ÉPREUVE COMMUNE DE TIPE 2013 - Partie D

TITRE :

Preuve de la correction de l'algorithme alpha-bêta

Temps de préparation : ……2 h 15 minutes
Temps de présentation devant les examinateurs : …….10 minutes
Dialogue avec les examinateurs : ……..10 minutes

GUIDE POUR LE CANDIDAT :

Le dossier ci-joint comporte au total : 13 pages
Document principal (11 pages, dont celle-ci) ; annexe : 2 pages

Travail **suggéré** au candidat :

Après une synthèse du document, le candidat pourra, au choix, orienter sa présentation sur :

- le principe des démonstrations par induction structurelle, en l'illustrant avec l'une des démonstrations effectuées dans ce document ;
- l'intérêt pratique des algorithmes présentés, ainsi que leurs limites.

Attention : s*i le candidat préfère effectuer un autre travail sur le dossier, il lui est* ***expressément recommandé*** *d'en informer le jury avant de commencer l'exposé.*

CONSEILS GENERAUX POUR LA PREPARATION DE L'EPREUVE :

* Lisez le dossier en entier dans un temps raisonnable.
* Réservez du temps pour préparer l'exposé devant les examinateurs.

- Vous pouvez écrire sur le présent dossier, le surligner, le découper … mais tout sera à remettre aux examinateurs en fin d'oral.
- En fin de préparation, rassemblez et ordonnez soigneusement TOUS les documents (dossier, transparents, *etc.*) dont vous comptez vous servir pendant l'oral. En entrant dans la salle d'oral, vous devez être prêt à débuter votre exposé.
- A la fin de l'oral, vous devez remettre aux examinateurs le présent dossier dans son intégralité. Tout ce que vous aurez présenté aux examinateurs pourra être retenu en vue de sa destruction.

IL EST INTERDIT DE SORTIR LE DOSSIER DU SITE DE L'EPREUVE

Preuve de la correction de l'algorithme alpha-bêta

Introduction

Un domaine particulier de l'intelligence artificielle est celui de la programmation des jeux à deux joueurs, comme le jeu de go, celui des échecs, etc. C'est précisément un grand précurseur de l'informatique et de l'intelligence artificielle, Alan Turing, qui, au début des années 1950, conçut le premier programme capable de jouer aux échecs. Cependant, Alan Turing ne disposant pas d'une machine suffisamment puissante pour exécuter son programme, devait l'exécuter à la main et chaque mouvement nécessitait plus d'une demi-heure de calculs. Une partie historique a été enregistrée en 1952 : le programme, qui jouait contre un collègue de Turing, a perdu la partie. Il faut attendre 1956 pour qu'un programme informatique batte un joueur humain. Le programme était exécuté sur une énorme machine (MANIAC1, construite au laboratoire américain de Los Alamos dans le cadre du programme de développement de l'arme nucléaire) qui pouvait effectuer jusqu'à 10 000 instructions par seconde. Cependant, son adversaire humain ne connaissait le jeu des échecs que depuis une semaine !

Le principe de ces programmes était d'exploiter la force de calcul des machines en explorant toutes les possibilités de jeu à partir d'une configuration donnée. En 1958, trois scientifiques de l'université Carnegie-Mellon, Newell, Shaw et Simon, apportèrent une amélioration qualitative notable en inventant l'algorithme *alpha-bêta*, qui permet d'éviter l'exploration de grandes parties de l'arbre de recherche sans affecter le résultat final. Cet algorithme est basé uniquement sur des propriétés mathématiques élémentaires et n'exploite pas les règles du jeu d'échecs : il s'applique donc à tous les jeux à deux joueurs.

Après avoir défini le cadre de l'étude, ce document présente l'algorithme de base, *Minimax*, qui examine tous les coups possibles, puis il présente et prouve la correction de deux algorithmes simples, *Minimax2* et *alpha-bêta*, conçus pour limiter le nombre de coups possibles que doit examiner un programme informatique à partir d'une position de jeu donnée.

1. Cadre de l'étude

a. Cadre théorique

On considère un jeu se jouant à deux personnes, chaque joueur effectuant tour à tour un unique coup. On supposera qu'aucune partie ne comporte un nombre infini de coups, et qu'à chaque état du jeu, appelé position, il existe un nombre fini de coups légaux. Alors, à chaque position p, il existe un nombre fini $N(p)$ tel que toute partie commençant à partir de p ne comporte pas plus de $N(p)$ coups.

Le but des algorithmes présentés dans ce document est de permettre, étant donnés une position p et un joueur, de choisir un coup "valable" pour ce joueur. L'ensemble des coups possibles à partir de p peut être représenté par un arbre, appelé arbre de jeu :

Définition : *On appelle* “arbre de jeu construit à partir de la position *p*” *l’**arbre***[1] *vérifiant les propriétés suivantes :*

- *La **racine** de l’arbre est la position p ;*
- *Les **nœuds** de l’arbre représentent les positions “atteignables” à partir de p ;*
- *Soit k un nœud de l’arbre :*
  - *si c_1, ..., c_n sont les coups possibles légaux à partir de la position représentée par k et p_1, ..., p_n les positions atteintes en jouant ces coups, alors ces positions sont représentées par des nœuds k_1, ..., k_n qui sont les **fils** du nœud k ;*
  - *si k représente une position terminale, c’est-à-dire pour laquelle plus aucun coup légal n’est possible à partir de la position représentée par k, alors k est une **feuille** de l’arbre.*

L’ensemble des arbres de jeu (pour lesquels on assimilera une position *p* et le nœud qui la représente) est un sous-ensemble défini inductivement. En conséquence certaines propriétés d’un arbre de jeu pourront être prouvées par induction structurelle[2]. C’est à partir de cet arbre que l’on construira et que l’on étudiera les algorithmes choisissant un coup à partir de la position *p*.

Pour une position terminale *p*, une *fonction d’évaluation f* permet d’associer une *valeur* à cette position, valeur que l’on notera par la suite *f* (*p*). Par exemple, cela peut être +1 si la partie se termine par une victoire pour le joueur que le programme représente, –1 pour une défaite, 0 pour un nul. C’est à partir de ces valeurs assignées aux feuilles de l’arbre de jeu que l’algorithme déterminera le coup choisi.

b. Cadre pratique

Dans la pratique, à cause de contraintes de temps de calcul et d’espace mémoire, on ne peut pas explorer un arbre de jeu dans sa totalité. On décide alors que les nœuds à une profondeur *d* de l’arbre sont terminaux. À tous ces nœuds, la fonction d’évaluation associe une estimation de la valeur de la position à l’aide de critères plus ou moins subjectifs. Pour les échecs, l’estimation peut se baser sur la valeur des pièces de part et d’autre, sur leur position, leur mobilité, l’avance de développement, etc. : par exemple, la valeur des pièces peut être de 900 points pour une dame, 500 points pour une tour, 300 points pour un fou ou un cavalier et 100 points pour un pion ; la valeur d’une tour sur une colonne ouverte peut être augmentée ; la valeur d’une pièce “clouée”, *i.e.* d’une pièce qui ne peut pas se déplacer sans exposer une pièce de plus grande valeur à une capture, peut être réduite, etc.

Les problèmes à résoudre sont les mêmes que dans le cadre théorique idéal, mais les contraintes de temps prennent une importance capitale : améliorer le temps de parcours de l’arbre signifie en effet pouvoir explorer une profondeur plus élevée et, ainsi, jouer mieux.

[1] Tous les mots dont la première occurrence apparaît en gras sont définis dans le glossaire en annexe 2.
[2] La définition d’un ensemble défini inductivement et le théorème justifiant les preuves par induction structurelle sont donnés en annexe 1.

2. L’algorithme Minimax

a. Présentation générale

Dans un jeu à deux personnes, les deux joueurs ont des objectifs contradictoires. On suppose par exemple que la fonction d’évaluation donne un score très élevé si un des deux joueurs gagne et un score très faible si le second joueur gagne. Le premier joueur aura intérêt à choisir des coups qui vont conduire à maximiser la fonction d’évaluation des nœuds terminaux. On appellera donc ce joueur *Max*. Réciproquement, le second voudra minimiser la fonction d’évaluation. On l’appellera *Min*.

Comment jouer de façon optimale ? Examinons l’arbre suivant (voir figure 1), de profondeur 2, où le nom du joueur est donné en marge.

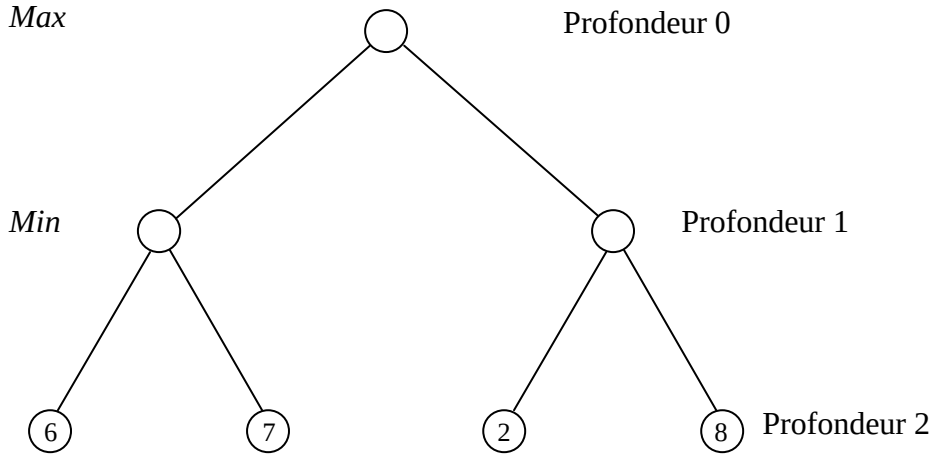

Figure 1 : L’algorithme *Minimax* (évaluation des feuilles)

Si Min joue au niveau 1, il n’a pas à se préoccuper des réponses de Max puisque ses coups donnent des positions finales. Dans la branche la plus à gauche, en jouant au mieux, il peut obtenir 6 (valeur minimale). La valeur de la position située au niveau à gauche peut donc être estimée à 6, puisque, si Min joue au mieux, c’est ce score qu’on atteint à la position terminale. De même, dans la partie la plus à droite, il peut obtenir 2. On remonte donc les valeurs 2 et 6 dans l’arbre, en calculant le minimum des valeurs des positions filles (voir figure 2).

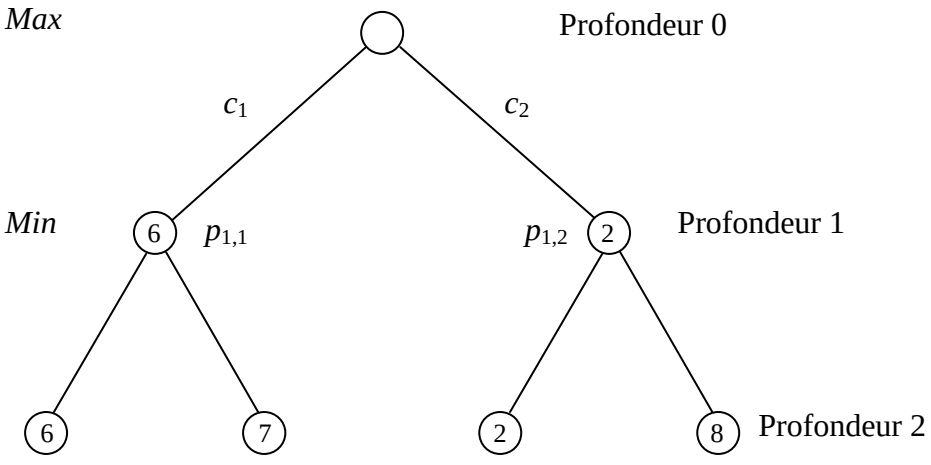

Figure 2 : Principe de l’algorithme *Minimax*

Soit maintenant à Max de joueur au niveau 0. En choisissant le coup c_1, il sait que si son adversaire Min joue au mieux, il obtiendra une position terminale valant 6. En choisissant c_2, il sait que cette fois Min pourra jouer de sorte que la position terminale vaille 2. Max choisit donc c_1 avec la certitude d'obtenir une position terminale valant au moins max(2,6) = 6.

Cet algorithme peut facilement être étendu, par induction, à tout un arbre de jeu fini. Il consiste à prendre le maximum des minima, puis le minimum des maxima, etc. C'est pourquoi on l'appelle *l'algorithme du minimax.*

Dans la suite, on appellera *Minimax*(*p*) la valeur calculée en *p* par l'algorithme du minimax.

b. L'algorithme

L'algorithme récursif suivant calcule le minimax d'un arbre de jeu :

```
Fonction Minimax (p)
   Déterminer les successeurs p1,...,pn de p;
   Si n=0 alors retourner f(p)
   Sinon
      Si p est un noeud où on maximise
       alors retourner Max(Minimax(p1),..., Minimax(pn)) ;
       sinon retourner Min(Minimax(p1),..., Minimax(pn)) ;
```

Remarque : Cet algorithme n'est pas forcément efficace contre de mauvais joueurs. Reprenons l'exemple précédent : si Min joue mal, alors si Max joue c_1, il n'obtiendra que 7, tandis que s'il avait choisi c_2, il aurait pu obtenir 8. Toutefois c'est une hypothèse raisonnable que de considérer que son adversaire joue le meilleur coup possible.

Si on peut explorer l'arbre en entier, et si l'information sur les positions terminales est totale (*i. e.* on sait déterminer si un des deux joueurs a gagné ou si c'est un nul), le minimax joue les coups optimaux pour les deux joueurs. Mais comme on l'a précédemment évoqué, dans les adaptations pratiques, c'est rarement le cas.

3. Une première amélioration

a. L'idée

On peut calculer le Minimax d'un arbre (*i. e.* le Minimax de sa racine) sans être obligé de calculer la valeur affectée à chaque nœud. A cet effet, examinons l'exemple suivant (voir figure 3). La formule de récurrence nous dit que *Minimax*(p) = max(*Minimax*(p_1), *Minimax*(p_2)). Or, sachant que *Minimax*($p_{2,1}$) = 6, on sait que *Minimax*(p_2) ≤ 6 (p_2 est un nœud où on minimise). On peut alors rejeter le coup c_2 sans estimer les *Minimax*($p_{2,i}$) pour $i > 1$, puisque leur valeur ne changera pas la valeur donnée à *Minimax*(p), ni le coup choisi.

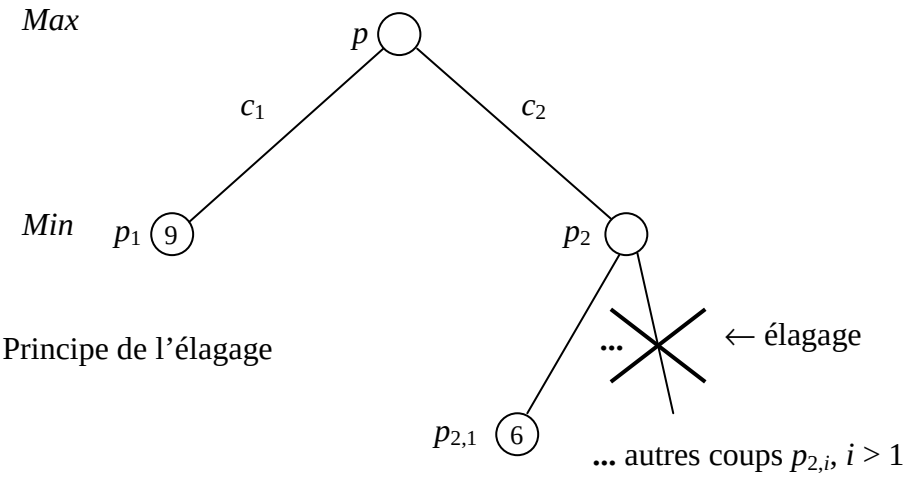

Figure 3 : Principe de l'élagage

b. L'algorithme

L'algorithme suivant implémente cette amélioration qui permet d'élaguer l'arbre :

```
Fonction Minimax2 (position p, entier compare) : entier ;
   Déterminer les successeurs p1,...,pn de p;
   Si n=0 alors retourner f(p)
   Sinon
      Si p est un noeud où on maximise
      alors
         max ← -infini ;
         fin ← faux ;
         i ← 1 ;
         Tant que (i<=n) et (fin=faux) faire {
            v ← Minimax2(pi,max) ;
            si v>max alors max ← v ;              // Mise à jour du maximum
            si max>=compare alors fin=vrai ;          // élagage de l'arbre
            i ← i+1                             } ;
         Retourner max
      sinon
         min ← +infini ;
         fin ← faux ;
         i ← 1 ;
         Tant que (i<=n) et (fin=faux) faire {
            v ← Minimax2(pi,min) ;
            si v<min alors min ← v ;
            si min<=compare alors fin=vrai ;
            i ← i+1                             } ;
         Retourner min ;
```

Lorsqu'on appelle *Minimax2* pour un nœud *k* qui minimise (comme le nœud p_2 de la figure 3), comme le père de *k* est un nœud qui maximise, la variable `compare` est le maximum des nœuds "frères" déjà visités (par exemple pour p_2, la variable `compare` vaut 9). On arrête les évaluations des nœuds fils de *k* dès que l'une d'entre elles, `v`, devient inférieure ou égale à `compare` (car l'évaluation de *k* sera forcément inférieure ou égale à `v`, donc inférieure ou égale à `compare`, donc le nœud père de *k*, qui maximise, ne retiendra pas le coup qui mène à *k*). De même, sur un nœud qui

maximise, la variable `compare` est le minimum des nœuds "frères" déjà visités et on arrête les évaluations des nœuds fils dès que l'une d'entre elles, v, devient supérieure ou égale à `compare`.

Lorsqu'on appelle initialement *Minimax2* pour la racine *p* de l'arbre, si *p* est un nœud qui maximise (respectivement minimise), le paramètre `compare` a pour valeur $+\infty$ (respectivement $-\infty$).

c. Preuve de la correction de l'algorithme *Minimax2*

Lemme : *En utilisant l'algorithme Minimax2* :

- *Si p est un nœud où on maximise* :
  - *Si Minimax*(p) ≤ `compare` *alors Minimax2*(p,`compare`) = *Minimax*(p)
  - *Si Minimax*(p) > `compare` *alors Minimax2*(p,`compare`) ≥ `compare`
- *Si p est un nœud où on minimise* :
  - *Si Minimax*(p) ≥ `compare` *alors Minimax2*(p,`compare`) = *Minimax*(p)
  - *Si Minimax*(p) < `compare` *alors Minimax2*(p,`compare`) ≤ `compare`

Démonstration : On démontre ce lemme par induction structurelle.

Base : Si *p* est une feuille, alors quelle que soit la valeur de `compare` on a *Minimax2*(p,`compare`) = *Minimax*(p) = f (p), ce qui est bien compatible avec tous les cas du lemme.

Induction : Soit un nœud *p* ayant *n* nœuds fils p_1, ..., p_n pour lesquels on suppose (hypothèse d'induction structurelle) que le lemme est vérifié.

Si *p* est un nœud qui maximise, notons m_i la valeur (si elle existe) de la variable `max` après l'exécution de la ligne commentée "`Mise à jour du maximum` " à la $i^{\text{ème}}$ itération. Montrons, par récurrence sur *i*, que pour tout $i \in [\![1, n]\!]$, si on exécute la $i^{\text{ème}}$ itération de la boucle, on a m_i = max(*Minimax*(p_1), ..., *Minimax*(p_i)) :

- Si $i = 1$, on a m_1 = v = *Minimax2*(p_1, $-\infty$). Comme, par hypothèse d'induction structurelle, p_1 vérifie le lemme, on a forcément *Minimax2*(p_1, $-\infty$) = *Minimax*(p_1) car p_1 est un nœud qui minimise et *Minimax*(p_1) > $-\infty$. La propriété est donc vérifiée pour $i = 1$.
- S'il existe *i* tel que m_i = max(*Minimax*(p_1), ..., *Minimax*(p_i)), alors :
  - Si *Minimax*(p_{i+1}) < m_i, alors, par hypothèse d'induction structurelle sur p_{i+1} (on applique le lemme à p_{i+1}), on a v = *Minimax2*(p_{i+1},m_i) ≤ m_i et donc m_{i+1} = max(m_i, v) = m_i.
    Comme, par hypothèse de récurrence, m_i = max(*Minimax*(p_1), ..., *Minimax*(p_i)) et comme *Minimax*(p_{i+1}) < m_i, alors max(*Minimax*(p_1), ..., *Minimax*(p_{i+1})) = m_i = m_{i+1}.
  - Si *Minimax*(p_{i+1}) ≥ m_i, alors, par hypothèse d'induction structurelle sur p_{i+1}, on a v = *Minimax2*(p_{i+1},m_i) = *Minimax*(p_{i+1}) et donc :
    m_{i+1} = max(m_i, v) = max(max(*Minimax*(p_1), ..., *Minimax*(p_i)), *Minimax*(p_{i+1}))
    = max(*Minimax*(p_1), ..., *Minimax*(p_{i+1})).

La propriété démontrée ($\forall\ i \in [\![1, n]\!]$, m_i = max(*Minimax*(p_1), ..., *Minimax*(p_i))) nous permet d'achever la démonstration du lemme :

- Si *Minimax*(p) ≤ `compare`, alors $\forall\ i \in [\![1, n]\!]$, *Minimax*($p_i$) ≤ `compare`,

donc $\forall\, i \in [\![1, n]\!]$, $m_i \leq$ `compare`,

donc *Minimax2*(*p*, `compare`) = m_n = max(*Minimax*(p_1), ..., *Minimax*(p_n)) = *Minimax*(*p*).

- Si *Minimax*(*p*) > `compare`, il existe un i_0, le plus petit possible, tel que *Minimax*(p_{i_0}) > `compare`. On a donc *Minimax2*(*p*, `compare`) = $m_{i_0} \geq$ `compare`.

La démonstration dans le cas où *p* est un nœud qui minimise est analogue. ∎

Le théorème ci-dessous, qui prouve la correction de l'algorithme *Minimax2*, découle immédiatement du lemme précédent :

Théorème : *L'algorithme Minimax2 est correct parce que* :

- *si p est un nœud qui maximise, alors Minimax2*(*p*, +∞) = *Minimax*(*p*) ;
- *si p est un nœud qui minimise, alors Minimax2*(*p*, –∞) = *Minimax*(*p*).

4. L'algorithme alpha-bêta

a. L'idée

L'algorithme précédent peut être amélioré en consultant encore moins de nœuds. L'idée est de transmettre récursivement une borne supérieure *et* une borne inférieure, et non plus seulement l'un ou l'autre. Des coupures plus profondes vont pouvoir être réalisées.

En effet, avec l'algorithme *Minimax2*, une position *p* n'a comme information que le Minimax de ses frères directs pour éviter de parcourir tous ses fils. Or, il est parfois utile d'avoir une idée du Minimax des frères de ses ascendants comme l'illustre la figure 4. On y voit que *p* (le nœud Min à la profondeur 0), après avoir évalué le nœud p_1, ne sélectionnera pas de nœud dont l'évaluation est supérieure à 7. Cette valeur de 7 est associée au paramètre *bêta* de l'algorithme et représente, pour le nœud p_2, le minimum des évaluations effectuées par son nœud père. Cependant, ce paramètre est utile à tous les descendants Max de profondeur supérieure : par exemple, p_{221}, après avoir évalué p_{2211} à 9, peut se dispenser d'évaluer ses autres nœuds fils car 9 > *bêta* = 7. La raison en est la suivante : l'évaluation de p_{221} est supérieure ou égale à 9 ; donc, dans le cas où p_{22} aurait d'autres fils dont l'évaluation serait inférieure à 9, le nœud p_{221} n'aurait pas été retenu ; et sinon, une valeur supérieure ou égale à 9 aurait été remontée à p_2 et n'aurait donc, de toute façon, pas été retenue par *p* qui "tient" déjà un nœud fils avec une évaluation inférieure (7). Le paramètre *bêta* représente donc la plus petite évaluation "détenue" par un ancêtre Min. Pour un nœud Max, descendant de cet ancêtre, il est donc inutile de continuer à évaluer ses fils quand l'évaluation de l'un d'entre eux dépasse *bêta* car il ne sera, dans tous les cas, jamais sélectionné. De manière duale, le paramètre *alpha* désignera la plus grande évaluation "détenue" par un ancêtre Max, et tout nœud Min descendant de cet ancêtre arrêtera l'évaluation de ses fils dès que l'évaluation de l'un d'entre eux sera inférieure ou égale à *alpha*. Les paramètres *alpha* et *bêta* sont passés en paramètre à chaque appel récursif et leurs valeurs dans l'appel initial sont respectivement –∞ et +∞.

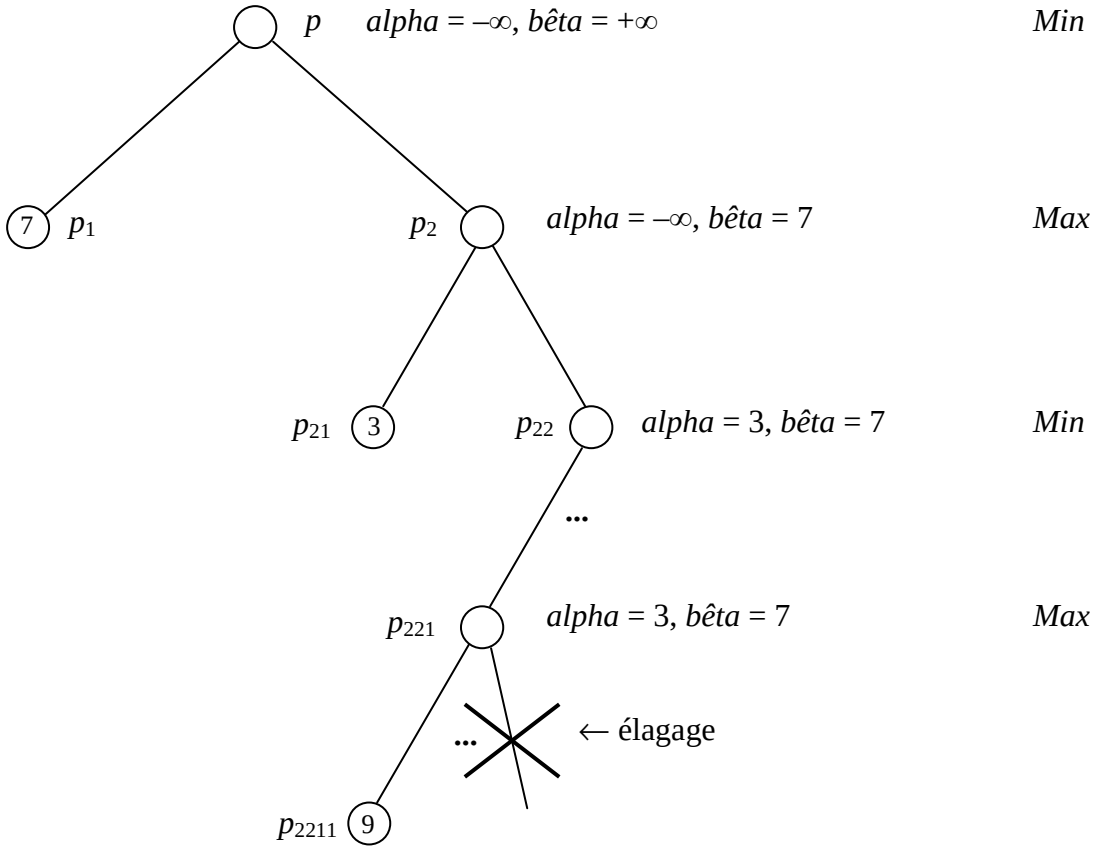

Figure 4 : Principe de l'algorithme *alpha-bêta*

b. L'algorithme

L'algorithme suivant implémente le principe décrit ci-dessus :

```
Fonction MinimaxAB (position p, entier alpha,beta) : entier ;
   Déterminer les successeurs p1,...,pn de p;
   Si n=0 alors retourner f(p)
   Sinon
      Si p est un noeud où on maximise
      alors
         max ← alpha ;
         fin ← faux ;
         i ← 1 ;
         Tant que (i<=n) et (fin=faux) faire {
            v ← MinimaxAB(pi,max,beta) ;
            si v>max alors max ← v ;              // Mise à jour du maximum
            si max>=beta alors fin=vrai ;             // élagage de l'arbre
            i ← i+1                            } ;
         Retourner max
      sinon
         min ← beta ;
         fin ← faux ;
         i ← 1 ;
         Tant que (i<=n) et (fin=faux) faire {
            v ← MinimaxAB(pi,alpha,min) ;
            si v<min alors min ← v ;
            si min<= alpha alors fin=vrai ;
            i ← i+1                            } ;
         Retourner min ;
```

c. Preuve de la correction de l'algorithme alpha-bêta

Lemme : *Avec l'algorithme MinimaxAB, pour toute position p* :

- *Si Minimax* $(p) \leq alpha$, *alors MinimaxAB*$(p,alpha,bêta) \leq alpha$;
- *Si* $alpha \leq$ *Minimax* $(p) \leq bêta$, *alors MinimaxAB*$(p,alpha,bêta) =$ *Minimax* (p) ;
- *Si Minimax* $(p) \geq bêta$, *alors MinimaxAB*$(p,alpha,bêta) \geq bêta$.

Démonstration : La démonstration se fait par induction structurelle.

Base : Si p est une feuille, alors quelles que soient les valeurs de *alpha* et *bêta* on a *MinimaxAB*$(p,alpha,bêta)$ = *Minimax*$(p) = f(p)$, ce qui est bien compatible avec tous les cas du lemme.

Induction : Soit un nœud p ayant n nœuds fils p_1, ..., p_n pour lesquels on suppose (hypothèse d'induction structurelle) que le lemme est vérifié. Considérons un nœud p qui maximise et notons m_i la valeur (si elle existe) de la variable `max` après l'exécution de la ligne commentée "`Mise à jour du maximum` " à la $i^{\text{ème}}$ itération. Au vu des deux instructions qui mettent à jour les variables `v` et `max`, on a, en posant $m_0 = alpha$: $\forall\ i \in [\![1, n]\!]$, $m_i = \max\left(m_{i-1}, MinimaxAB(p_i, m_{i-1}, bêta)\right)$. La suite des valeurs m_i est donc une suite croissante.

- Premier cas : *Minimax* $(p) \leq alpha$

  Comme *Minimax* $(p) = \max\limits_{1\leq i\leq n}$ *Minimax* (p_i), on a : $\forall\ i \in [\![1, n]\!]$, *Minimax* $(p_i) \leq alpha$.

  Montrons par récurrence que $\forall\ i \in [\![1, n]\!]$, $m_i = alpha$.

  La propriété est immédiatement vraie pour $i = 0$.

  Supposons qu'il existe $k \in [\![1, n]\!]$ tel que $m_k = alpha$. Alors $m_{k+1} = \max\left(m_k, MinimaxAB(p_{k+1}, m_k, bêta)\right) = \max\left(alpha, MinimaxAB(p_{k+1}, alpha, bêta)\right)$. Or, par hypothèse d'induction structurelle, le lemme s'applique à p_{k+1} : comme *Minimax* $(p_{k+1}) \leq alpha$, alors *MinimaxAB*$(p_{k+1}, alpha, bêta) \leq alpha$. Par conséquent, $m_{k+1} = alpha$.

  Comme $\forall\ i \in [\![1, n]\!]$, $m_i \leq alpha$, finalement *MinimaxAB*$(p,alpha,bêta) = m_n \leq alpha$.

- Deuxième cas : $alpha \leq$ *Minimax* $(p) \leq bêta$

  Il existe $i_0 \in [\![1, n]\!]$ tel que $alpha \leq$ *Minimax* $(p_{i_0}) \leq bêta$ et $\forall\ i < i_0$, *Minimax* $(p_i) < alpha$.

  D'après ce qui précède, $m_0 = ... = m_{i_0-1} = alpha$. Donc $m_{i_0} = \max\left(alpha, MinimaxAB(p_{i_0}, alpha, bêta)\right)$. Or, par hypothèse d'induction structurelle, le lemme s'applique à p_{i_0} : comme $alpha \leq$ *Minimax* $(p_{i_0}) \leq bêta$, alors *MinimaxAB*$(p_{i_0}, alpha, bêta) =$ *Minimax* (p_{i_0}) et donc

  $$m_{i_0} = \text{Minimax}\ (p_{i_0}) = \max_{1\leq k\leq i_0} \text{Minimax}\ (p_k).$$

  Montrons par récurrence que $\forall\ i \in [\![i_0, n]\!]$, $m_i = \max\limits_{1\leq k\leq i}$ *Minimax* (p_k).

  La propriété est immédiatement vraie pour $i = i_0$.

  Supposons qu'il existe $i \in [\![i_0, n]\!]$ tel que $m_i = \max\limits_{1\leq k\leq i}$ *Minimax* (p_k). On a $m_{i+1} = \max\left(m_i, MinimaxAB(p_{i+1}, m_i, bêta)\right)$. Or, par hypothèse d'induction structurelle, le lemme s'applique à p_{i+1} :

  - Si *Minimax* $(p_{i+1}) \leq m_i$, alors *MinimaxAB*$(p_{i+1}, m_i, bêta) \leq m_i$

    $$\Rightarrow m_{i+1} = m_i = \max_{1\leq k\leq i+1} \text{Minimax}\ (p_k).$$

- Si $m_i \leq Minimax\ (p_{i+1}) \leq bêta$, $MinimaxAB(p_{i+1}, m_i, bêta) = Minimax\ (p_{i+1})$
  $\Rightarrow m_{i+1} = Minimax\ (p_{i+1}) = \max_{1 \leq k \leq i+1} Minimax\ (p_k)$.

Finalement, $MinimaxAB(p,alpha,bêta) = m_n = \max_{1 \leq k \leq n} Minimax\ (p_k) = Minimax\ (p)$.

- Troisième cas : $Minimax\ (p) \geq bêta$

  Il existe $i_0 \in [\![1, n]\!]$, le plus petit possible, tel que $Minimax\ (p_{i_0}) \geq bêta$. On a : $\forall\ i < i_0$, $m_i < bêta$ et $m_{i_0} = \max\ (m_{i_0-1}, MinimaxAB(p_{i_0}, m_{i_0-1}, bêta))$. Par hypothèse d'induction structurelle, le lemme s'applique à p_{i_0} : $Minimax\ (p_{i_0}) \geq bêta \Rightarrow MinimaxAB(p_{i_0}, m_{i_0-1}, bêta) \geq bêta$, et donc $m_{i_0} \geq bêta$.

  Finalement : $MinimaxAB(p,alpha,bêta) = m_{i_0} \geq bêta$.

  La démonstration dans le cas où p est un nœud qui minimise est analogue. ∎

Le théorème ci-dessous, qui prouve la correction de l'algorithme *alpha-bêta*, découle immédiatement du lemme précédent :

Théorème : *L'algorithme alpha-bêta est correct* : $MinimaxAB(p,-\infty,+\infty) = Minimax(p)$.

Enfin, le lemme ci-dessous montre que l'algorithme *alpha-bêta* explore un sous ensemble strict des positions examinées par l'algorithme *Minimax2*.

Lemme : *Pour un nœud p examiné, si p est un nœud qui maximise, bêta* ≤ `compare` *et si p est un nœud qui minimise, alpha* ≥ `compare`.

Conclusion

Dans un jeu à deux joueurs, la recherche du meilleur coup nécessite de parcourir le plus profondément possible un arbre de jeu. Deux pistes de progrès existent : d'un côté, l'augmentation des performances des ordinateurs, c'est-à-dire ce que l'on appelle communément la "force brute", et de l'autre, l'intelligence artificielle. L'apport de l'algorithme *alpha-bêta* s'inscrit plutôt dans la première approche puisqu'il évite des calculs inutiles uniquement sur la base de propriétés mathématiques générales, sans utiliser les règles du jeu. La seule fonction qui contient de "l'intelligence du jeu" est la fonction d'évaluation.

Les progrès effectués par les programmes d'échecs depuis 1958 sont essentiellement dus à la force brute comme le montre l'évolution du nombre de positions examinées par seconde : de 160 000, en 1983, par Belle (une machine spécialisée, développée aux laboratoires Bell), jusqu'à 200 millions, en 1997, par Deep Blue, le superordinateur construit par IBM qui utilisait plus de 200 circuits intégrés spécifiques et qui reste célèbre pour avoir battu (lors d'un match revanche, en 1997, et hors conditions exigées lors des championnats du monde) le champion du monde d'échecs Garry Kasparov. Même si, en début de partie (où il faut éviter toute faute stratégique d'ouverture) ou bien en fin de partie (où, en cas de léger avantage, il est généralement très difficile de transformer celui-ci en victoire), la puissance de calcul n'est pas suffisante, c'est encore la force brute qui pallie ces faiblesses en faisant appel aux gigantesques capacités de mémorisation que possèdent les ordinateurs actuels : utilisation de bibliothèques d'ouvertures en début de partie (qui

permettent d'éviter tout calcul sur les 15 à 20 premiers coups) ; mémorisation de toutes les configurations de fin de partie (jusqu'à 6 pièces) qui permettent d'annoncer un mat de manière certaine (parfois en plusieurs dizaines de coups !).

Le point faible des programmes d'échecs est donc le milieu de partie, où il y a en général une quarantaine de coups différents possibles. Un être humain va en envisager sérieusement entre deux et cinq, les autres vont immédiatement lui sembler faibles. Si les machines actuelles pouvaient disposer de cette intelligence de la position, elles seraient totalement invincibles, mais ce n'est pas le cas. Ceci explique pourquoi les logiciels de go (qui est un jeu beaucoup plus stratégique que tactique) sont aujourd'hui encore loin d'égaler les performances des programmes d'échecs.

Annexe 1 : Ensembles définis inductivement et preuves par induction structurelle

Soit U un ensemble, appelé *univers*, et considéré comme l'ensemble de tous les éléments possibles.

Définition 1 : Un *schéma d'induction* est un couple $(B, \mathcal{R})$, où $B \subset U$ est appelé la *base* du schéma, et $\mathcal{R} = \{ r_1, ..., r_n$, avec $r_i : U^{k_i} \to U \}$ est un ensemble fini de *règles*, chaque règle r_i associant un élément de U à k_i éléments donnés de U : $r_i : x_1, ..., x_{k_i} \mapsto r_i(x_1, ..., x_{k_i}) \in U$.

Définition 2 : Un sous-ensemble E de U est *défini inductivement* par le schéma $(B, \mathcal{R})$ si et seulement si :

- $B \subset E$;
- pour toute règle $r : U^k \to U$, $x_1, ..., x_k \in E \Rightarrow r(x_1, ..., x_k) \in E$; on dit que l'élément $r(x_1, ..., x_k)$ est *construit* à partir de $x_1, ..., x_k$;
- E ne contient que des éléments construits à partir des seuls éléments de la base en appliquant un nombre fini de fois les règles de $\mathcal{R}$.

Remarque : $\mathbb{N}$ est défini inductivement par $B = \{0\}$ et $\mathcal{R} = \{ n \mapsto n + 1 \}$.

Théorème : *Preuve par induction structurelle*

Soient E un sous-ensemble défini par un schéma d'induction $(B, \mathcal{R})$ et P un prédicat dépendant de $x \in E$. Si les deux conditions suivantes sont vérifiées :

- Base : $\forall x \in B$, P(x) est vrai ;
- Induction : si, pour toute règle $r : U^k \to U$, on a :

$$(\forall i \in [\![1, k]\!], \mathrm{P}(x_i) \text{ vrai} \Rightarrow \mathrm{P}(r(x_1, ..., x_k)) \text{ vrai })$$

Alors $\forall x \in E$, P(x) est vrai.

Remarque : La preuve par récurrence est une preuve par induction structurelle sur $\mathbb{N}$.

Annexe 2 : Les arbres

Un ensemble d'arbres A peut être construit avec un schéma d'induction $(S, \mathcal{R})$ (cf. annexe 1).

La base est un ensemble $S \subset A$, d'éléments appelés *sommets* : les sommets sont donc des arbres.

Les règles de construction $c \in \mathcal{R}$ ont la forme générale suivante :

$$c : \begin{cases} S \times A^n \to A \\ s, a_1, \ldots, a_n \mapsto c(s, a_1, \ldots, a_n) \end{cases}, n \in \mathbb{N}^*.$$

Ainsi, avec un sommet s et n arbres, a_1, ..., a_n, on peut construire un nouvel arbre.

Pour un arbre réduit à un sommet s ou pour un arbre de type c (s, a_1, ..., a_n), s est appelé la *racine* de l'arbre. Les racines des arbres a_1, ..., a_n (notées s_1, ..., s_n sur la figure 5) sont les *nœuds fils* de s et s est leur *nœud père*. Lorsque l'on considère un arbre a quelconque, sa racine est donc l'unique nœud qui n'a pas de nœud père. Un nœud qui n'a pas de nœud fils est appelé une *feuille*.

Les arbres se représentent naturellement sous forme graphique comme dans la figure ci-dessous :

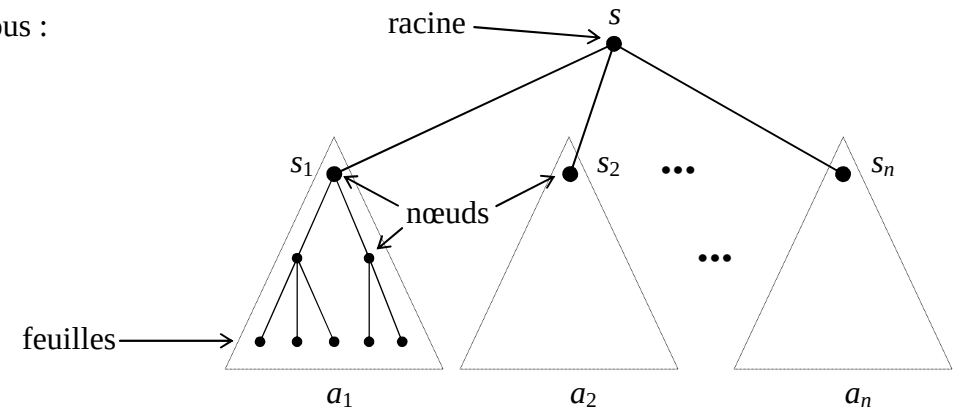

Figure 5 : L'arbre c (s, a_1, ..., a_n)

L'ensemble des arbres de jeu construits à partir d'une position p peut donc être construit avec un schéma d'induction $(S, \mathcal{R})$, où $S = \{p\}$ et où l'ensemble des règles de construction $\mathcal{R}$ se définit avec les règles de mobilité des pièces du jeu d'échec : si p_1, ..., p_n sont des positions accessibles, en un seul coup, à partir de p et si a_1, ..., a_n sont des arbres de jeu construits à partir des positions p_1, ..., p_n, alors c (p, a_1, ..., a_n), est un arbre de jeu construit à partir de la position p dans lequel p_1, ..., p_n sont les nœuds fils de p.

L'arbre de jeu de racine p représente donc l'ensemble des parties d'échecs qui peuvent être jouées à partir de la position p, l'ensemble U, quant à lui, représentant donc l'ensemble des parties d'échecs qui peuvent être jouées.