V Algorithme Min-Max

25 janvier 2026

1 Heuristique et principe

Dans le chapitre précédent, nous avons vu comment faire jouer de manière parfaite une IA dans un jeu à deux joueurs, dans le cadre des stratégies sans mémoire.

Cette approche fonctionne sur de « petits » jeux, comme le jeu de Nim ou le morpion 3×3, mais n'est pas praticable pour des jeux plus développés, ne serait-ce que ceux comme le puissance 4. Pour beaucoup de jeux, le graphe du jeu est bien trop gros (le jeu d'échecs comporterait plus de 10^{46} positions possibles) pour que l'on puisse l'explorer de manière exhaustive.

Nous allons donc étudier une méthode permettant à une IA de jouer à un jeu, sans exploration complète du graphe, et donc de manière non parfaite.

Dans ce chapitre, le joueur J_0 portera le nom de « Maximilien » (que l'on abrégera en « Max »), et le joueur J_1 portera le nom de « Minerva » (que l'on abrégera en « Min »).

On commence par supposer qu'il existe une fonction qui permet d'évaluer la qualité d'une position, et qui renvoie un nombre (éventuellement infini). Plus cette position est favorable à Maximilien, plus le nombre renvoyé est censé être élevé, et plus cette position est favorable à Minerva, plus le nombre renvoyé est censé être bas : une telle fonction est appelée *heuristique*.

Remarque 1.0.1.
Le mot heuristique vient du grec verbe grec εὑρίσκειν : rencontrer, trouver par hasard. Le mot εὕρηκα (eurêka) en est dérivé.

Littré donne pour heuristique : « L'art d'inventer, de faire des découvertes. ».

Le CNRTL donne une définition plus appropriée à notre cadre « Qui sert à la découverte ; qui est propre à guider une recherche ou à vérifier une hypothèse. ».

Nous pouvons en donner une définition plus formelle.

Définition 1.0.2 (heuristique)**.**
Soit (S, A) le graphe orienté modélisant le jeu. Une *heuristique* sur le jeu est une fonction

$$h : S \mapsto \mathbb{R} \cup \{-\infty, +\infty\}.$$

Remarque 1.0.3.
Une heuristique est souvent construite de manière expérimentale. Plus l'heuristique utilisée sera pertinente, meilleure sera la qualité de l'IA construite dessus.

Remarque 1.0.4.
On associe souvent la valeur $+\infty$ aux positions de victoire pour Maximilien, et $-\infty$ à celles de victoire pour Minerva.

Exemple 1.0.5.
Dans le jeu de morpion 3×3, étant donné une grille, on pourrait procéder ainsi (voir figure 1 pour un exemple où l'on considère la case du milieu d'un alignement).

- ▷ On considère les 8 alignements de 3 cases, et parmi ceux-ci ceux où Minerva n'a pas joué :
  - on attribue 100 points si Maximilien a joué dans les trois cases ;
  - on attribue 10 points à ceux qui comptent deux cases jouées par Maximilien ;
  - on attribue 1 points à ceux qui comptent une cases jouées par Maximilien.
- ▷ On procède de même parmi les alignements de 3 cases où Maximilien n'a pas joué :
  - on attribue -100 points si Minerva a joué dans les trois cases ;
  - on attribue -10 points à ceux qui comptent deux cases jouées par Minerva ;
  - on attribue -1 points à ceux qui comptent une cases jouées par Minerva.

On somme alors les points obtenus.

Figure 1 – Exemple d'heuristique sur un alignement, pour la case du milieu

Exemple 1.0.6.
Pour (presque) n'importe quel jeu, à partir de toute position, on peut jouer des parties « au hasard », et prendre pour heuristique un score construit sur ces résultats (par exemple, le pourcentage de victoires de Maximilien parmi les parties non nulles, ou $0,5$ si toutes les parties finissent en des matchs nuls).

L'idée générale de l'algorithme Min-Max est le suivant : pour jouer à partir d'une position, les joueurs s'autorisent à explorer « n coups en avant », ce paramètre n étant fixé. Les deux joueurs partent de l'heuristique, comme fonction d'évaluation des positions, et pour chaque coup :

- ▷ Maximilien cherche le coup maximisant le score défini par l'heuristique ;
- ▷ Minerva cherche le coup minimisant le score défini par l'heuristique.

Cela s'organise donc ainsi (on donnera des traductions plus concrètes dans la partie suivante).

- ▷ On commence par fixer une profondeur d'exploration n.
- ▷ On fixe aussi une heuristique h.
- ▷ Si $n = 0$, ou si la position p que l'on étudie n'a aucun successeur possible, alors on l'évalue à l'aide de l'heuristique. On lui attribue donc un score $h(p)$.
- ▷ Sinon, on construit à partir de la position p étudiée la liste des successeurs $\{p_1, \dots, p_k\}$ de p, obtenus en jouant un coup, ainsi que leurs scores $\{s_1, \dots, s_k\}$ obtenus en explorant subséquemment le graphe à une profondeur $n-1$. À partir de ces scores :
  - si Maximilien joue, alors le score de la position est
    $$s = \max\{s_1, \dots, s_k\},$$
    et Maximilien joue sur le (ou un) coup atteignant ce score ;
  - si Minerva joue, alors le score de la position est
    $$s = \min\{s_1, \dots, s_k\},$$
    et Minerva joue sur le (ou un) coup atteignant ce score.

Exercice 1.0.7.
Considérons une profondeur d'exploration de 4, et un jeu décrit par le graphe suivant (voir figure 2, les nœuds où l'on utilise l'heuristique sont représentés avec les valeurs de l'heuristique). Remplir le graphe avec les valeurs correspondantes, et déterminer

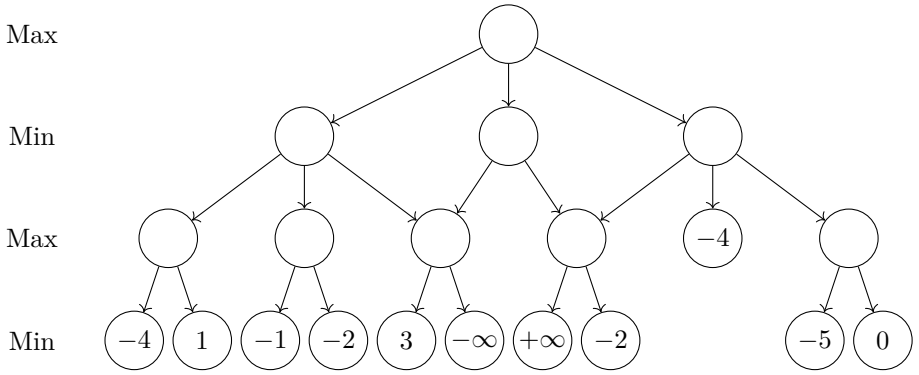

Figure 2 – Un exemple de Min-Max

le coup joué par Maximilien.

Peut-on prédire le coup joué par Minerva ?

2 Code naïf

Dans cette partie, nous supposons que l'on dispose de trois fonctions.

- `coups_possibles` : elle prend en argument une position du jeu, et renvoie la liste des positions atteignables (donc des coups que l'on peut jouer).
- `joueur_courant` : elle prend un argument une position du jeu, et renvoie le joueur qui joue sur cette position (0 pour Maximilien, et 1 pour Minerva).
- `heuristique` : elle prend en argument une position du jeu, et renvoie le score donné par l'heuristique.

Nous allons travailler avec des listes de couples (position,score).

Commençons par écrire une fonction prenant en argument une liste de couples, contenant les positions et leurs scores, et renvoyant le couple de score maximum.

```
Fonction position_score_max(listePositionsScores) :
    positionMax,scoreMax ← listePositionsScores[0]
    Pour chaque position,score De listePositionsScores[1 :],
Faire
        Si score>scoreMax Alors
            positionMax,scoreMax ← position,score
    Renvoyer positionMax,scoreMax
Fin Fonction
```

On peut alors effectuer la même chose pour renvoyer le couple de score minimum.

```
Fonction position_score_min(listePositionsScores) :
    positionMin,scoreMin ← listePositionsScores[0]
    Pour chaque position,score De listePositionsScores[1 :],
Faire
        Si score<scoreMin Alors
            positionMin,scoreMin ← position,score
    Renvoyer positionMin,scoreMin
Fin Fonction
```

On peut maintenant écrire une fonction prenant en argument la position du jeu, la profondeur d'exploration, et renvoyant le couple contenant le score attribué à la position, et le coup à jouer.

```
Fonction minimax(position,profondeur) :
    listeCoups ← coups_possibles(position)
    Si profondeur = 0 ou listeCoups est vide Alors
        Renvoyer -1,heuristique(position)
    Sinon
        listePositionsScores ← liste vide
        Pour chaque positionSuivante De listeCoups, Faire
            _,score ← minimax(positionSuivante,n-1)
            ajouter (positionSuivante,score) à
listePositionsScores
        joueur ← joueur_courant(position)
        Si joueur = 0 Alors
            Renvoyer position_score_max(listePositionsScores)
        Sinon
            Renvoyer position_score_min(listePositionsScores)
Fin Fonction
```

3 Code mémoïsé

On peut remarquer que le code précédent présente un défaut lorsque le graphe du jeu n'est pas un arbre (comme dans l'exercice 1.0.7) : on peut être ammené à calculer plusieurs fois le score pour un même nœud. Vous savez toutefois proposer une solution à ce défaut : il suffit de mémoïser les scores sur chaque nœud. Le défaut d'une telle stratégie est alors la construction d'une structure de données (qui consomme donc de l'espace mémoire), mais qui permet potentiellement de gagner en complexité temporelle.

Nous choisirons de mémoïser pour une position trois informations, à l'aide de dictionnaire :

- ▷ le coup optimal à partir de la position ;
- ▷ le score de la position ;
- ▷ la profondeur d'exploration.

En effet, si l'on rencontre un nœud déjà évalué, il convient de le réexplorer si la profondeur demandée est supérieure (strictement) à celle à laquelle le nœud a été exploré précédemment.

```
Fonction minimax_memoisé(position,profondeur) :
    memo ← dictionnaire vide
    Fonction aux(position,profondeur) :
        Si position ∈ memo Alors
            coupMemo,scoreMemo,profondeurMemo ← memo[position]
            Si profondeurMemo ⩾ profondeur Alors
                Renvoyer (positionMemo,scoreMemo)
        listeCoups ← coups_possibles(position)
        Si profondeur = 0 ou listeCoups est vide Alors
            score ← heuristique(position)
            memo[position] ← -1,score,profondeur
            Renvoyer -1,score
        listePositionsScores ← liste vide
        Pour chaque positionSuivante De listeCoups, Faire
            _,score ← aux(positionSuivante,n-1)
            ajouter (positionSuivante,score) à
listePositionsScores
        joueur ← joueur_courant(position)
        Si joueur = 0 Alors
            coup,score ← position_score_max(listePositionsScores)
        Sinon
            coup,score ← position_score_min(listePositionsScores)
        memo[position] ← coup,score,profondeur
        Renvoyer coup,score
    Fin Fonction
    Renvoyer aux(position,profondeur)
Fin Fonction
```