sommet doit appartenir à un éventuel noyau (car un noyau est absorbant). Considérons le graphe (S', A') obtenu en supprimant de (S, A) le sommet s ainsi que ces prédécesseurs.

  — Si (S', A') est le graphe vide, c'est que s est un noyau de (S, A) , et c'est le seul (tous les autres sommets admettent s comme successeur).

  — Dans le cas contraire, observons que (S', A') reste acyclique. Par hypothèse de récurrence il possède un unique noyau N' . Dans ce cas, $N = N' \cup s$ est un noyau de (S, A). Pour justifier qu'il est unique, considérons un noyau quelconque N de (S, A). Il doit contenir s, et N s est un noyau de (S', A'). Il est donc égal à N'.

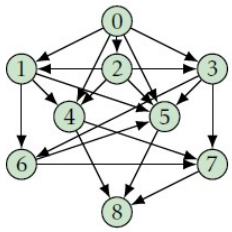
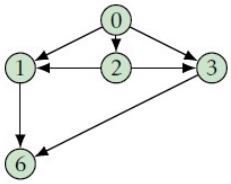
Cette preuve a le mérite de d'être constructive ; pour calculer le noyau de (S, A) il suffit de :

— chercher un sommet s de (S, A) sans successeur ;

— supprimer de (S, A) s et ses prédécesseurs ;

— recommencer tant qu'il reste des sommets.

Voici par exemple comment on calcule le noyau du jeu de Chomp (2 ; 3) :

	8 n'a pas de successeur ; il appartient au noyau et on le supprime ainsi que tous ses prédécesseurs.
	6 n'a pas de successeur ; il appartient au noyau et on le supprime ainsi que tous ses prédécesseurs.
	2 n'a pas de successeur ; il appartient au noyau et on le supprime ainsi que tous ses prédécesseurs.

Le noyau est donc (2,6,8)

Remarque : *Dans le cas du jeu de Chomp les éléments du noyau sont les positions perdantes et les autres sommets les positions gagnantes. La stratégie gagnante consiste, pour chaque sommet qui n'est pas dans le noyau, à jouer un coup qui s'y ramène.*

Algorithme du calcul du noyau :
On considère un graphe acyclique (S, A). Il est représenté en machine par la donnée d'un dictionnaire

d dont les clefs sont les sommets et les valeurs sont les successeurs des clefs (sous forme de liste). Par exemple, le graphe du jeu de Chomp (2,3) est représenté en mémoire par le dictionnaire suivant :

```
d = {
0: [1, 2, 3, 4, 5],
1: [4, 5, 6],
2: [1, 3, 4, 5],
3: [5, 6, 7],
4: [7, 8],
5: [8],
6: [5, 7],
7: [8],
8: [],
}
```

Q1. Proposer une fonction `sansSuccesseur(d)` qui renvoie un sommet sans successeur.
Q2. Proposer une fonction `predecesseurs(d,j)` qui renvoie la liste de tous les prédecesseurs du sommet j.
Q3. Proposer une fonction `supprimeSommet(d,j)` qui supprime un sommet j du graphe ainsi que tous les sommets possibles dans les listes associées.
Q4. En déduire une fonction `noyau(d)` qui renvoie la liste des sommets constituants le noyau de (S, A).

Remarque : *Cet Algorithme a une complexité en $O(n^3)$. Sachant que le jeu de Chomp(p, q) possède $n = \binom{p+q}{p} - 1$ sommets, le calcul devient rapidement très gourmand dès lors que p et q deviennent trop grands car pour p=q on a $n \sim \binom{2p}{p} \sim \frac{4^p}{\sqrt{\pi p}}$ soit une complexité exponentielle avec p*

II Algorithme min-max

Dans le cas d'un jeu à deux joueurs plus complexe, le calcul du noyau n'est pas possible. L'algorithme que nous avons écrit à une complexité en $O(n^3)$, où n est le nombre de sommets du graphe, autrement dit le nombre de positions que l'on peut rencontrer lors d'une partie. Cependant, cet entier n est souvent extrêmement grand : il est estimé de l'ordre de 10^{32} pour les dames, entre 10^{43} et 10^{50} pour le jeu d'échecs, de l'ordre de 10^{100} pour le jeu de go. Il devient donc nécessaire de s'appuyer non plus sur une évaluation exacte de la position, mais sur une estimation de la valeur de la position atteinte.

II.1 Heuristique

Dans la suite de cette section, nous supposerons posséder une fonction h qui à toute position légale p du jeu associe une valeur dans $\mathbb{R}$, de sorte que :

— plus $h(p)$ est grand, meilleure est la position pour Adam ;

— plus $h(p)$ est petit, meilleure est la position pour Ève.

Une telle fonction est appelée une *heuristique*.

Pour illustrer cette section, nous allons prendre l'exemple du Puissance 4 : le but du jeu 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.

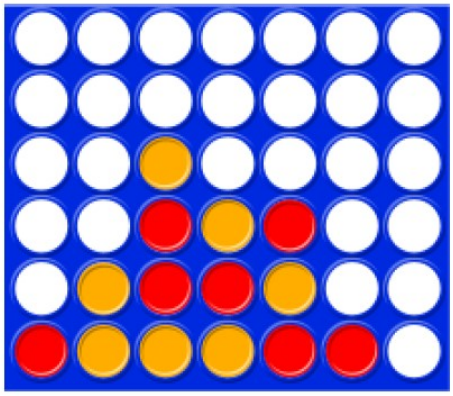


FIGURE 4 – Exemple de position de puissance 4

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 d'Adam, négativement pour ceux d'Ève).

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 5 – Valeur associées pour l'heursitique

Par exemple, si on convient que les pions jaunes sont ceux d'Adam, la valeur de l'heuristique de la position présentée figures 4 et 5 est égale à $4+5+7+6+8+13+11-3-5-4-8-10-11-11=2$. Évidemment, l'heuristique sera égale à $+\inf$ pour une position gagnante pour Adam, et à $-\inf$ pour une position gagnante pour Ève.

II.2 Min-Max

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 Adam) ou minimale (pour Ève).

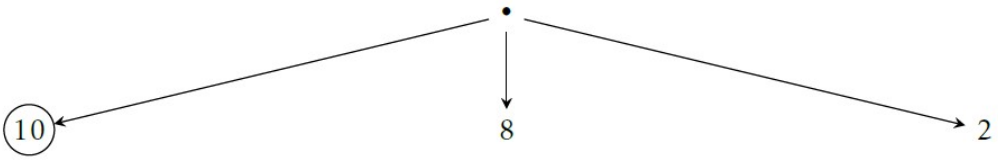


FIGURE 6 – 3 coups possibles pour Adam

Ainsi, dans l'exemple de la figure 6, Adam choisira le coup le plus à gauche, correspondant à une évaluation égale à 10. Mais Adam peut aussi tenir compte du coup que va jouer Ève ensuite, et donc calculer l'heuristique de chacune des positions qu'Ève pourra atteindre. Si on observe la figure 7, on constate qu'il vaut mieux pour Adam jouer le coup central, en partant du principe qu'Ève joue au mieux son coup.

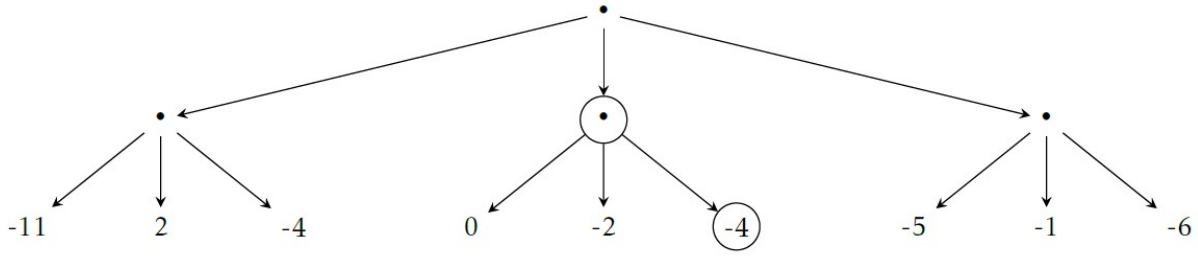


FIGURE 7 – A Adam de jouer, 3 coups possibles pour Adam

Bien évidemment, on peut réitérer ce raisonnement et tenir compte du coup suivant, joué cette fois par Adam. La figure 8 montre qu'en tenant compte des deux coups suivant, Adam a en fait intérêt à jouer le coup le plus à droite.

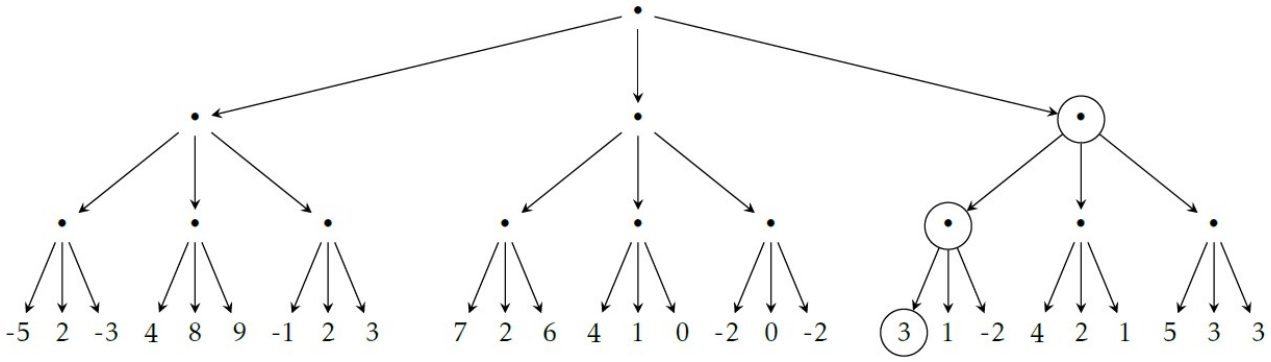


FIGURE 8 – C'est à Adam de jouer, en tenant compte du coup suivant d'Ève

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

L'algorithme de min-max :
Pour calculer le meilleur coup d'Adam, 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, Ève 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 Adam aura joué en dernier). Ainsi, de proche en proche chaque position de l'arbre se verra attribuer une valeur (illustration figure 9).

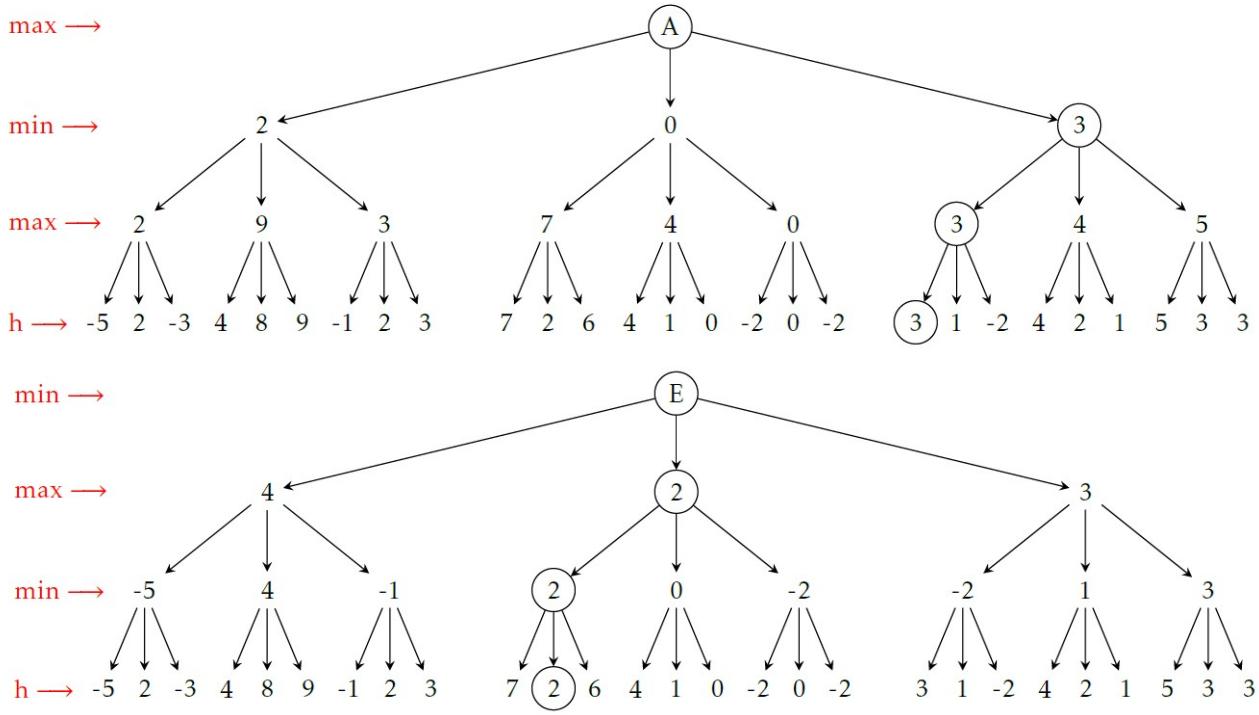


FIGURE 9 – Le résultat de l'algorithme min-max à une profondeur 2 (le premier arbre si c'est à Adam de jouer, le second si c'est à Ève).

Nous allons écrire deux fonctions :

— `maximin(p, n)` (destinée à Adam) va chercher à maximiser l'heuristique après n coups en partant de la position p, en supposant que son adversaire joue au mieux ;

— `minimax(p, n)` (destinée à Ève) va chercher à minimiser l'heuristique après n coups en partant de la position p, en supposant que son adversaire joue au mieux.

Ces deux fonctions sont mutuellement récursives : pour calculer `maximin(p, n)` on calcule pour chaque position $p_1, ..., p_k$ atteignable à partir de p la valeur de l'heuristique des positions `minimax(pi, n-1)` avant de choisir la position conduisant à la valeur maximale.

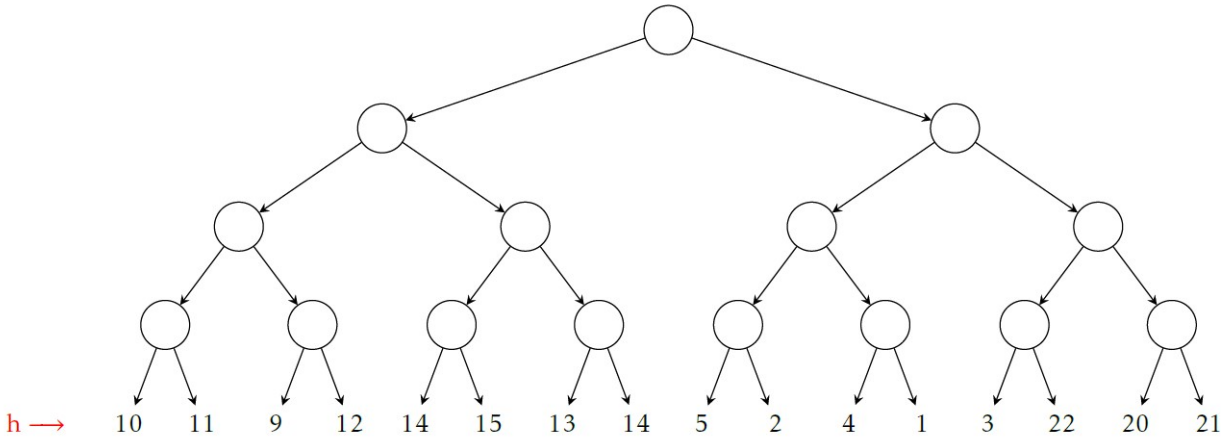
De manière symétrique, pour calculer `minimax(p, n)` on calcule pour chaque position $p_1, ..., p_k$ atteignable à partir de p la valeur de l'heuristique des positions `maximin(pi, n-1)` avant de choisir la position conduisant à la valeur minimale.

Pour la rédaction de l'algorithme, on suppose définie la fonction `h(p)` qui prend pour argument une position du jeu et renvoie la valeur de son heuristique, ainsi que la fonction `successeurs(p)` qui renvoie la liste des positions atteignables à partir de la position p.

```python
def minimax(p, n):
        if n == 0 or successeurs(p) == []:
                return h(p)
        mini = np.inf
        for pk in successeurs(p):
                s = maximin(pk, n - 1)
                if s < mini:
                        mini = s
        return mini
```

```python
def maximin(p, n):
        if n == 0 or successeurs(p) == []:
                return h(p)
        maxi = -np.inf
        for pk in successeurs(p):
                s = minimax(pk, n - 1)
                if s > maxi:
                        maxi = s
        return maxi
```

Q5. 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.

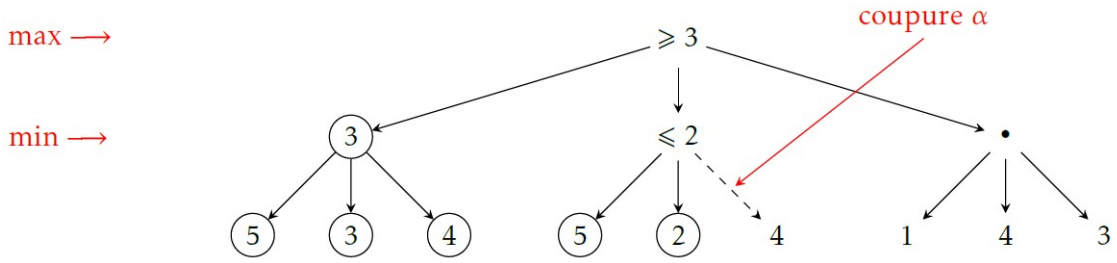


II.3 Elagage Alpha-beta (hors programme)

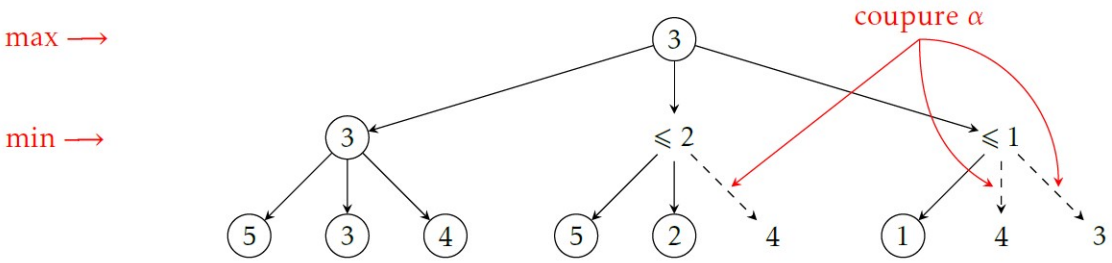
L'élagage alpha-beta est une amélioration de l'algorithme min-max qui vise à ne pas explorer certaines des branches de l'arbre dont on sait qu'elle n'interviendront pas dans l'évaluation de la position courante.

Coupure alpha :

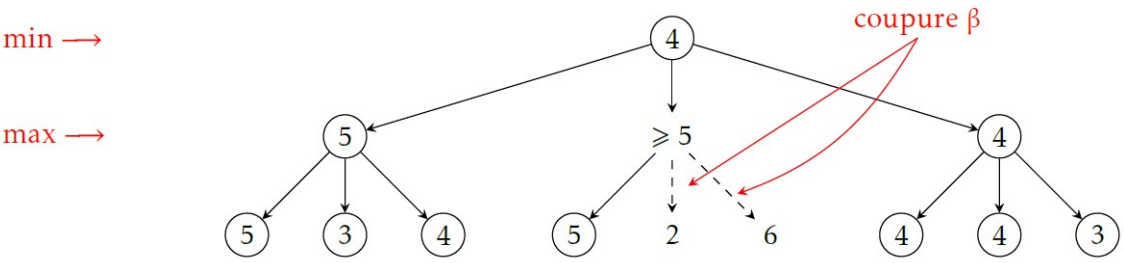
Considérons l'exemple ci-dessous, qui concerne une étape de calcul de minimum. L'exploration est en cours, les feuilles et noeuds qui ont été examinés sont marqués d'un cercle.



On constate qu'il n'est pas nécessaire de poursuivre l'exploration de la branche centrale : on sait déjà que sa valeur sera inférieure ou égale à 2 donc qu'elle ne sera pas choisie à l'étape suivante qui sera un calcul de maximum. Une deuxième coupure α se produit après l'exporation du premier fils de la branche de droite : le minimum sera inférieur ou égal à 1 donc ne sera pas pris en compte pour le calcul du maximum de l'étape suivante :



Coupure beta :
La situation est bien évidemment symétrique dans le cas d'une étape de calcul de maximum, comme l'illustre l'exemple ci-dessous :



Q6. Reprendre l'exemple de la question 5 en indiquant les positions des coupures α et β (on suppose les noeuds explorés de la gauche vers la droite).

Mise en oeuvre :
La mise en oeuvre de cette amélioration consiste à ajouter deux arguments (traditionnellement alpha et beta) qui désignent respectivement une borne inférieure et une borne supérieure de la valeur du noeud.

```
1  def minimax(alpha, beta, p, n):
2          if n == 0 or successeurs(p) == []:
3                  return h(p)
4          mini = np.inf
5          for pk in successeurs(p):
6                  s = maximin(alpha, beta, pk, n-1)
7                  if s < mini:
8                          mini = s
9                  if mini <= alpha:
10                         return mini
11                 beta = min(beta, mini)
12         return mini
```

Les lignes 9-10 correspondent à une coupure α, la ligne 11 à une mise à jour du majorant de la valeur du noeud.

```
1  def maximin(alpha, beta, p, n):
2          if n == 0 or successeurs(p) == []:
3                  return h(p)
4          maxi = -np.inf
5          for pk in successeurs(p):
6                  s = minimax(alpha, beta, pk, n-1)
7                  if s > maxi:
8                          maxi = s
9                  if maxi >= beta:
10                         return maxi
11                 alpha = max(alpha, maxi)
12         return maxi
```

Bibliographie

[1] Cours informatiques - Jean-Pierre Becirspahic