EXPLORATION ET HEURISTIQUES

For progress there is no cure.

John Von Neumann [**von_neumann_can_1955**]

À la fin de ce chapitre, je sais :

- ☞ expliquer la notion d'heuristique
- ☞ appliquer les algorithmes A* et minimax à des cas simples

A Au-delà des jeux d'accessibilité, les heuristiques

De nombreux autres jeux existent qui ne sont pas des jeux d'accessibilité. Par exemple, les échecs ou le go. Pour ces jeux, les joueurs n'ont pas les mêmes options : le joueur qui a les blancs ne peut pas manipuler les noirs de la même manière. Donc la théorie de Sprague-Grundy ne peut pas s'appliquer.

Par ailleurs, pour ces jeux, il est impensable de construire l'arbre de jeu, la combinatoire nous indique que le nombre de parties possibles est bien trop grand. Le nombre de Shannon est une tentative pour estimer le nombre de parties différentes[1] possibles. Il vaut 10^{123} pour les échecs ce qui est plus grand que le nombre d'atomes dans l'univers observable…

S'il nous faut donc renoncer à explorer ces arbres de jeux d'une manière exhaustive, rien ne nous empêche de les explorer localement. Si on utilise un arbre de jeu exhaustif, la partie se finit lorsque la position est une feuille à laquelle est associé un gain ou un score. Lorsqu'on explore localement un arbre de jeu, les feuilles sont parfois absentes voire encore très loin de la position. C'est pourquoi le développement de ces algorithmes s'appuie sur des **heuristiques** capables d'estimer le gain d'une position sans disposer de l'intégralité de l'arbre de jeu.

■ **Définition 1 — Heuristique.** Une heuristique[a] est une approche de résolution de problème à partir de connaissances incomplètes. Une heuristique doit permettre d'aboutir en un temps limité à des solutions néanmoins acceptables même si elles ne sont pas nécessairement optimales.

a. J'ai trouvé en grec ancien.

1. qui ont un sens

B Algorithme Minimax

> Tu sais que ce que tu peux espérer de mieux est d'éviter le pire.
>
> Italo Calvino, 1979

On se place dans le cadre d'un jeu :

— séquentiel à somme nulle,
— à deux joueurs,
— déterministe (sans hasard),
— à information parfaite et complète,
— pour lequel on dispose d'une fonction permettant d'évaluer une position.

R Parmi ces jeux, on trouve par exemple les jeux impartiaux, le morpion, les échecs, le go ou le jeu de hex.

Dans ce cadre, l'algorithme Minimax associé à une heuristique permet de développer des stratégies sans avoir à explorer tout l'arbre de jeu. Il découle naturellement du théorème du même nom démontré par Von Neumann en 1926.

L'algorithme 1 détaille la procédure Minimax. Le principe est le suivant : on construit un arbre de jeu incomplet comme celui de la figure 1. La hauteur de l'arbre est un entier fixé qui limite la profondeur de l'exploration de l'algorithme.

Chaque niveau de l'arbre est contrôlé par un seul joueur. Comme, pour les jeux d'accessibilité considérés, les gains de l'un sont les pertes de l'autre, on choisit de nommer les joueurs J_{max} en rouge et J_{min} en cyan. Le but du jeu est de maximiser le gain pour J_{max}, son score est donc positif, et, symétriquement, minimiser le gain pour J_{min} dont le score est donc négatif.

■ **Définition 2 — Définition du score des joueurs pour une position donnée.** Le jeu associe à chaque feuille de l'arbre minimax un gain qui devient le score du joueur qui atteint cette feuille. Si la feuille est en position p alors on choisit de noter ce score $s(p) \in \mathbb{R}$. À partir du score associé aux feuilles, on peut définir récursivement le score maximal ou minimal associé à un nœud interne p de l'arbre minimax comme suit :

$$\mathcal{S}(p) = \begin{cases} s(p) & \text{si } p \text{ est une feuille} \\ \max\{\mathcal{S}(f), f \text{ fils de } p\} & \text{si } p \text{ est contrôlé par } J_{max} \\ \min\{\mathcal{S}(f), f \text{ fils de } p\} & \text{si } p \text{ est contrôlé par } J_{min} \end{cases} \quad (1)$$

Ainsi, à la fin de la partie, si J_{max} en position p effectue les choix décrits ci-dessus, son score final sera au moins $\mathcal{S}(p)$. De même, le score de J_{min} en position p sera au plus $\mathcal{S}(p)$. Ceci peut se démontrer par récurrence.

Le score d'un joueur ne peut se calculer tel que décrit ci-dessus puisqu'on ne connaît pas l'intégralité de l'arbre de jeu. L'algorithme 1 suppose donc qu'on connaît une **heuristique** $\mathcal{H}$

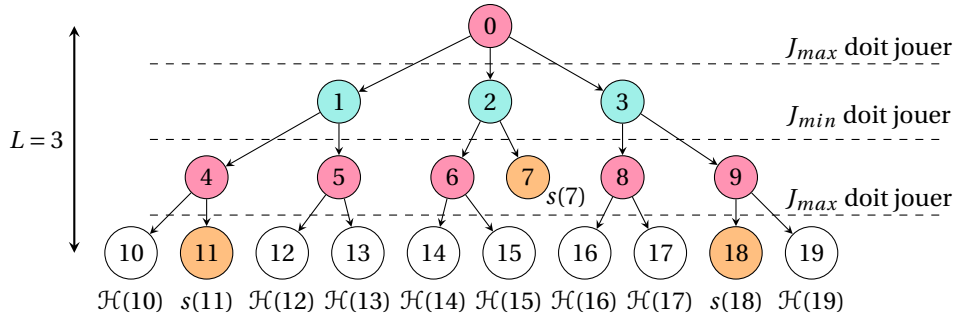

FIGURE 1 – Exemples d'arbre minimax. Chaque niveau est contrôlé par un seul joueur, J_{max} en rouge et J_{min} en cyan. La hauteur de l'arbre L est telle que seule une partie de l'arbre de jeu est accessible. Certaines feuilles sont visibles (en orange, c'est l'automne). L'arbre minimax s'achève donc parfois sur des nœuds internes pour lesquels on donne une estimation du gain (non coloré).

pour estimer le score d'une position d'un nœud intermédiaire. Cette heuristique est une fonction de la position p dans l'arbre et sa valeur est un nombre réel $\mathcal{H}(p)$.

■ **Exemple 1 — Calcul des scores sur un arbre Minimax.** La figure 2 superpose les scores calculés par l'algorithme Minimax sur chaque nœud. Le joueur J_{max} peut donc espérer au plus un score de 6 s'il se trouve en position 0.

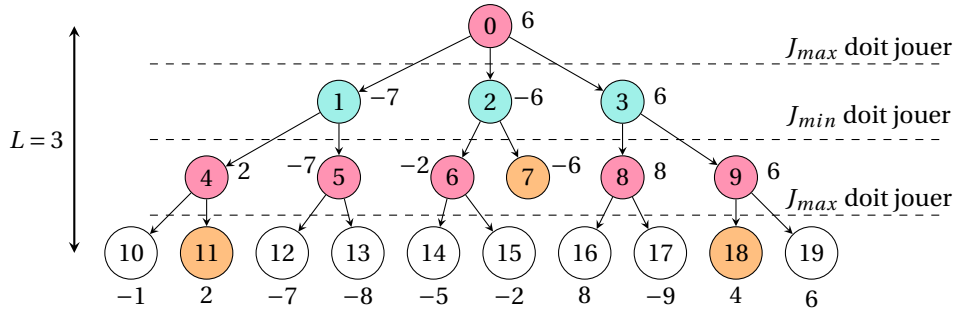

FIGURE 2 – Exemples d'arbre minimax complété avec les scores. Chaque niveau est contrôlé par un seul joueur, J_{max} en rouge et J_{min} en cyan.

R Minimax peut fonctionner sans heuristique particulière si l'arbre à explorer est de taille raisonnable. Mais, dès lors que l'arbre à explorer est grand, il est nécessaire de ne plus explorer toutes les branches et d'évaluer la position courante : c'est le rôle de l'heuristique.

R Une bonne heuristique pour l'algorithme minimax calcule un score : positif pour un joueur et négatif pour l'autre. Ce score doit :

- permettre une bonne **différenciation** des positions : il doit être possible de distinguer les positions gagnantes, celles qui sont équilibrées et les perdantes,
- donner une bonne **approximation** du score réel de la position : plus cette valeur est proche de la valeur exacte obtenue par exploration complète de l'arbre, plus les décisions prises seront pertinentes,
- être **rapide à calculer** pour ne pas ralentir l'exploration de l'arbre.

C Élagage $\alpha\beta$ sur un arbre Minimax --→ HORS PROGRAMME

Selon les situations considérées, limiter la profondeur d'exploration de l'arbre de jeux peut s'avérer être insuffisant pour réduire la complexité du problème. L'élagage $\alpha\beta$ est une technique pour ne pas explorer certaines branches de l'arbre qui n'ont pas besoin de l'être. Le principe est détaillé sur la figure 3.

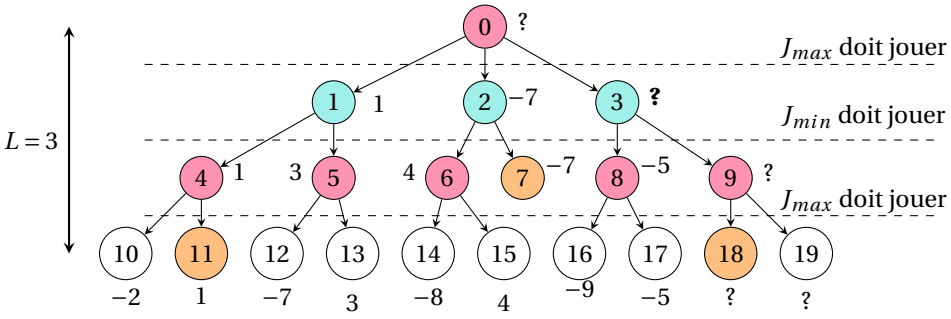

FIGURE 3 – L'élagage $\alpha\beta$ permet sur cet exemple d'éviter l'exploration complète du sous-arbre du nœud numéro 3. En effet, comme c'est un nœud de J_{min}, que le premier nœud du niveau a un score de 1 et que le score calculé du nœud 8 vaut -5, alors on comprend que J_{min} remontera au moins -5 et que J_{max} ne choisira pas ce nœud numéro 3 car ce n'est pas le maximum du niveau.

On distingue deux types d'élagage possible, les types α et β comme le montrent les figures 4 et 5. Pour améliorer la complexité de l'algorithme Minimax, on peut choisir d'élaguer comme le montre l'algorithme 2.

Algorithme 1 Minimax

Entrée : p une position dans l'arbre de jeu (un nœud de l'arbre)
Entrée : s la fonction de score sur les feuilles
Entrée : $\mathcal{H}$ l'heuristique de calcul du score pour un nœud interne
Entrée : L la profondeur maximale de l'arbre Minimax

```
1 : Fonction MINIMAX(p, s, H , L)
2 :     si p est une feuille alors
3 :         renvoyer s(p)
4 :     si L = 0 alors
5 :         renvoyer H(p)                          ▷ On arrête d'explorer, on estime
6 :     si p est contrôlé par J_max alors
7 :         M ← −∞
8 :         p_M un nœud vide
9 :         pour chaque fils f de p répéter
10 :            v, _ ←MINIMAX(f,s,H, L − 1)          ▷ v est un score de J_min
11 :            si v > M alors
12 :                M ← v
13 :                p_M = f
14 :        renvoyer M, p_M          ▷ Valeur maximale trouvée et la racine de cette solution
15 :    sinon
16 :        m ← +∞
17 :        p_m un nœud vide
18 :        pour chaque fils f de p répéter
19 :            v, _ ←MINIMAX(f,s,H, L − 1)          ▷ v est un score de J_max
20 :            si v < m alors
21 :                m ← v
22 :                p_m = f
23 :        renvoyer m, p_m          ▷ Valeur minimale trouvée et la racine de cette solution
```

Algorithme 2 Minimax avec élagage $\alpha\beta$

Entrée : p une position dans l'arbre de jeu (un nœud de l'arbre)
Entrée : s la fonction de score sur les feuilles
Entrée : $\mathcal{H}$ l'heuristique de calcul du score pour un nœud interne
Entrée : L la profondeur maximale de l'arbre Minimax
Entrée : α le niveau de coupure α ▷ $-\infty$ au démarrage
Entrée : β le niveau de coupure β ▷ $+\infty$ au démarrage

```
1 : Fonction MINIMAX_αβ(p, s, H , L, α, β)
2 :     si p est une feuille alors
3 :         renvoyer s(p)
4 :     si L = 0 alors
5 :         renvoyer H(p)                          ▷ On arrête d'explorer, on estime
6 :     si p est contrôlé par J_max alors
7 :         M ← −∞
8 :         p_M un nœud vide
9 :         pour chaque fils f de p répéter
10 :            v, _ ←MINIMAX_αβ(f,s,H, L − 1, α, β)        ▷ v est un score de J_min
11 :            si v > M alors
12 :                M ← v
13 :                p_M = f
14 :            si v ⩾ β alors
15 :                renvoyer M                      ▷ Élagage de type β
16 :            α ← max(α, M)                       ▷ Mise à jour du niveau de l'élagage
17 :        renvoyer M, p_M          ▷ Valeur maximale trouvée et la racine de cette solution
18 :    sinon
19 :        m ← +∞
20 :        p_m un nœud vide
21 :        pour chaque fils f de p répéter
22 :            v, _ ←MINIMAX_αβ(f,s,H, L − 1, α, β)        ▷ v est un score de J_max
23 :            si v < m alors
24 :                m ← v
25 :                p_m = f
26 :            si v ⩽ α alors
27 :                renvoyer m                      ▷ Élagage de type α
28 :            β ← min(β, m)                       ▷ Mise à jour du niveau de l'élagage
29 :        renvoyer m, p_m          ▷ Valeur minimale trouvée et la racine de cette solution
```

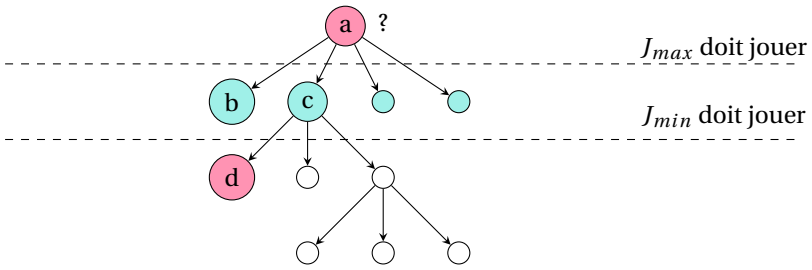

FIGURE 4 – L'élagage α : supposons que $\mathcal{S}(b)$ ait déjà été calculé par l'algorithme Minimax. Si $\mathcal{S}(b) \geqslant \mathcal{S}(d)$, alors il est inutile d'explorer le sous-arbre c : J_{max} ne choisira pas le nœud c, il lui préfèrera b.

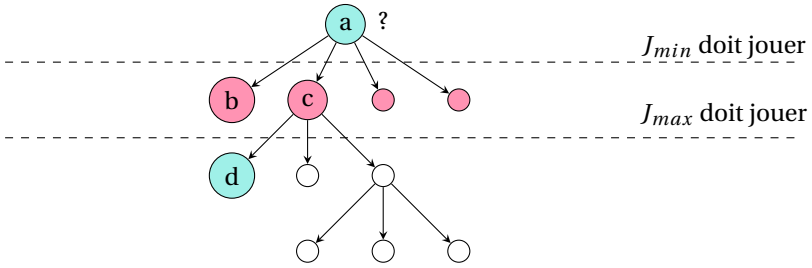

FIGURE 5 – L'élagage β : supposons que $\mathcal{S}(b)$ ait déjà été calculé par l'algorithme Minimax. Si $\mathcal{S}(b) \leqslant \mathcal{S}(d)$, alors il est inutile d'explorer le sous-arbre c : J_{min} ne choisira pas le nœud c, il lui préfèrera b.

D A* pour trouver un chemin

Si le jeu que l'on considère ne peut pas être modélisé selon un arbre Minimax, alors la situation n'est pas désespérée car il nous reste encore les graphes. Il est toujours possible de modéliser l'évolution d'un jeu comme une succession d'état, chaque coup joué permettant de passer d'un état à un autre. Développer une stratégie dans un jeu modélisé par un graphe d'états revient donc à trouver un chemin d'un sommet à un autre dans un graphe. Le premier sommet correspond aux conditions initiales du jeu, le dernier à une condition de gain.

L'algorithme A^* [2] évalue chaque sommet du graphe à l'aide de la distance au sommet de départ et d'une heuristique qui l'informe sur le reste du chemin à parcourir jusqu'au sommet de destination. Le chemin qu'il reste à parcourir est rarement déjà exploré : la plupart du temps, le graphe est trop grand pour être exploré dans son entièreté. C'est pourquoi, l'heuristique évalue le reste du chemin à parcourir sans le parcourir : c'est une **estimation**. Si h est bien choisie, alors A^* converge aussi vite voire plus vite que Dijkstra[**unswmechatronics_dijkstras_2013**].

Soit $G = (S, A, w)$ un **graphe pondéré positivement**. Soit d la fonction de distance utilisée par l'algorithme de Dijkstra (cf. algorithme **??**). Soit h une fonction permettant d'évaluer la distance au point d'arrivée. **L'évaluation de la priorité p d'un sommet s par A^* se fait comme suit :**

$$p(s) = d(s) + h(s) \tag{2}$$

p est la somme de la distance réelle au sommet de départ et de l'estimation de la distance au point d'arrivée procurée par l'heuristique.

L'algorithme A* fonctionne comme un algorithme de Dijkstra pour lequel la priorité dans la file est p. Inversement, l'algorithme de Dijkstra peut être vu comme un cas particulier de A^* avec une heuristique nulle. Le choix d'une heuristique est particulièrement important.

■ **Définition 3 — Heuristique admissible.** Une heuristique $\mathcal{H}$ est admissible si pour tout sommet du graphe d'état, $\mathcal{H}(s)$ est une borne inférieure de la plus courte distance séparant le sommet de départ du sommet d'arrivée.

R Une heuristique admissible ne surestime jamais le coût.

R Une heuristique admissible est toujours nulle sur le sommet d'arrivée.

■ **Définition 4 — Heuristique monotone.** Une heuristique $\mathcal{H}$ est monotone si pour tout arête (a, b) du graphe d'état $G = (S, A, w)$, $\mathcal{H}(a) \leqslant w(a, b) + \mathcal{H}(b)$.

R Une heuristique monotone garantit qu'A^* ne revisite jamais de sommets avec un coût plus bas, ce qui améliore son efficacité : le coût estimé est croissant le long de tout chemin. C'est pourquoi, dans ce cas, A^* ne révisite pas les états redécouverts et sera optimal.

2. prononcer *A étoile* ou *A star*

R Une heuristique monotone est admissible, mais l'inverse n'est pas vrai.

Comment choisir une heuristique pour A^* ?

— Si l'heuristique est monotone, A^* fonctionne de manière optimale, les sommets n'ont pas besoin d'être redécouverts.

— Si l'heuristique est seulement admissible, mais pas monotone, A^* peut encore trouver un chemin optimal, mais il peut être plus lent car certains nœuds doivent être réévalués.

— Si l'heuristique n'est ni admissible ni monotone, A^* ne garantit plus d'obtenir un chemin optimal et peut même ne pas terminer dans certains cas.

L'algorithme 3 donne le détail de la procédure à suivre.

Algorithme 3 Algorithme A*

```
1 : Fonction ASTAR(G = (S, A, w), depart, but, h)        ▷ h est l'heuristique
2 :                                                      ▷ G est un graphe pondéré positivement
3 :     d ← tableau associatif initialisé à +∞           ▷ Distances depuis départ
4 :     Π ← tableau associatif vide                      ▷ Parents pour reconstruire le chemin
5 :     F ← file de priorité vide                        ▷ Contient les sommets à explorer
6 :
7 :     d[départ] ← 0
8 :     ENFILER(F, départ, h(départ))
9 :
10 :    tant que F n'est pas vide répéter
11 :        u ← DEFILER(F)                               ▷ u est le sommet de F minimisant d[u] + h(u)
12 :
13 :        si u = but alors                             ▷ On a trouvé l'objectif!
14 :            renvoyer RECONSTRUIRECHEMIN(Π, s_but)
15 :
16 :        pour chaque voisin v de u répéter
17 :            distance ← d[u] + w(u, v)
18 :            si distance < d[v] alors                 ▷ On a trouvé un meilleur chemin vers v
19 :                d[v] ← distance
20 :                Π[v] ← u                             ▷ Mettre à jour le parent sur le chemin
21 :                priorité ← d[v] + h(v)               ▷ Estimation du coût total
22 :                ENFILER(F, v, proirité)
23 :    renvoyer ∅                                       ▷ Échec : pas de chemin
```