L'Algorithme MINIMAX

Notes de cours 3

Nadjib Lazaar
Université Paris-Saclay
lazaar@lisn.fr

L'algorithme MINIMAX est une méthode de décision utilisée dans les jeux à deux joueurs à somme nulle, où chaque joueur cherche à maximiser ses gains tout en minimisant ceux de l'adversaire. Il s'appuie sur une exploration récursive de l'arbre de jeu pour déterminer la meilleure action à chaque étape, en supposant que les deux joueurs jouent de manière optimale.

1 Histoire et Historique de l'Algorithme Minimax

L'algorithme Minimax trouve ses origines dans la théorie des jeux, un domaine des mathématiques appliquées. Il a été formalisé dans les années 1920 et 1930 par le mathématicien **John von Neumann**.

1.1 Origines et Premiers Travaux

L'algorithme Minimax a été introduit par **John von Neumann** dans les années 1920. Ce dernier a développé les bases de la théorie des jeux dans son travail pionnier, qui a jeté les fondations de l'algorithme Minimax. En 1928, von Neumann a proposé que les jeux à deux joueurs avec somme nulle (où l'un gagne ce que l'autre perd) pouvaient être modélisés et résolus par des stratégies optimales.

1.2 La Théorie des Jeux

La théorie des jeux a été formellement établie avec la publication du livre *Theory of Games and Economic Behavior* en 1944, coécrit par **John von Neumann** et **Oskar Morgenstern**. Cet ouvrage a défini le cadre de la théorie des jeux comme un domaine mathématique autonome, traitant des décisions stratégiques dans des contextes compétitifs. Von Neumann a démontré que les jeux à somme nulle pouvaient être résolus par une stratégie optimale, qui correspond à l'algorithme Minimax.

1.3 Formalisation du Minimax

Le principe du Minimax repose sur l'idée que chaque joueur choisit une stratégie qui minimise le gain potentiel de l'adversaire tout en maximisant son propre gain. Cette approche de prise de décision a été cruciale pour la résolution des jeux à deux joueurs dans lesquels chaque participant cherche à optimiser ses chances tout en contrôlant les choix de l'autre.

1.4 Premiers Programmes Informatiques

L'algorithme Minimax a été utilisé pour résoudre des jeux à deux joueurs comme les échecs et le jeu de dames. En 1951, le programme informatique *Turochamp*, développé par **Alan Turing** et **D. G. Champernowne**, a été l'un des premiers à appliquer une version rudimentaire de l'algorithme Minimax pour jouer aux échecs. Ce fut un premier pas dans l'usage de l'IA pour simuler des jeux de stratégie complexes.

1.5 Optimisations et Améliorations

Au fil du temps, il est devenu évident que l'explosion combinatoire dans les grands jeux rendait l'algorithme Minimax impraticable sans améliorations. En 1950, **Allen Newell**, **Herbert A. Simon** et **John McCarthy** ont introduit des techniques d'optimisation telles que l'*élagage alpha-bêta* qui réduit le nombre de nœuds explorés dans l'arbre de décision tout en garantissant le même résultat. Cette optimisation rend l'algorithme plus efficace, en permettant de traiter des arbres de décision plus grands.

1.6 Applications Modernes

Aujourd'hui, l'algorithme Minimax reste largement utilisé dans des jeux à somme nulle comme les échecs et le jeu de dames. Cependant, son application a aussi inspiré des approches similaires dans d'autres domaines de l'IA, notamment dans la robotique, la stratégie militaire et la finance. Les techniques dérivées du Minimax continuent d'être enseignées dans les cours d'IA et de théorie des jeux.

1.7 Conclusion

L'algorithme Minimax est l'une des premières applications de la théorie des jeux dans le domaine de l'informatique. Depuis ses premières applications dans les jeux de stratégie, il a évolué pour devenir une pierre angulaire des systèmes de décision autonomes. Son développement et ses améliorations ont permis de réaliser des percées majeures dans l'IA, et son héritage est toujours présent dans les technologies actuelles.

Algorithme 1 : MINIMAX Algorithm

Input : s : current state ; d : max depth ;
$player_1$: Indicates whether it is the maximizing player's turn.
Output : `bestValue` : The utility value for the current state.

```
1  begin
2      if isLeaf(s) ∨ d = 0 then
3          return EVAL(s)
4      if player_1 then
5          bestValue ← −∞
6          foreach s' ∈ child(s) do
7              bestValue ← max(bestValue, MINIMAX(s', d − 1, false))
8          return bestValue
9      else
10         bestValue ← +∞
11         foreach s' ∈ child(s) do
12             bestValue ← min(bestValue, MINIMAX(s', d − 1, true))
13         return bestValue
14 Function EVAL(s) :
15     if isLeaf(s) then return score(s)  // Return the exact score
16     value ← 0
17     foreach f ∈ feature(s) do
18         value ← value + cost(f)
19     return value
```

2 L'Algorithme

L'algorithme alterne entre deux types d'étapes :

— **Étapes MAX** : représentent les décisions du joueur principal (Joueur 1), qui cherche à maximiser son gain.

— **Étapes MIN** : représentent les décisions de l'adversaire (Joueur 2), qui cherche à minimiser le gain du joueur principal.

Chaque nœud de l'arbre de jeu est évalué avec une fonction d'utilité, et les valeurs des nœuds sont propagées vers la racine selon les règles suivantes :

— Un nœud MAX prend la valeur maximale parmi ses enfants.

— Un nœud MIN prend la valeur minimale parmi ses enfants.

L'algorithme MINIMAX fonctionne de manière récursive en explorant les états futurs du jeu. Si l'état courant est une feuille de l'arbre (fin de partie) ou si la profondeur maximale d est atteinte, la fonction d'évaluation EVAL retourne une valeur représentant l'utilité de cet état. Lorsque c'est au tour du joueur maximisant ($player_1 = true$), l'algorithme initialise une variable `bestValue` à $-\infty$, puis évalue chaque coup possible et met à jour cette valeur avec le maximum

des résultats des appels récursifs. Inversement, quand c'est au tour du joueur minimisant ($player_1 = false$), `bestValue` est initialisé à $+\infty$ et mis à jour avec le minimum des résultats des appels récursifs. L'objectif est d'anticiper les coups de l'adversaire pour optimiser les décisions. La fonction d'évaluation EVAL peut soit donner un score exact pour les états terminaux, soit estimer une valeur en se basant sur des caractéristiques spécifiques de l'état :

- **Exactitude pour les feuilles :** Lorsqu'un état s est une feuille, cela signifie que le jeu est terminé à cet état (victoire, défaite, ou match nul). Dans ce cas, la fonction EVAL(s) retourne un score exact basé sur les règles du jeu, car l'issue est définitivement déterminée. Par exemple, dans un jeu comme les échecs, une victoire peut être représentée par un score élevé, une défaite par un score bas, et un match nul par un score neutre. Ces scores reflètent directement la valeur réelle de l'état pour le joueur maximisant.
- **Estimation pour les nœuds non terminaux :** Lorsque s n'est pas une feuille, la partie n'est pas terminée, et l'issue dépend encore des décisions à venir. Dans ce cas, la fonction EVAL(s) doit fournir une estimation de la qualité de cet état, souvent basée sur des caractéristiques spécifiques (heuristiques). Par exemple, dans les échecs, cette estimation peut prendre en compte le nombre et la qualité des pièces, le contrôle du centre, ou la sécurité du roi. Ces heuristiques donnent une approximation raisonnable, mais elles ne garantissent pas une précision parfaite, car elles ne prennent pas en compte toutes les séquences possibles de coups à partir de cet état.

3 Complexité

L'algorithme MINIMAX prend en paramètre une profondeur d, qui limite la recherche à une certaine hauteur dans l'arbre de jeu. Dans le pire des cas, cette profondeur d correspond à la hauteur complète de l'arbre, ce qui signifie que tous les états possibles du jeu sont explorés jusqu'aux feuilles. La complexité temporelle de l'algorithme est alors donnée par $O(n^d)$, où n est le **facteur de branchement**, c'est-à-dire le nombre moyen d'actions possibles à chaque étape du jeu, et d est la **profondeur maximale** explorée. Ainsi, plus n ou d sont grands, plus le nombre total d'états à explorer augmente de façon exponentielle. Cette croissance rapide rend l'algorithme coûteux en temps pour des jeux où le facteur de branchement ou la profondeur sont importants, ce qui peut rendre son utilisation impraticable sans optimisations comme l'élagage alpha-bêta.

4 Limites et Extensions

Bien que l'algorithme MINIMAX garantisse une stratégie optimale lorsque les joueurs jouent parfaitement, il souffre de l'explosion combinatoire due à la

taille de l'arbre de recherche. Pour pallier ces limites et améliorer son efficacité, plusieurs optimisations et extensions peuvent être mises en œuvre :

- **Élagage Alpha-Bêta** : Cette méthode réduit le nombre de nœuds explorés en éliminant les branches qui ne peuvent pas influencer le résultat final. Elle maintient deux valeurs (α pour le meilleur score garanti pour le joueur maximisant, et β pour le joueur minimisant) et interrompt l'exploration des branches inutiles lorsque les limites sont dépassées, tout en conservant une stratégie optimale.
- **Tri des coups** : L'ordre dans lequel les coups sont explorés peut influencer l'efficacité, notamment avec l'élagage alpha-bêta. Trier les coups prometteurs en premier maximise les opportunités d'élagage et réduit le travail nécessaire.
- **Mémorisation (Table de transposition)** : En utilisant une table de transposition pour stocker les résultats des états déjà évalués, on évite de recalculer plusieurs fois les mêmes valeurs dans l'arbre de recherche. Cela est particulièrement utile dans les jeux où les états peuvent se répéter fréquemment, comme aux échecs.
- **Bases de données de fin de partie** : Certaines positions gagnantes ou nulles en fin de partie peuvent être pré-calculées et stockées dans une base de données. Lorsqu'un état correspond à une position connue, l'algorithme peut immédiatement consulter la base pour déterminer la meilleure décision.
- **Séquences de bons coups** : Identifier et mémoriser des motifs ou séquences de coups efficaces pour certaines configurations de jeu permet de guider l'algorithme vers des solutions rapides et précises, limitant ainsi l'exploration exhaustive.
- **Heuristiques** : Lorsque l'arbre de recherche est trop profond, il est courant de limiter la recherche à une profondeur d fixée et d'utiliser une fonction heuristique pour évaluer les nœuds intermédiaires. Cette fonction peut intégrer des caractéristiques spécifiques du jeu, comme la qualité des pièces aux échecs ou le contrôle de zones stratégiques sur un plateau.
- **Élagages basés sur des motifs** : Reconnaître et exploiter des motifs fréquents ou des configurations spécifiques du jeu pour les élaguer ou les évaluer rapidement permet de réduire la complexité sans affecter la précision de la recherche.

Ces optimisations et extensions, utilisées conjointement, permettent de rendre l'algorithme Minimax plus performant en réduisant le nombre de nœuds explorés et en guidant la recherche vers les solutions les plus prometteuses, atténuant ainsi les limites liées à l'explosion combinatoire.

5 Illustration de l'algorithme MINIMAX et de la fonction EVAL : Exemple sur le morpion

Pour illustrer le fonctionnement de l'algorithme MINIMAX, prenons l'exemple du jeu du morpion (*tic-tac-toe*). Nous supposons que deux joueurs, X (Joueur 1) et O (Joueur 2), alternent leurs tours. La partie se termine lorsqu'un joueur gagne ou qu'il n'y a plus de cases disponibles.

État s_i : Considérons le plateau intermédiaire suivant :

X	O	
	X	O

À ce tour, X doit jouer. L'objectif est de maximiser la valeur de l'état du jeu en choisissant un coup optimal. La fonction EVAL est utilisée pour évaluer la qualité de chaque état possible.

Définition de la fonction EVAL

La fonction EVAL repose sur deux éléments :

1. Des *features* extraites du plateau, avec des valeurs attribuées selon leur importance.
2. Une fonction de score appliquée aux feuilles terminales du jeu.

Ces deux éléments sont détaillés dans les tableaux ci-dessous :

Feature	**Coût**
Lignes/colonnes/diagonales gagnables pour X ou O	+1
Lignes/colonnes/diagonales presque gagnantes pour X	+3
Lignes/colonnes/diagonales presque gagnantes pour O	-3
Centre contrôlé par X	+2
Centre contrôlé par O	-2

TABLE 1 – Features utilisées pour l'évaluation d'un état intermédiaire dans le morpion

État terminal (feuille)	**Score**
Victoire immédiate pour X	$+\infty$
Victoire immédiate pour O	$-\infty$
Égalité (plateau complet sans gagnant)	0

TABLE 2 – Scores attribués aux feuilles terminales dans le morpion

Les lignes/colonnes/diagonales sont dites *gagnables* si elles ne contiennent que des X, des O, ou sont vides. Une ligne est *presque gagnante* lorsqu'elle contient exactement deux X ou deux O et aucune pièce adverse.

Analyse de l'état s_i

À partir du plateau donné, nous identifions les *features* présentes :

1. **Lignes gagnables :**
   — Ligne 3 $(\_, \_, \_)$ est gagnable pour X ou O : $+1$.
2. **Colonnes gagnables :**
   — Colonne 1 $(X, \_, \_)$ est gagnable pour X ou O : $+1$.
   — Colonne 3 $(\_, O, \_)$ est gagnable pour X ou O : $+1$.
3. **Diagonales gagnables :**
   — Diagonale 2 $(\_, X, \_)$ est gagnable pour X ou O : $+1$.
4. **Diagonales presque gagnantes pour X :**
   — Diagonale 1 $(X, X, \_)$: $+3$.
5. **Centre contrôlé :**
   — Case centrale contrôlée par X : $+2$.

Score total de EVAL pour l'état s_i :

$$\text{eval}(s) = +1(\text{ligne 3}) + 1(\text{colonne 1}) + 1(\text{colonne 3}) + 1(\text{diagonale 2}) + 3(\text{diagonale 1 presque gagnante})$$
$$+2(\text{centre contrôlé}) = +9$$

Déroulement de MINIMAX

X (Joueur 1, maximisation) examine les options possibles. Supposons que les coups disponibles soient :

— s_1 : X joue en (3,3) :

X	O	
	X	O
		X

$\text{eval}(s_1) = +\infty$ (victoire immédiate pour X).

— s_2 : X joue en (3,2) :

X	O	
	X	O
	X	

$\text{eval}(s_2) = +8$.

— s_3 : X joue en (3,1) :

X	O	
	X	O
X		

$\text{eval}(s_3) = +13$.

Jeu	branchement	Profondeur	Taille estimée
Morpion (Tic-Tac-Toe)	3	9	26 830
Puissance 4	7	42	10^{14}
Dames	8	50	10^{20}
Othello (Reversi)	10	60	10^{58}
Échecs	35	80	10^{120} (nombre de Shannon)
Go	250	300+	10^{170}

TABLE 3 – Exemples de tailles d'arbres de jeu pour différents jeux

Choix optimal pour X : L'algorithme choisit s_1 car il maximise immédiatement le score en garantissant une victoire ($\text{eval}(s_1) = +\infty$).

6 Élagage ALPHA-BÊTA

L'algorithme ALPHA-BÊTA, ou élagage ALPHA-BÊTA, a été conçu pour améliorer l'efficacité de l'algorithme MINIMAX, utilisé pour la prise de décision dans les jeux combinatoires comme les échecs. Il est basé sur des idées introduites pour la première fois dans les années 1940 et 1950.

Les premières mentions formelles de l'élagage ALPHA-BÊTA apparaissent dans les travaux de **John McCarthy** et d'autres chercheurs pionniers de l'IA. Cependant, le véritable développement et la formalisation de l'algorithme sont attribués à deux groupes indépendants : **Allen Newell** et **Herbert Simon**, d'une part, et des chercheurs comme **Arthur Samuel**, d'autre part, au cours des années 1950 et 1960.

L'idée principale est d'élaguer (c'est-à-dire d'éviter d'explorer) des branches de l'arbre de décision qui ne peuvent pas influencer le résultat final. Cette optimisation réduit significativement le nombre de nœuds évalués, permettant une exploration plus profonde avec les mêmes ressources computationnelles.

Au fil des décennies, l'algorithme Alpha-Bêta a joué un rôle central dans le développement des systèmes de jeu d'intelligence artificielle, y compris des programmes célèbres comme Deep Blue d'IBM. Son importance réside dans sa simplicité et son efficacité, qui en font encore un standard pour de nombreux jeux à deux joueurs.

6.1 L'Algorithme ALPHA-BÊTA

L'algorithme ALPHA-BÊTA est une optimisation de l'algorithme MINIMAX qui permet de réduire considérablement le nombre de nœuds à explorer dans l'arbre de jeu. En pratique, il élaguera les branches qui ne peuvent pas affecter la décision finale, ce qui permet d'améliorer l'efficacité de l'algorithme MINIMAX, tout en garantissant les mêmes résultats.

L'algorithme repose sur deux valeurs, α et β, qui servent de bornes pour déterminer si une branche peut être éliminée (pruning).

— α représente la valeur maximale que le joueur maximisant (Joueur 1) est prêt à accepter.
— β représente la valeur minimale que le joueur minimisant (Joueur 2) est prêt à accepter.

Lorsque l'algorithme explore un nœud, il met à jour ces deux bornes, et si la valeur de l'un des nœuds dépasse α ou β, il n'est plus nécessaire d'explorer ses descendants, car ils ne pourront pas influencer la décision finale.

6.2 Principe de l'algorithme Alpha-Bêta

L'algorithme Alpha-Bêta utilise une approche similaire à Minimax, mais avec des bornes dynamiques qui permettent d'élaguer des branches non pertinentes de l'arbre de jeu. Le processus est le suivant :

— Lors de l'exploration d'un nœud MAX, si la valeur d'un enfant est supérieure à β, on peut arrêter l'exploration de ce sous-arbre (pruning), car le joueur MIN (joueur 2) ne permettrait jamais d'atteindre cette valeur.
— Lors de l'exploration d'un nœud MIN, si la valeur d'un enfant est inférieure à α, on peut arrêter l'exploration de ce sous-arbre, car le joueur MAX ne permettrait jamais d'accepter cette valeur.

Explication de l'algorithme Alpha-Bêta

L'algorithme utilise deux bornes, α et β, pour représenter les meilleures valeurs possibles respectivement pour le joueur maximisant ($player_1$) et le joueur minimisant.

— α : La meilleure valeur trouvée jusqu'à présent pour le joueur maximisant.
— β : La meilleure valeur trouvée jusqu'à présent pour le joueur minimisant.

L'algorithme explore récursivement l'arbre de jeu en suivant ces étapes :

1. **Condition de fin** : Si l'état courant s est une feuille (par exemple, une victoire, une défaite ou une égalité) ou si la profondeur maximale d est atteinte, la fonction retourne la valeur d'évaluation de l'état courant, donnée par la fonction Eval.
2. **Tour du joueur maximisant** :
   — La meilleure valeur (`bestValue`) est initialisée à $-\infty$.
   — Pour chaque état successeur s', l'algorithme appelle Alpha-Bêta récursivement.
   — Si la valeur retournée dépasse ou égale β, l'élagage est appliqué car le joueur minimisant ne permettra jamais d'atteindre cette branche.
   — Sinon, la valeur de α est mise à jour si un meilleur choix est trouvé.
3. **Tour du joueur minimisant** :
   — La meilleure valeur (`bestValue`) est initialisée à $+\infty$.
   — Pour chaque état successeur s', l'algorithme appelle Alpha-Bêta récursivement.
   — Si la valeur retournée est inférieure ou égale à α, l'élagage est appliqué car le joueur maximisant ne permettra jamais d'atteindre cette branche.

— Sinon, la valeur de β est mise à jour si un meilleur choix est trouvé.

Avantages de l'élagage L'élagage Alpha-Bêta permet de ne pas explorer des branches inutiles de l'arbre, réduisant ainsi le nombre de nœuds évalués :

— Dans le meilleur des cas (exploration ordonnée des successeurs), l'algorithme peut réduire la complexité de $O(b^d)$ à $O(b^{d/2})$, où b est le facteur de branchement et d est la profondeur.
— Cela permet d'explorer l'arbre plus profondément pour une même quantité de ressources.

Résultat À la fin, l'algorithme retourne la valeur utilitaire optimale pour l'état courant s, permettant au joueur de prendre la meilleure décision possible en fonction de la profondeur spécifiée et des règles d'élagage.

6.3 Complexité

L'algorithme Alpha-Bêta améliore la complexité temporelle de l'algorithme Minimax en permettant d'élaguer des branches inutiles de l'arbre de jeu. La complexité temporelle dans le meilleur des cas (lorsque l'élagage est optimal) est $O(n^{\frac{d}{2}})$, où n est le facteur de branchement et d est la profondeur maximale. Dans le pire des cas (lorsqu'il n'y a pas d'élagage), la complexité reste $O(n^d)$, ce qui est équivalent à celle de Minimax. En pratique, l'élagage peut réduire significativement le temps de calcul, surtout pour des arbres de jeu très larges.

6.4 Limites et Extensions

Malgré l'élagage efficace, l'algorithme Alpha-Bêta reste sensible à l'explosion combinatoire pour des jeux ayant un grand nombre de possibilités. Cependant, il est plus performant que Minimax pour des arbres de jeu de taille importante.

Les extensions possibles de cet algorithme incluent :

— **Évaluation adaptative** : Utiliser des fonctions d'évaluation spécifiques pour les jeux à grande échelle.
— **Optimisation parallèle** : Exploiter les ressources parallèles pour explorer différents sous-arbres en parallèle.

Algorithme 2 : ALPHA-BÊTA Algorithm

Input : s : current state ; d : max depth maximale ;
$player_1$: Indicates wheter it is the maximizing player's turn ;
α : max value for player 1 ; β : min value for player 2.
Output : `bestValue` : The utility value for the current state.

```
1  begin
2      if isLeaf(s) ∨ d = 0 then
3          return EVAL(s)
4      if player_1 then
5          bestValue ← −∞ ;
6          foreach s' ∈ child(s) do
7              bestValue ← max(bestValue,
               ALPHA-BÊTA(s', d − 1, false, α, β)) ;
8              if bestValue ≥ β then
9                  return bestValue ;                    // prunning
10             if bestValue > α then
11                 α← bestValue ;
12         return bestValue
13     else
14         bestValue ← +∞ ;
15         foreach s' ∈ child(s) do
16             bestValue ← min(bestValue,
               ALPHA-BÊTA(s', d − 1, true, α, β)) ;
17             if bestValue ≤ α then
18                 return bestValue ;                    // prunning
19             if bestValue < β then
20                 β← bestValue ;
21         return bestValue
```