# MinMax

Nous allons maintenant implémenter un algorithme **MinMax**, permettant de prédire des futures positions (à un degré donné) pour évaluer la position actuelle, et donner le prochain meilleur coup, de façon un peu bourrine (mais on l’améliorera avec un élagage alpha-beta).

Brièvement, **MinMax** retourne le prochain coup de façon à minimiser la meilleure position de l’adversaire du joueur courant.

On crée donc deux fonctions :

* **minimax**, qui va s’appeler récursivement en diminuant son degré (sa profondeur) actuel jusqu’à atteindre le degré voulu, où il va utiliser **EvalPosition**. Cette fonction prend en paramètres :
  + La dimension de notre gomoku : *N*
  + La position actuelle dont on veut évaluer la position : *P*
  + Le nombre de pions à aligner : *A*
  + Le degré (la profondeur) actuel : on a déjà prédit combien de coups par rapport à la position à évaluer ? : *degre\_actuel*
  + Le degré voulu : combien de coups on veut prédire à l’avance : *degre\_voulu*
  + Quel joueur joue à ce tour ? : *joueur*

Cette fonction va retourner le score de la position qu’on lui a donné, ainsi que les coordonnées du coup optimal

* **MinMax**, qui va effectuer l’appel principal. Cette fonction a les mêmes paramètres que minimax à l’exception de *degre\_actuel*.

Remarque : Utiliser **MinMax** à un degré 0 revient à faire simplement appel à **EvalPosition** sur la position courante.

Si le degré voulu est strictement positif, **MinMax** vérifie si le degré voulu est supérieur au nombre de cases vides, puis appelle minimax (avec un *degré\_actuel* à 0), qui procède comme ci-décrit.

D’abord, on détermine toutes les prochaines positions possibles :

* Chaque case vide est un potentiel prochain coup à jouer

🡺 On récupère toutes les cases vides

* Pour chacune de ces cases vides
  + On crée une nouvelle position où le joueur courant (humain ou ordinateur) a joué dans cette case vide
  + On sauvegarde cette nouvelle position

De cette manière, on récupère toutes les prochaines positions.

Il nous faut maintenant évaluer chacune de ces positions. C’est ici qu’on fait apparaitre l’aspect récursif de **minimax** :

* On initialise le *score optimal* à –inf. si c’est le joueur qui joue, +inf. si c’est l’ordinateur, et le *coup optimal* à Null
* On diminue le degré actuel de 1
* Si le degré devient égal au degré voulu, pour chaque prochaine position *Next\_P*
  + On fait simplement **EvalPosition** sur cette position *Next\_P* pour récupérer son score
  + Sinon on fait **minimax** sur cette position pour récupérer le score de *Next\_P*, avec les mêmes paramètres (degré a déjà été diminué de 1), sauf *joueur* où on spécifie cette fois l’autre joueur, et *P* qui devient la prochaine position *Next\_P*
  + Pour chaque prochaine position, on vérifie si son score est meilleur (= inférieur si c’est l’ordi qui va jouer, =supérieur si c’est l’humain) que le *score optimal* (le meilleur score parmi les scores des positions qu’on a déjà parcouru)
    - Si c’est le cas, ce nouveau score devient le *score optimal*, et le *coup optimal* devient le coup joué pour arriver à la prochaine position
* Enfin, on retourne le *score optimal* et le *coup optimal* précédemment calculé

Et le tour est joué ! Faisons maintenant un exemple de déroulement sur une partie simple : avec un degré voulu de 2

|  |  |  |
| --- | --- | --- |
| X | O |  |
| O |  |  |
| X |  |  |

Ici, on admet que c’est à l’ordinateur (X) de jouer.

Les 5 coups possibles suivants sont les suivants :

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| X | O | X |  | X | O |  |  | X | O |  |  | X | O |  |  | X | O |  |
| O |  |  |  | O | X |  |  | O |  | X |  | O |  |  |  | O |  |  |
| X |  |  |  | X |  |  |  | X |  |  |  | X | X |  |  | X |  | X |

Ici, minimax va calculer le score de chacune des prochaines positions (les *sous-positions* où l’humain a joué un nouveau coup) de chacune de ces positions avec **EvalPosition**, et donc évaluer chacune de ces positions ci-dessus en gardant le score où l’humain est le plus avantagé (🡺 le score maximum) parmi les scores des *sous-positions*.

Enfin, on refait la même chose pour évaluer la position de base, sauf que cette fois on garde le score où l’ordinateur est le plus avantagé (🡺 le score minimum) parmi les scores des positions calculés. Au final, dans cet exemple, le 2ème cas est le meilleur car il aura deux lignes potentielles à compléter, et aura donc un score minimum (optimal pour l’ordinateur).

# Elagage alpha-beta :

MinMax permet d’anticiper tous les prochains coups, ce qui peut très vite devenir long. L’idée de cette implémentation est de retirer les parcours inutiles, d’éliminer les coups qu’on sait pas optimaux, de façon à accélérer l’algorithme. Il exploite le fait qu’un coup est forcément favorable (ou nul) au joueur qui l’a joué.

Pour cela, on ajoute deux paramètres à **minimax** : *alpha* et *beta*, qui représentent respectivement le *score* optimal du degré précédent (minimum pour *alpha*, maximum pour *beta*). On n’utilise qu’un des deux à la fois, dépendant de si le nœud courant est un nœud max (tour de l’humain : dans ce cas c’est *beta* qui est utilisé) ou si c’est un nœud min (tour de l’ordinateur : dans ce cas on utilise alpha).

L’algorithme est le suivant :

* Si on est au tour du joueur,
  + On fait comme ce que faisait minimax
  + Durant le parcours des sous-positions, à la fin, on vérifie si notre *score optimal* (maximum) est supérieur à *beta*, le minimum de la position mère.
    - Si c’est le cas, ça veut dire que le score de la position courante (sous position de la position mère) sera forcément supérieur (car on cherche le max 🡺 il ne peut qu’augmenter) au *score optimal* de la position mère, qui cherche à être minimum. On arrête donc le parcours des sous-positions en retournant le *score optimal* actuel ainsi que le coup à jouer pour arriver dans cette position.

Prenons l’exemple