Programmation des jeux

Jean-Marc Alliot[1]

[1]IRIT

4 juin 2020

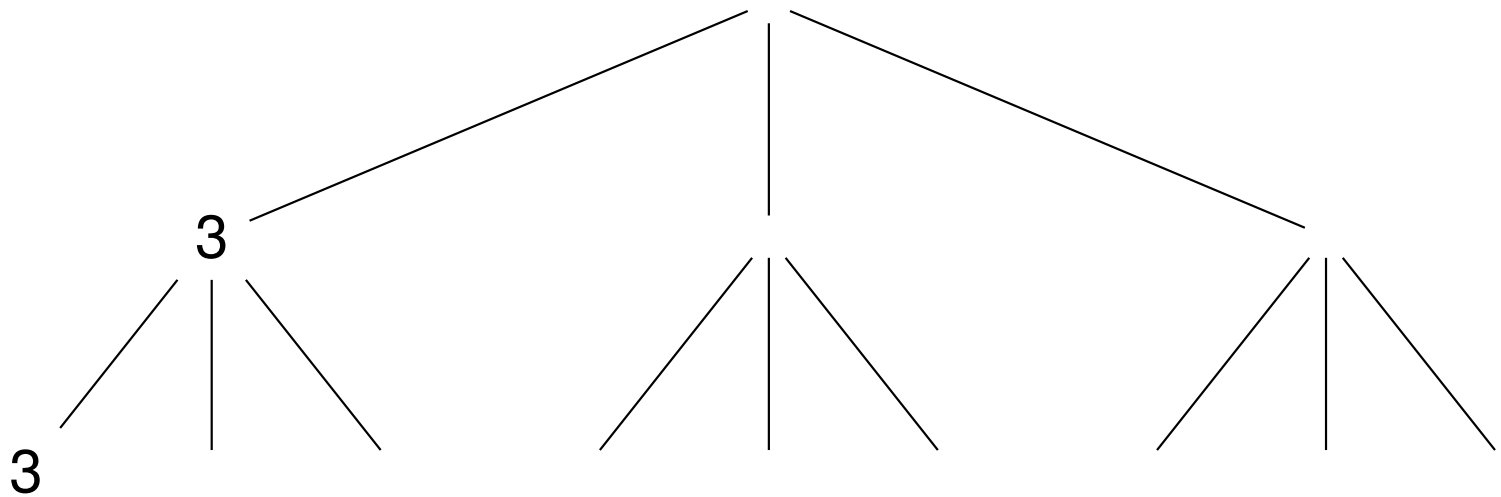
Contenu

Plan

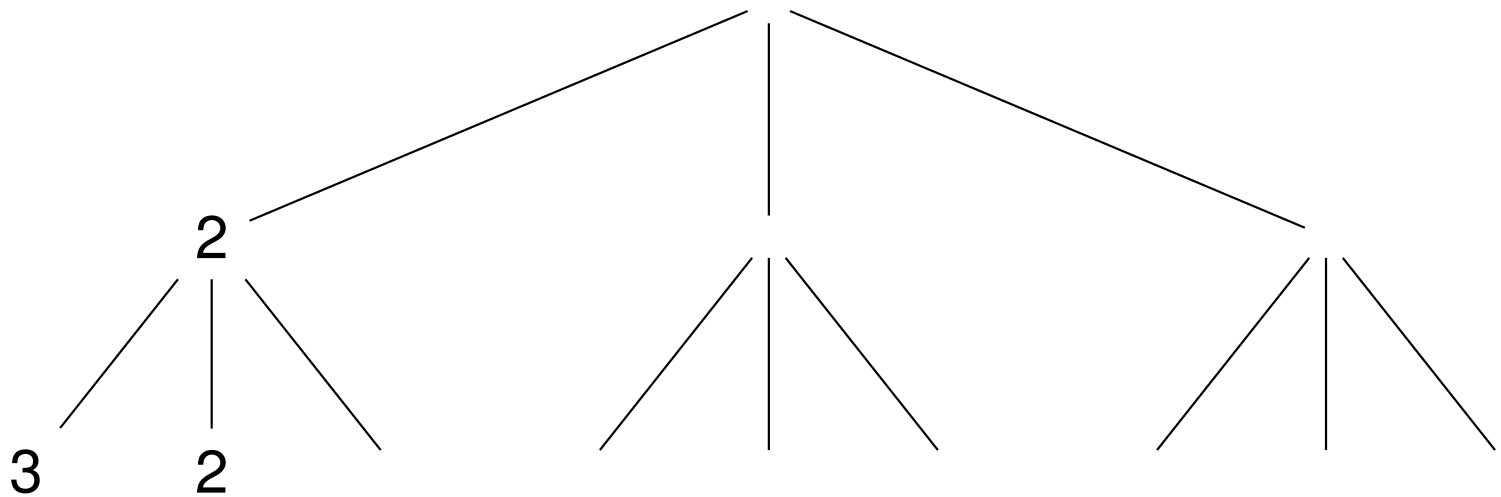
- Fil rouge : le jeu “puissance 4” (7 colonnes, 6 lignes)
- Résolution complète du jeu depuis la dimension 4x4 (minimax) jusqu’à la dimension 7x6 (α-β avec tables de transposition et utilisation des symétries)
- Jeu contre un adversaire avec réponse en temps limité en utilisant un iterα-β avec tables de transposition et fonction d’évaluation heuristique

Minimax

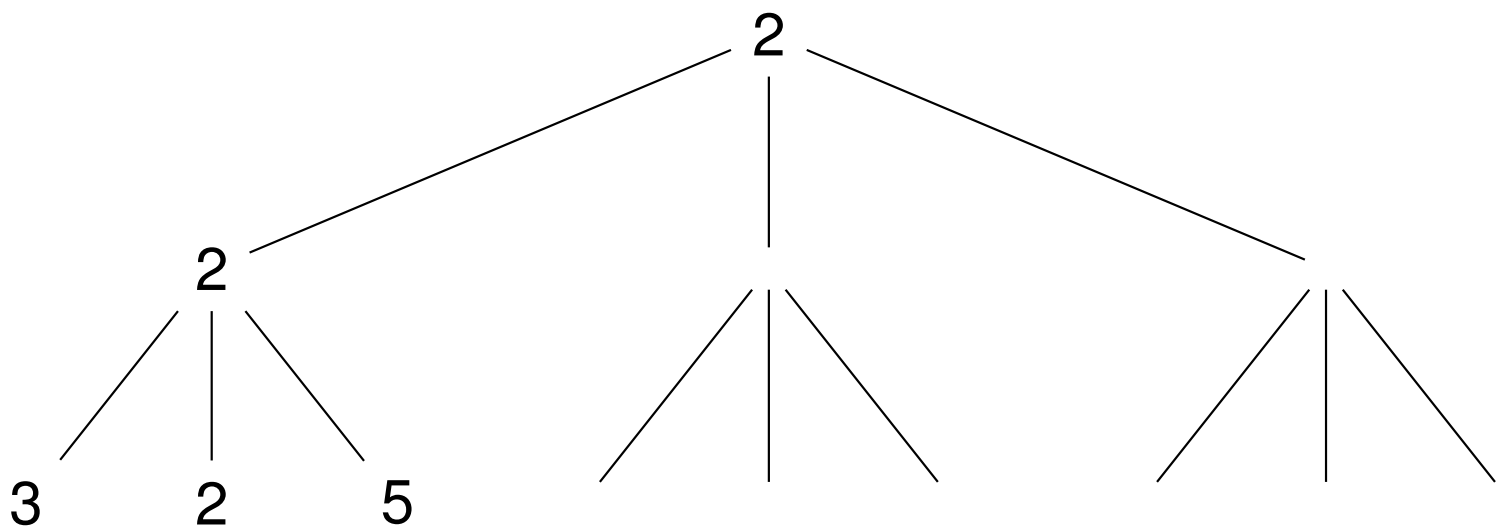
Minimax



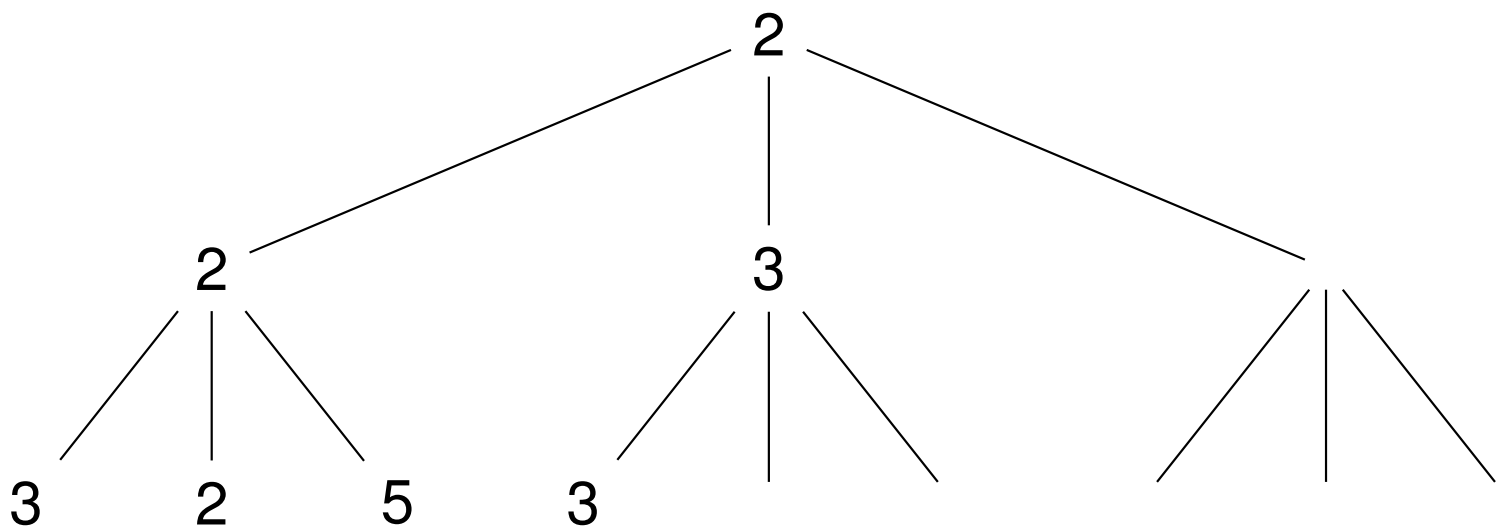
Minimax



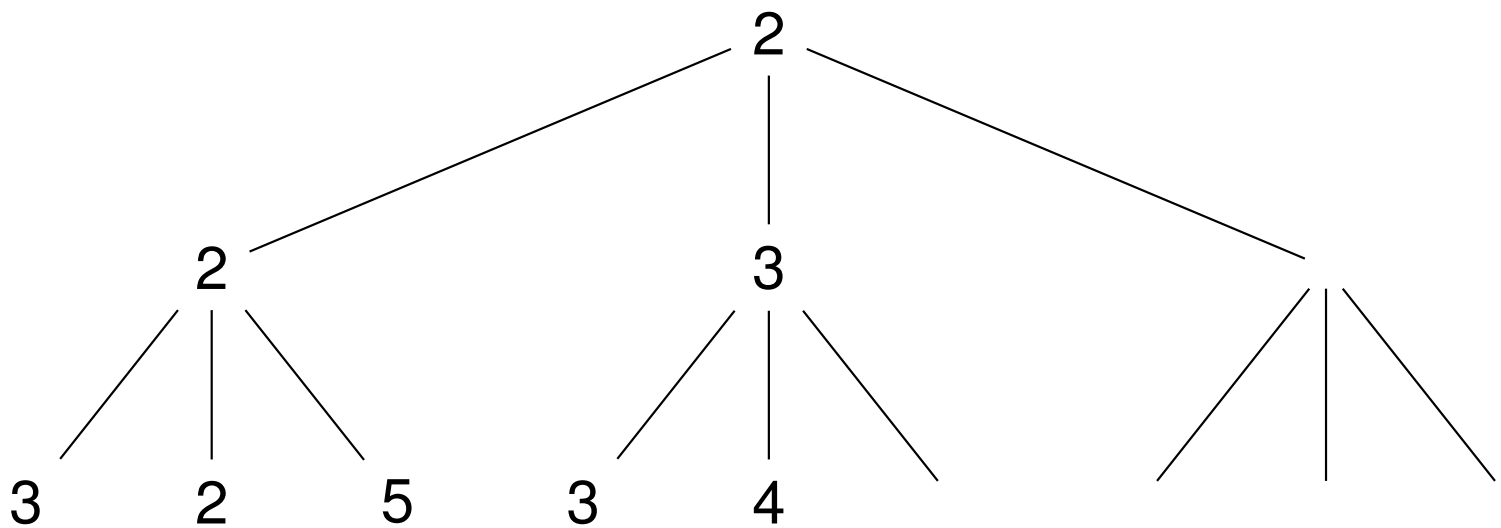
Minimax



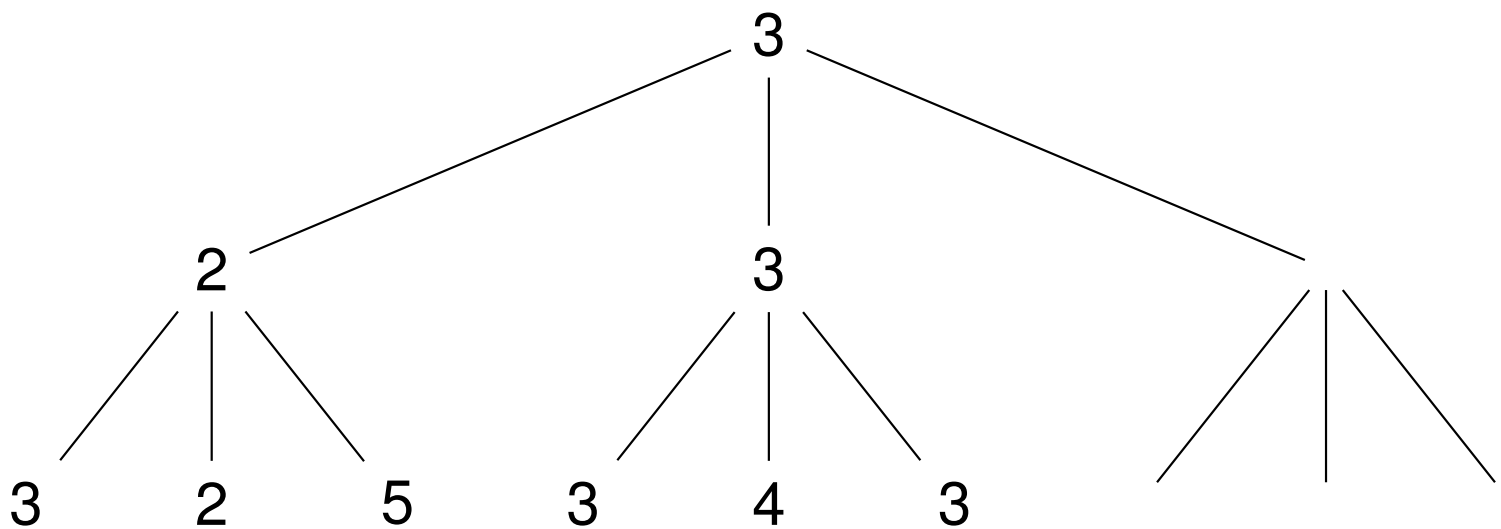
Minimax



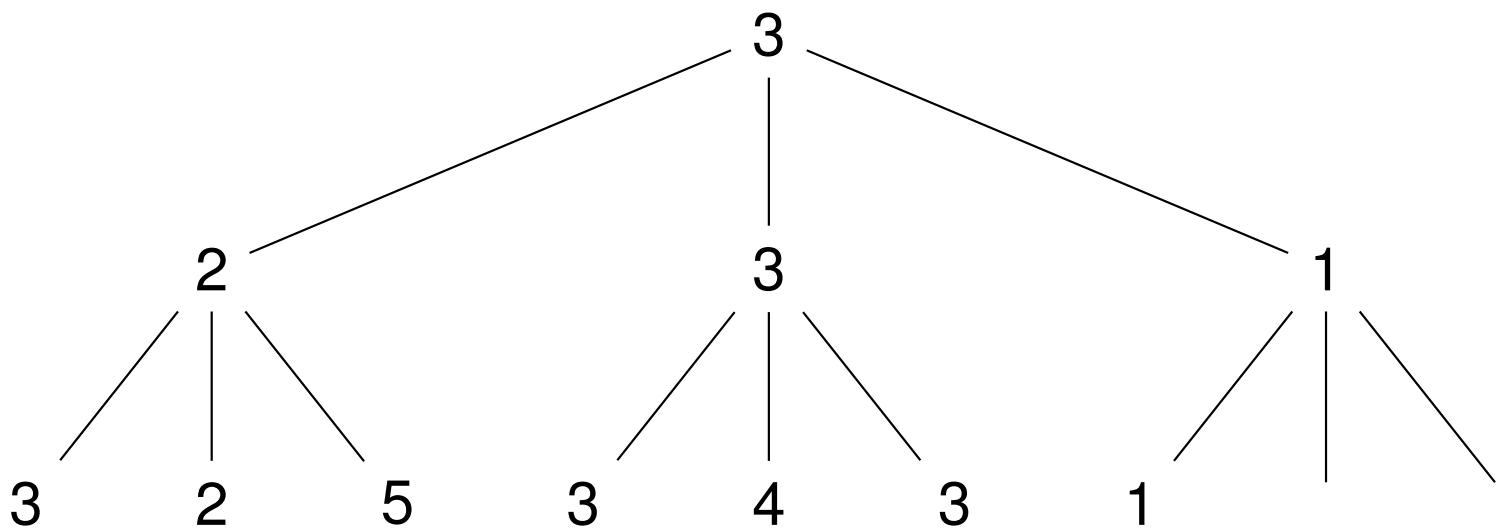
Minimax



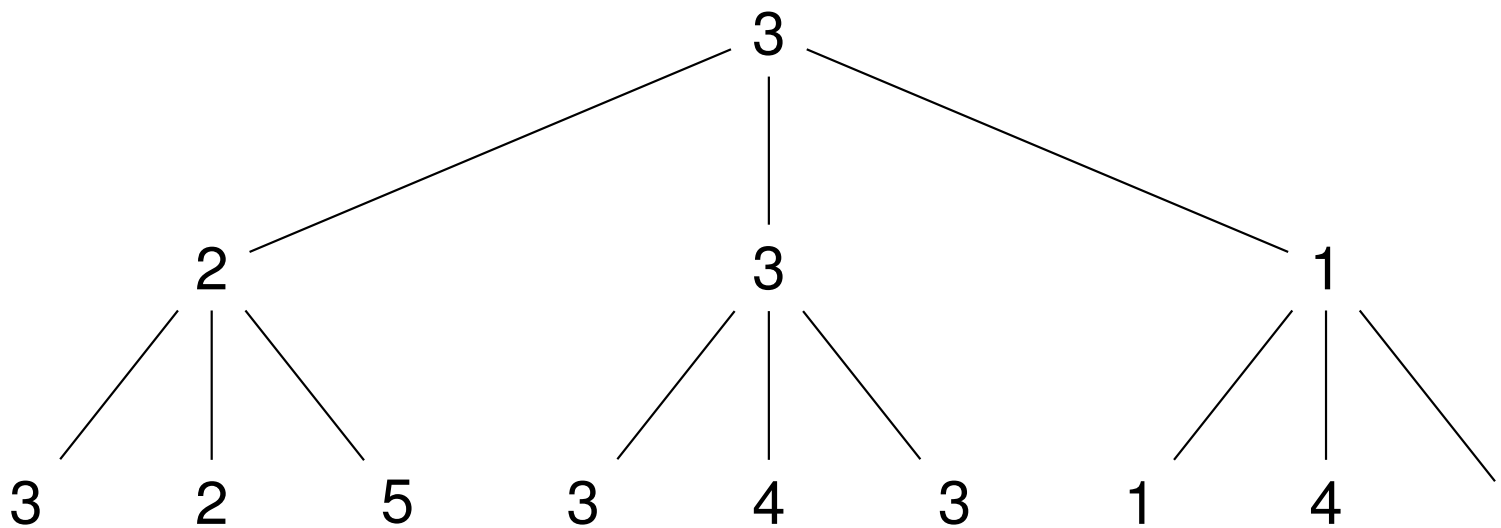
Minimax



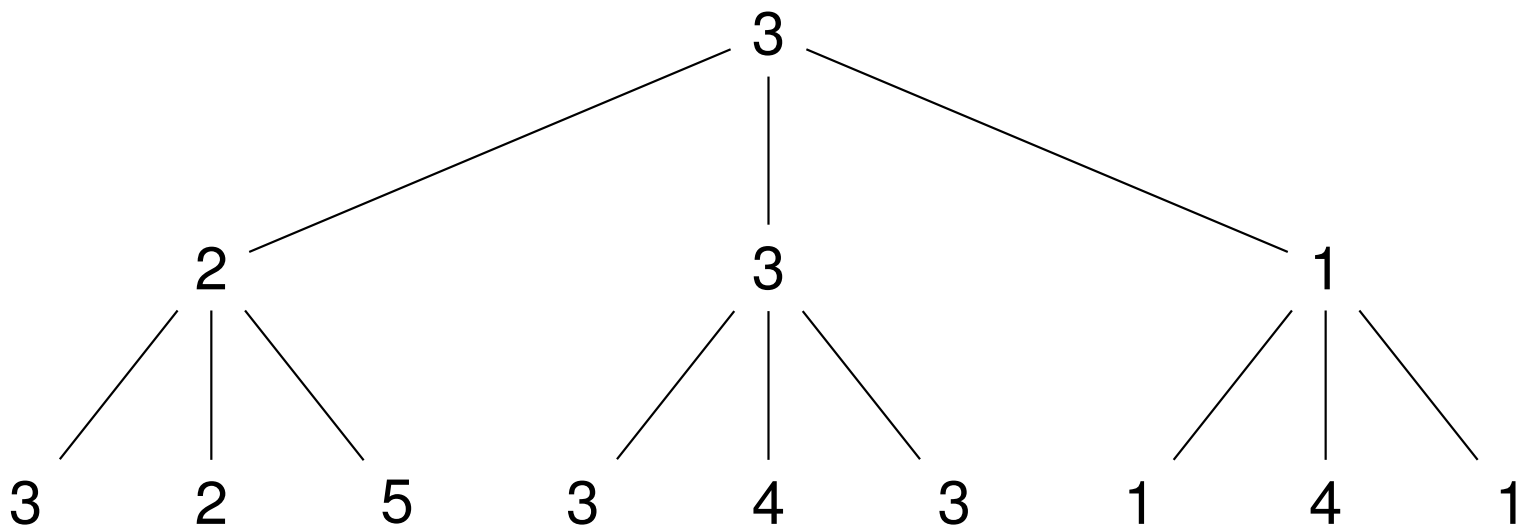
Minimax



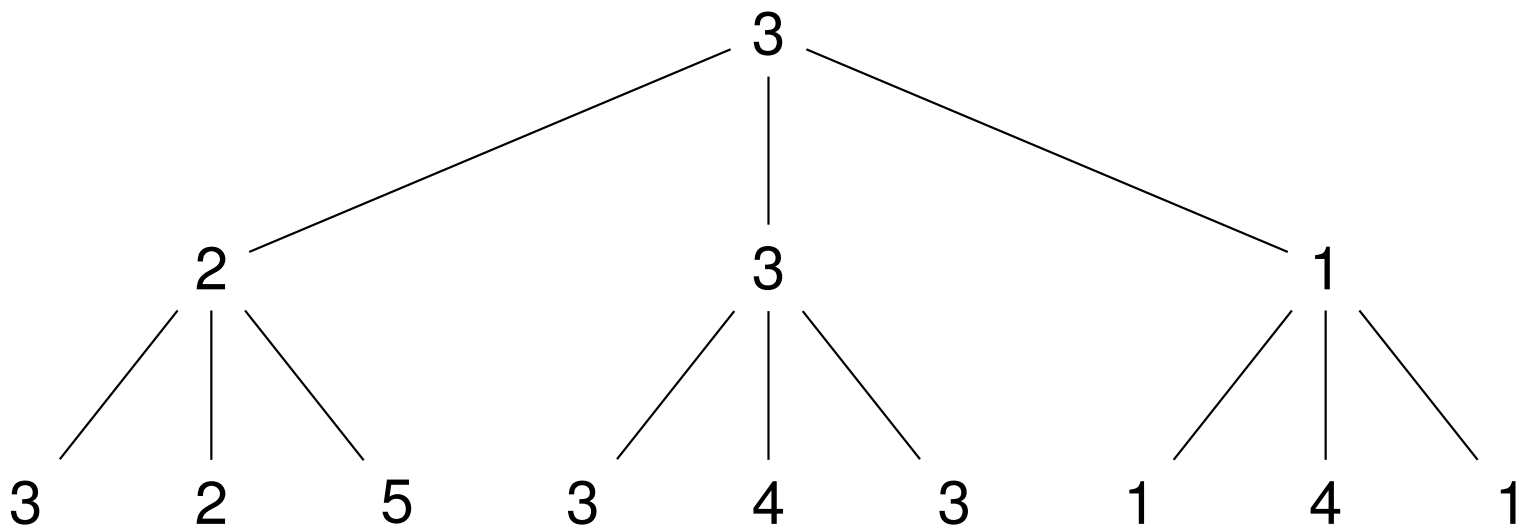
Minimax



Minimax



Minimax



Pseudo-code

Algorithme 1 : Minimax

; ; ; n est le sommet dont on veut calculer la valeur;

`Minimax(`n`)`;
si *n est terminal* **alors retourner** $h(n)$;
si *n est de type Max* **alors**
    Soit $j \leftarrow 1$;
    Soit $g \leftarrow -\infty$;
    Soit $(f_1 \ldots f_k)$ *les fils de n*;
    tant que $(j \leq k)$ **faire**
        $g \leftarrow \max(g,$`Minimax(`f_j`)`$)$;
        $j \leftarrow (j+1)$;
    retourner g;
sinon
    Soit $j \leftarrow 1$;
    Soit $g \leftarrow +\infty$;
    Soit $(f_1 \ldots f_k)$ *les fils de n*;
    tant que $(j \leq k)$ **faire**
        $g \leftarrow \min(g,$`Minimax(`f_j`)`$)$;
        $j \leftarrow (j+1)$;
    retourner g;

Puissance 4 minimax

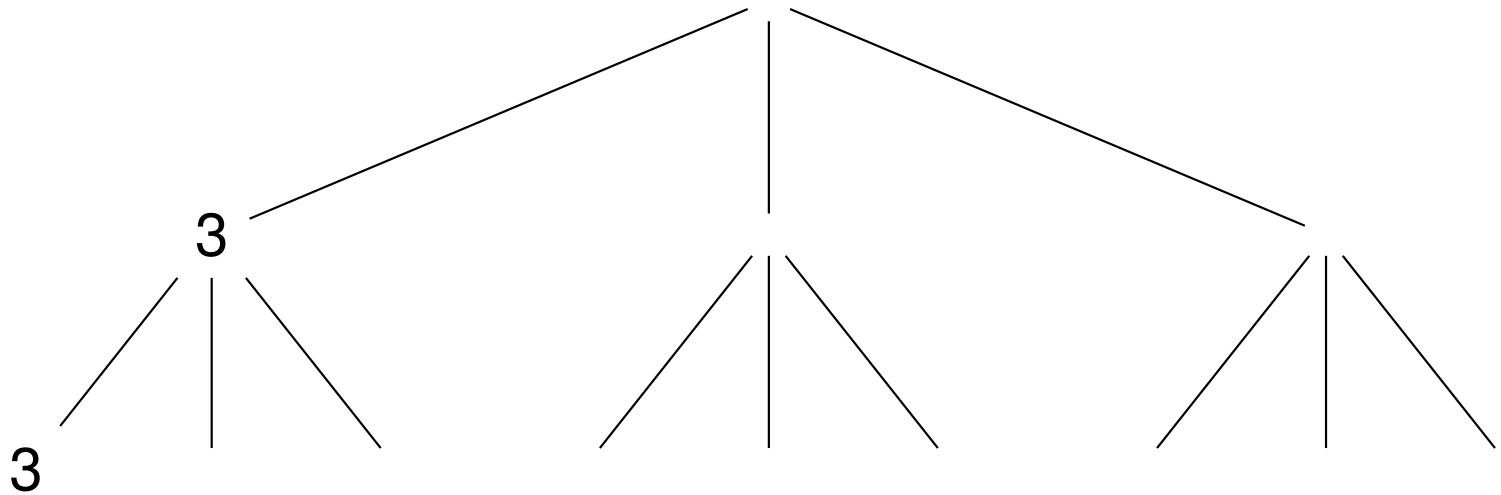
```
int minimax(int color,int depth)
{
  int g,x,y,v;

  /* Test if there is a winning move */
  for (x=0;x<SIZEX;x++) {
    y=first[x];
    if (y!=SIZEY) {v=eval(x,y,color);if (v!=0) return v;}
  }
  /* If no winning move and only one empty square => draw */
  if (depth==(SIZEX*SIZEY-1)) return 0;

  /* Classical minimax */
  if (color==WHITE) g=-32767; else g=32767;
  for (x=0;x<SIZEX;x++) {
    y=first[x];
    if (y!=SIZEY) {
      tab[x][y]=color;
      first[x]++;
      v=minimax(-color,depth+1);
      first[x]--;
      tab[x][y]=EMPTY;
      if (color==WHITE) g=max(g,v);
      else g=min(g,v);
    }
  }
  return g;
}
```

Puissance 4 minimax optimisé

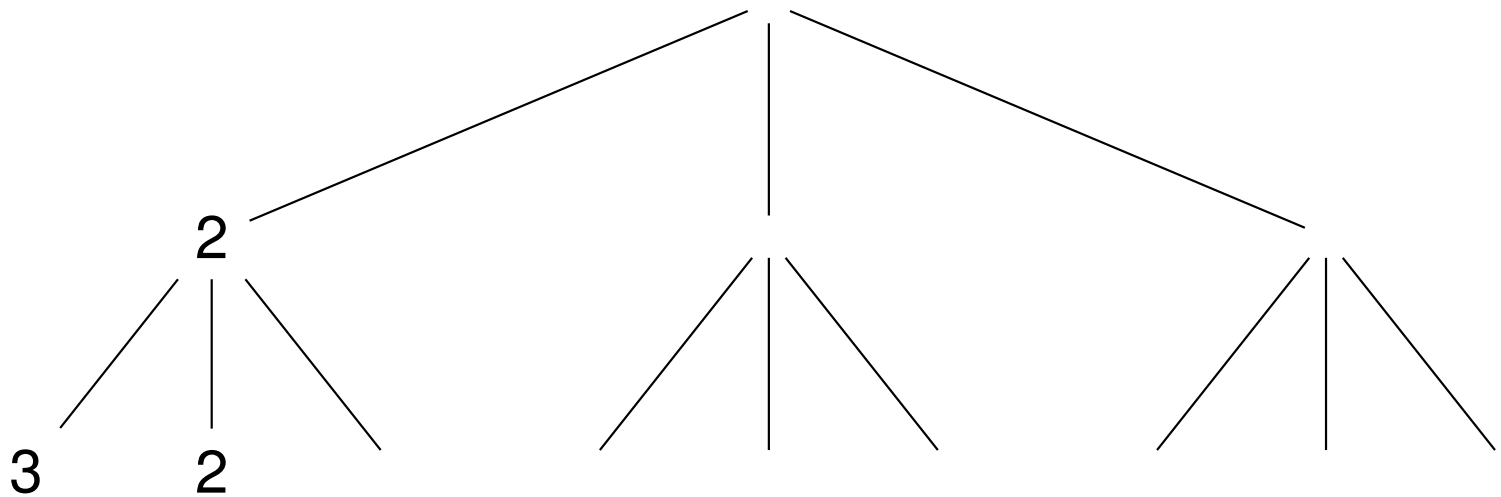
```
int minimax(int color,int depth)
{
  int g,x,y,v,ix;
  /* Test if there is a winning move */
  for (ix=0;ix<SIZEX;ix++) {
    x=indx[ix];y=first[x];
    if (y!=SIZEY) {v=eval(x,y,color);if (v!=0) return v;}
  }
  /* If no winning move and only one empty square => draw */
  if (depth==(SIZEX*SIZEY-1)) return 0;
  /* Optimized Minimax */
  if (color==WHITE) g=-32767; else g=32767;
  for (ix=0;ix<SIZEX;ix++) {
    x=indx[ix];y=first[x];
    if (y!=SIZEY) {
      tab[x][y]=color;first[x]++;
      v=minimax(-color,depth+1);
      first[x]--;tab[x][y]=EMPTY;
      if (color==WHITE) {g=max(g,v);if (g==1) return g;}
      else {g=min(g,v);if (g==-1) return g;}
    }
  }
  return g;
}
```

Alpha-béta

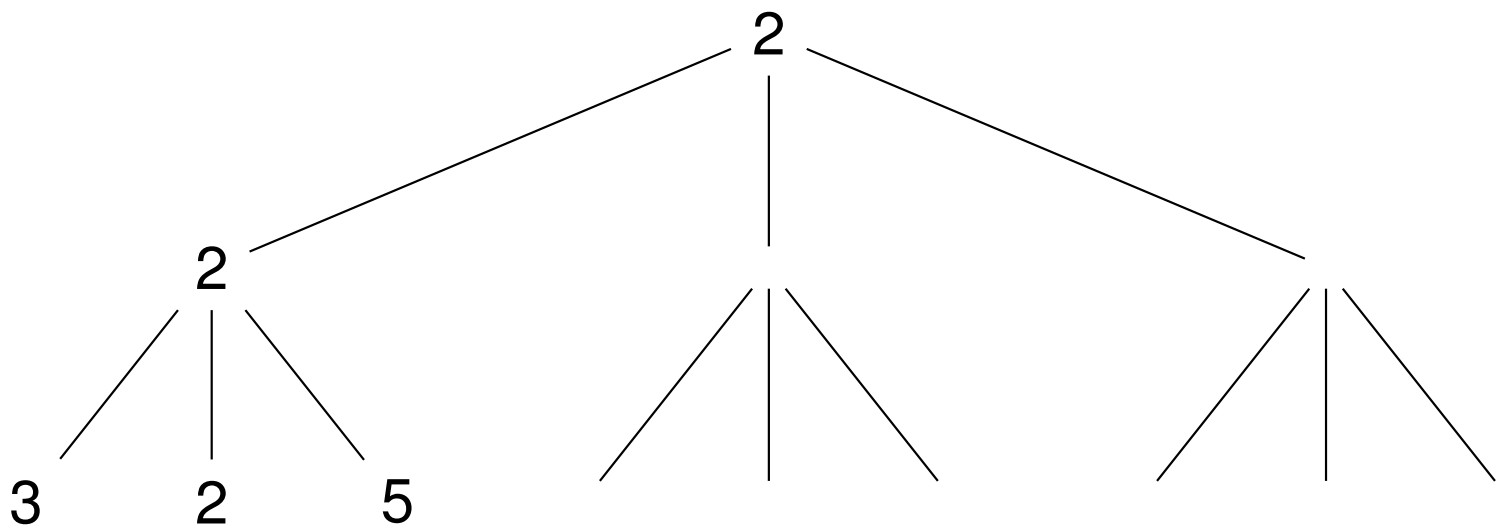
Alpha-béta



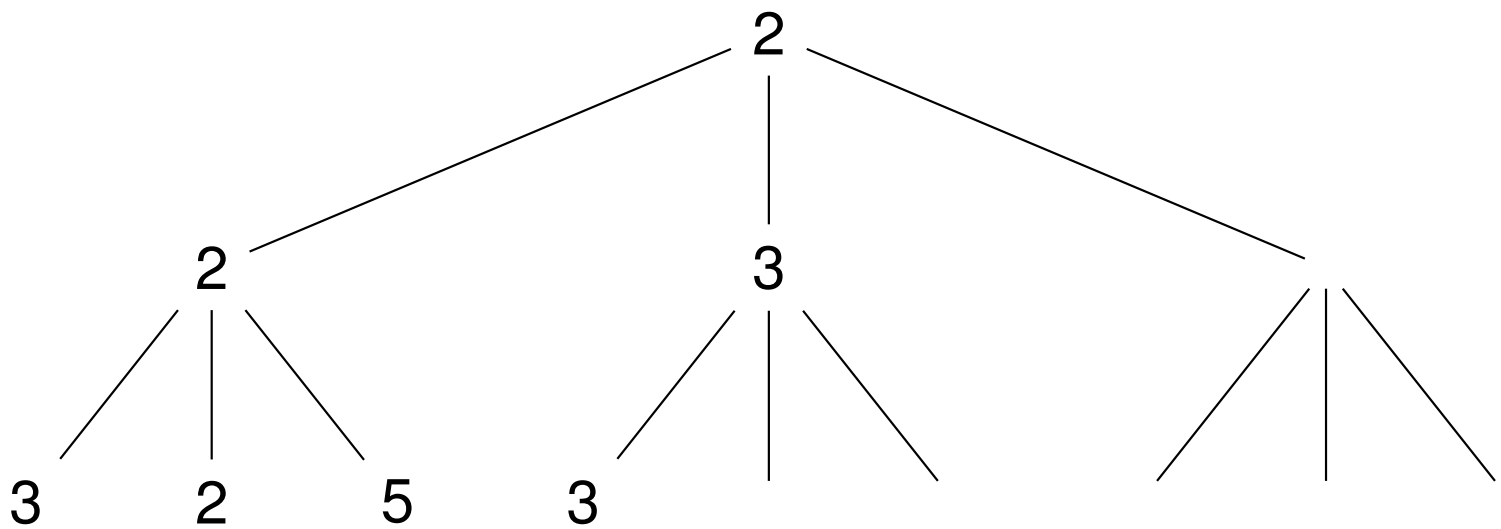
Alpha-béta



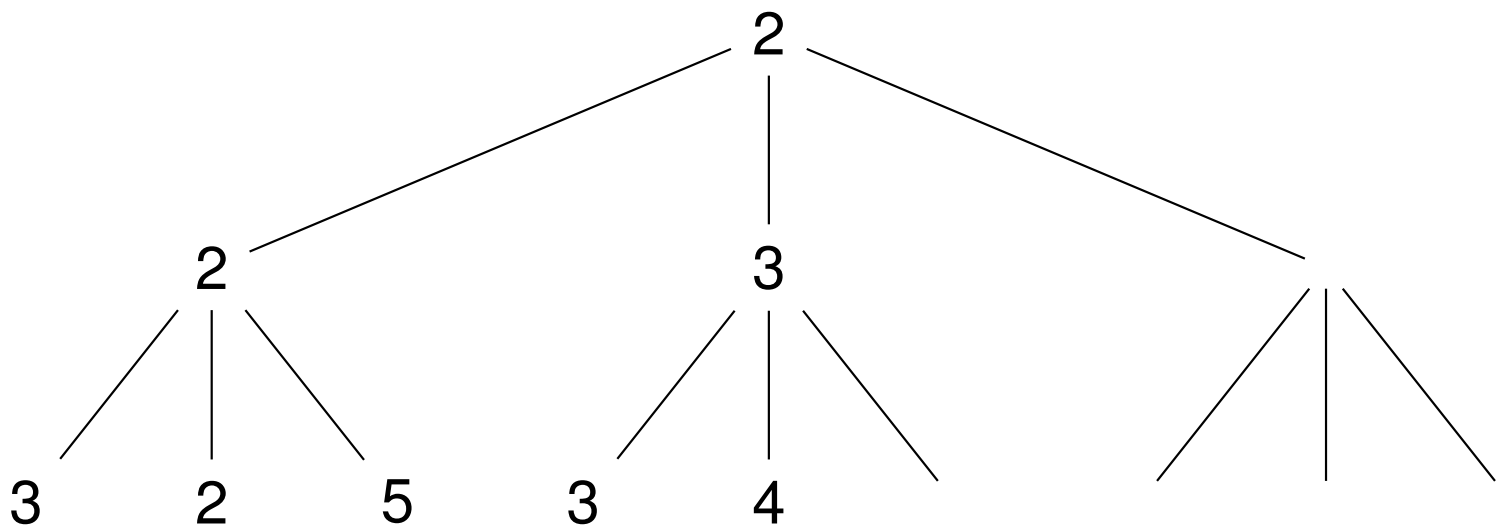
Alpha-béta



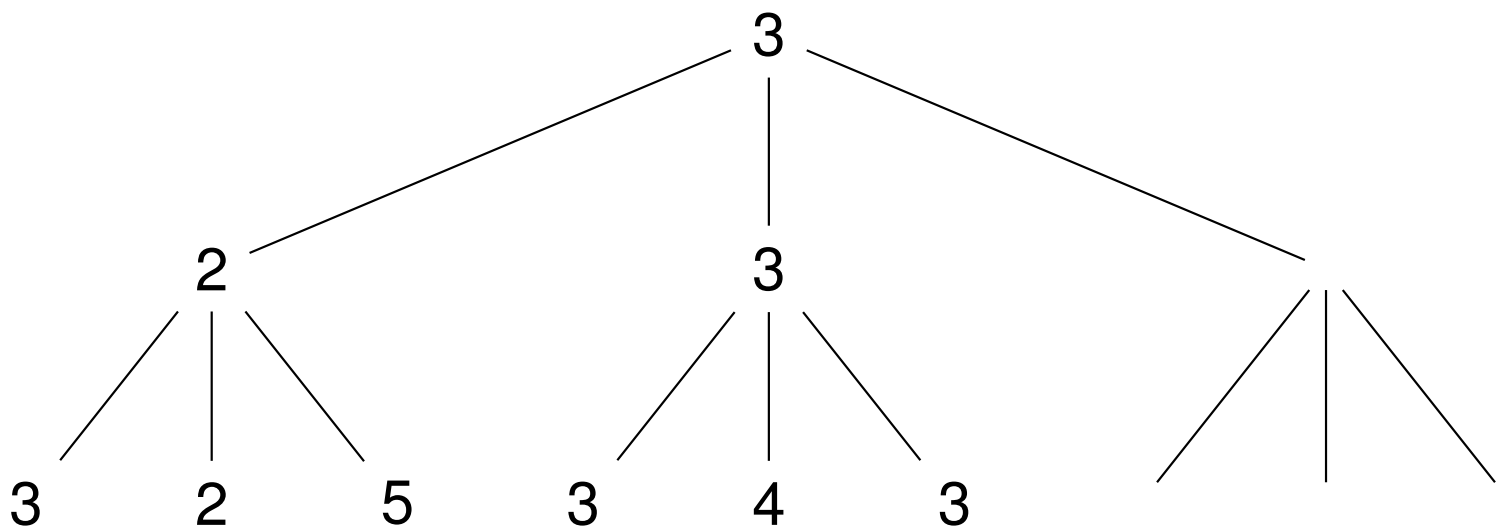
Alpha-béta



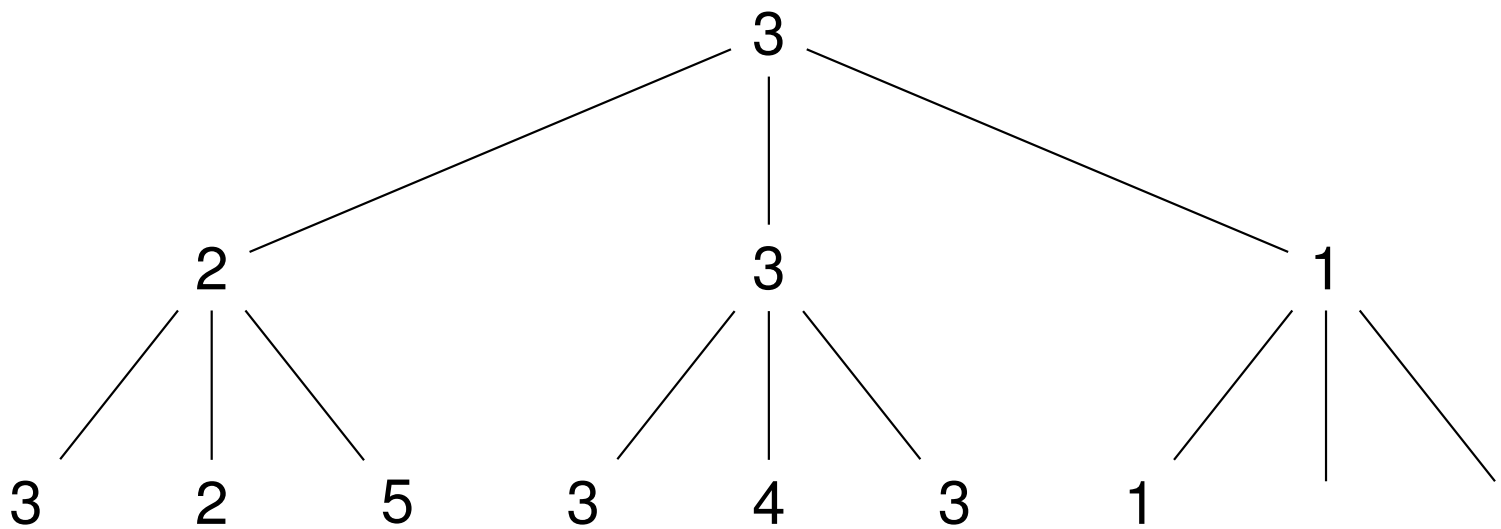
Alpha-béta



Alpha-béta



Alpha-béta



Alpha-Béta convention minimax

Algorithme 2 : Alpha-Béta

; ; ; n est le sommet dont on veut calculer la valeur;
; ; ; α et β valent initialement $-\infty$ et $+\infty$ respectivement;

```
Alpha-Béta(n, α, β);
si n est terminal alors retourner h(n);
si n est de type Max alors
    Soit j ← 1;
    Soit (f_1 ... f_k) les fils de n;
    tant que (α < β et j ≤ k) faire
        α ← max(α,Alpha-Béta(f_j, α, β));
        j ← (j + 1);
    retourner α;
sinon
    Soit j ← 1;
    Soit (f_1 ... f_k) les fils de n;
    tant que (α < β et j ≤ k) faire
        β ← min(β,Alpha-Béta(f_j, α, β));
        j ← (j + 1);
    retourner β;
```

Alpha-Béta convention négamax

Algorithme 3 : Alpha-Béta version négamax

; ; ; n est le sommet dont on veut calculer la valeur;
; ; ; α et β valent initialement $-\infty$ et $+\infty$ respectivement;

Alpha-Béta(n, α, β);
si *n est terminal* **alors retourner** $h(n)$;
Soit $j \leftarrow 1$;
Soit $(f_1 \ldots f_k)$ *les fils de n*;
tant que $(\alpha < \beta$ *et* $j \leq k)$ **faire**
    $\alpha \leftarrow \max(\alpha, -$Alpha-Béta$(f_j, -\beta, -\alpha))$;
    $j \leftarrow (j + 1)$;

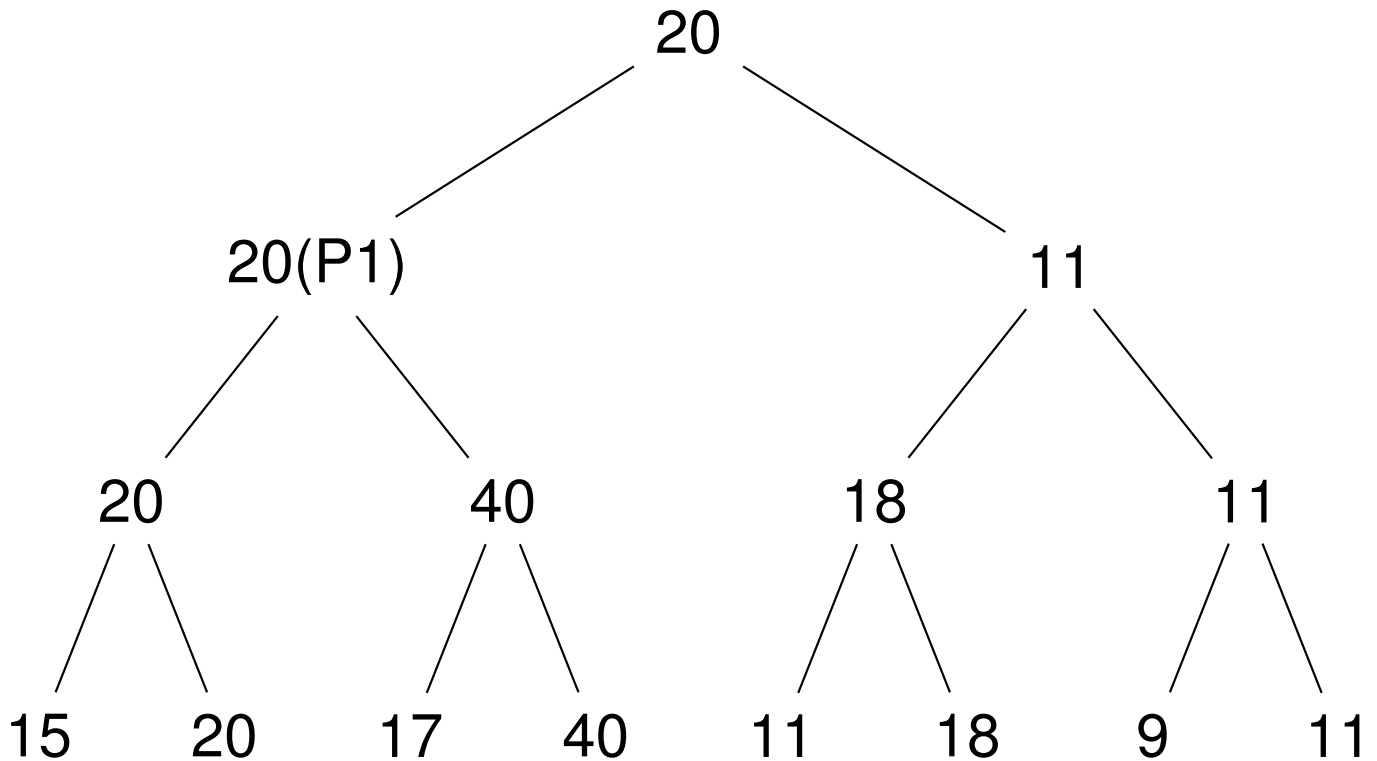
retourner α;

Puissance 4 alphabeta

```
int alpha_beta(int alpha,int beta,int color,int depth)
{
  int a,b,g,x,y,v;
  /* Test if there is a winning move */
  for (x=0;x<SIZEX;x++) {
    y=first[x];
    if (y!=SIZEY) {v=eval(x,y,color);if (v!=0) return v;}
  }
  /* If no winning move and only one empty square => draw */
  if (depth==(SIZEX*SIZEY-1)) return 0;
  /* Classical alpha-beta */
  a=alpha;b=beta;
  if (color==WHITE) g=-32767; else g=32767;
  for (x=0;x<SIZEX;x++) {
    y=first[x];
    if (y!=SIZEY) {
      tab[x][y]=color;first[x]++;
      if (color==WHITE) v=alpha_beta(a,beta,-color,depth+1);
      else v=alpha_beta(alpha,b,-color,depth+1);
      first[x]--;tab[x][y]=EMPTY;
      if (color==WHITE) {
        g=max(g,v);a=max(a,g);if (g>=beta) break; /* Beta cut */
      }
      else {
        g=min(g,v);b=min(b,g);if (g<=alpha) break; /* Alpha cut */
      }
    }
  }
  return g;
}
```

Transposition

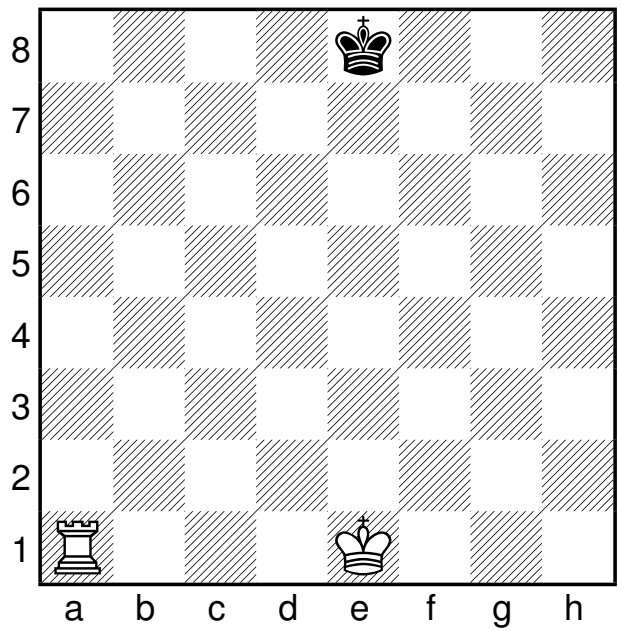
Principe



position	evaluation	coup	distance
P1	20	G	2

Pièce	couleur	position	valeur de hash-coding (générée aléatoirement)
Tour	blanc	a1	10101011110110110101010110101101
Tour	blanc	a2	01000101001011001110100111101010
...	...	...	...
Roi	blanc	e1	11010110101001001010001011010110
...	...	...	...
Roi	noir	e8	00100101010110110100100010101000
...	...	...	...

$black\_turn = 01101001010010101001101011001001$



	10101011110110110101010110101101
xor	11010110101001001010001011010110
xor	00100101010110110100100010101000
	01011000001001001011111111010011
and	00000000000000001111111111111111
	1011111111010011

```
hash[1011111111010011].sig=01011000001001001011111111010011;
hash[1011111111010011].val=...;
hash[1011111111010011].bmove=...;
```

Mise en oeuvre

Il faut stocker :

- La signature de la position avec suffisamment de bits pour que la probabilité que deux positions différentes aient la même signature soit “suffisamment” faible.
- L'évaluation de la position (ou les bornes inf et sup pour l'alpha-béta).
- Des éléments permettant de gérer la table en cas de conflits, c'est à dire lorsque deux positions différentes ont la même signature et doivent stocker leurs informations au même endroit dans le tableau :
  - la distance à la fin de la partie (si disponible)
  - la distance à la fin de l'évaluation
  - l'âge du dernier accès
  - ...
- Le meilleur coup trouvé dans le cas de l'Iterα-β que nous verrons plus tard.

Le “tuning”

Les tables de transposition doivent être assez finement “réglées” en ce qui concerne certains paramètres :

- la taille (en bits) de la signature : trop grande elle occupe trop de place, trop petite on risque des erreurs
- La taille de la table : trop grande elle ralentit le programme en déclenchant trop de défauts de cache, trop petite, on ne stocke pas suffisamment de valeurs
- La gestion des conflits : il faut savoir quelle position garder lorsque l’on a conflit

Alpha-Béta à mémoire

Algorithme 4 : Alpha-Béta à mémoire (profondeur fixe)

```
;;; n est le sommet dont on veut calculer la valeur;
Alpha-Béta(n, α, β);
si n est dans la table alors
    si n.bas = n.haut alors retourner n.bas;
    si n.bas >= β alors retourner n.bas;
    si n.haut <= α alors retourner n.haut;
    α ← max(α, n.bas);
    β ← min(β, n.haut);
si n est terminal alors retourner h(n);
si n est de type Max alors
    Soit g ← −∞;
    Soit j ← 1;
    Soit a ← α;
    Soit (f_1 ... f_k) les fils de n;
    tant que (g < β et j ≤ k) faire
        g ← max(g,Alpha-Béta(f_j, a, β));
        a ← max(a, g);
        j ← (j + 1);
si n est de type Min alors
    Soit g ← +∞;
    Soit j ← 1;
    Soit b ← β;
    Soit (f_1 ... f_k) les fils de n;
    tant que (g > α et j ≤ k) faire
        g ← min(g,Alpha-Béta(f_j, α, b));
        b ← min(b, g);
        j ← (j + 1);
si g <= α alors n.haut ← g;
si g >= β alors n.bas ← g;
si g > α et g < β alors
    n.bas ← g;
    n.haut ← g;
retourner g
```

Attention !!!

- L'algorithme précédent n'a de sens que pour une seule exécution à profondeur fixée ou totale, ce qui est le cas à présent puisque nous résolvons totalement le jeu. Nous verrons plus loin que dans le cadre de l'Iterα-β, on peut réutiliser les tables de hachage à des profondeurs différentes, mais dans ce cas la valeur de l'évaluation n'a de sens que si la distance à la feuille de l'arbre est la même !
- L'α-β est déjà en soit un algorithme assez "casse-gueule" à débugger. L'α-β à mémoire est beaucoup plus infernal, et peut causer des bugs sournois et très difficiles à trouver. Il est donc fortement conseillé de toujours avoir sous la main un α-β "standard" et de vérifier les résultats de l'α-β à mémoire avec l'α-β standard. Même si le nombre de noeuds évalués sera différent, l'évaluation de la position (toujours) et le meilleur coup (en général) doivent rester les mêmes à la même profondeur !

Iterα-β, MTD(f)

Nous avons jusqu'ici travaillé en supposant que le jeu était totalement soluble. Notre fonction d'évaluation ne nous fournit que 3 valeurs -1(perdu), 0(nul) et 1(gagné), et nous allons jusqu'au bout de l'arbre de résolution. Cependant, cette façon de faire ne fonctionne que pour des jeux extrèmement simples. En général, il est impossible de résoudre complètement le jeu, et l'on doit faire alors appel à des techniques un peu différentes.

- Il faut développer des fonctions d'évaluation *heuristiques*, c'est à dire qui donnent une évaluation imparfaite de la position permettant d'évaluer sa probabilité de victoire.
- Il faut introduire des techniques de contrôle du temps de jeu, un adversaire humain appréciant rarement de devoir attendre pendant des heures la réponse du programme.
- On peut utiliser des algorithmes de recherche légèrement différents permettant de mieux intégrer contrôle de temps et fonctions d'évaluation heuristiques.

Iter α-β à mémoire

Algorithme 5 : Alpha-Béta à mémoire (profondeur variable)

```
;;; n est le sommet dont on veut calculer la valeur;
;;; d est la distance à la feuille de l'arbre;
Alpha-Béta(n, α, β, d);
Soit old_best ← −1, curr_best ← −1;
si n est dans la table alors
    old_best ← n.bmove;
    si d = n.d alors
        si n.bas = n.haut alors retourner n.bas;
        si n.bas >= β alors retourner n.bas;
        si n.haut <= α alors retourner n.haut;
        α ← max(α, n.bas);
        β ← min(β, n.haut);
si d = 0 alors retourner h(n);
si n est de type Max alors
    Soit g ← −∞, j ← 1, a ← α;
    Soit (f_1 ... f_k) les fils de n;
    si old_best ≠ −1 alors
        Réordonner (f_old_best, f_1 ... f_k)
    tant que (g < β et j ≤ k) faire
        t ←Alpha-Béta(f_j, a, β, d − 1));
        si t > g alors g ← t, curr_best ← j;
        a ← max(a, g);
        j ← (j + 1);
si n est de type Min alors
    Soit g ← +∞, j ← 1, b ← β;
    Soit (f_1 ... f_k) les fils de n;
    si old_best ≠ −1 alors
        Réordonner (f_old_best, f_1 ... f_k)
    tant que (g > α et j ≤ k) faire
        t ←Alpha-Béta(f_j, α, b, d − 1));
        si t < g alors g ← t, curr_best ← j;
        b ← min(b, g);
        j ← (j + 1);
si curr_best ≠ −1 alors n.bmove ← curr_best;
si g <= α alors
    n.haut ← g;
    si n.d ≠ d alors n.bas ← −∞;
si g >= β alors
    n.bas ← g;
    si n.d ≠ d alors n.haut ← +∞;
si g > α et g < β alors
    n.bas ← g;
    n.haut ← g;
n.d ← d;
retourner g
```

Iterα-β à mémoire

- L'idée de l'Iterα-β est de lancer l'algorithme de façon répétitive pour $d = 1$, $d = 2$, etc... En sauvegardant le meilleur coup pour chaque position, on ordonne efficacement la recherche. Il est souvent plus efficace qu'un α-β standard.
- L'iterα-β permet un bien meilleur contrôle du temps, puisqu'il est possible de récupérer le résultat de l'itération précédente si l'on doit interrompre l'itération en cours.
- Les informations gagnées lors du *coup* précédent peuvent servir lors du coup suivant.
- Attention !!! Ce pseudo code ne gère pas :
  - l'initialisation de la table
  - les conflits entre des positions ayant la même valeur de hash-coding mais des signatures différentes.

  Donc ne l'implantez pas bêtement !!!

En pratique

- La création du tableau peut se faire en utilisant la primitive C *calloc*. Dans ce cas là, tous les éléments de la table sont initialisés automatiquement à 0.
- La gestion des conflits se fait généralement en conservant les positions les plus proches de la racine de l'arbre (le plus grand d) pour une position initiale identique.
- Attention : il faut donc parfois aussi stocker une notion d'état de la partie. Par exemple, au puissance 4, on peut stocker le numéro du coup courant à la *racine* de l'arbre pour la recherche en cours. Cela permet d'arbitrer entre deux positions qui ont des numéros de coup différents : il faut conserver en priorité celle dont le numéro de coup est le plus élevé, car les autres ont plus de chance d'être caduques, les lignes de jeu ayant divergé.

α-β à fenêtre

- Un α-β à fenêtre réduit dès l'appel l'amplitude $[-\infty, +\infty]$ à $[min, max]$, où *min* et *max* sont des valeurs estimées.
- *min* et *max* peuvent être estimés pour l'appel à la profondeur $d + 1$ à partir de la valeur retournée à la profondeur d.
- L'algorithme fonctionne bien si la fonction d'évaluation est stable, c'est à dire qu'elle évolue peu lorsque le trait alterne entre les deux joueurs.
- Il n'est pas simple de construire des fonctions d'évaluation stable, car jouer confère un avantage.
- *MTD*(f) est une version "extrême" de cette technique, où la fenêtre est réduite à $[f - 1, f]$ où f est la valeur estimée.
- Dans le cas de l'Iterα-β à mémoire, les appels faits pour une première valeur de f donnent des informations sur les bornes haute et basse de chaque noeud qui restent stockées dans les tables de transposition et sont utiles pour les appels suivants.

α-β à fenêtre

Algorithme 6 : AlphaBetaWin(*min*, *max*)

; ; ; *root* est la racine de l'arbre;

AlphaBetaWin(*root*, *min*, *max*);
Soit $\alpha \leftarrow$ *min*;
Soit $\beta \leftarrow$ *max*;
g $\leftarrow$`Alpha-Béta(`*root*, α, β`)`;
si $g > \alpha$ *et* $g < \beta$ **alors retourner** *g*;
sinon si $g \leq \alpha$ **alors retourner**
`Alpha-Béta(`*root*, $-\infty$, α`)`;
sinon retourner `Alpha-Béta(`*root*, β, $+\infty$`)`;

MTD(f)

Algorithme 7 : MTD(f)

; ; ; *root* est la racine de l'arbre;

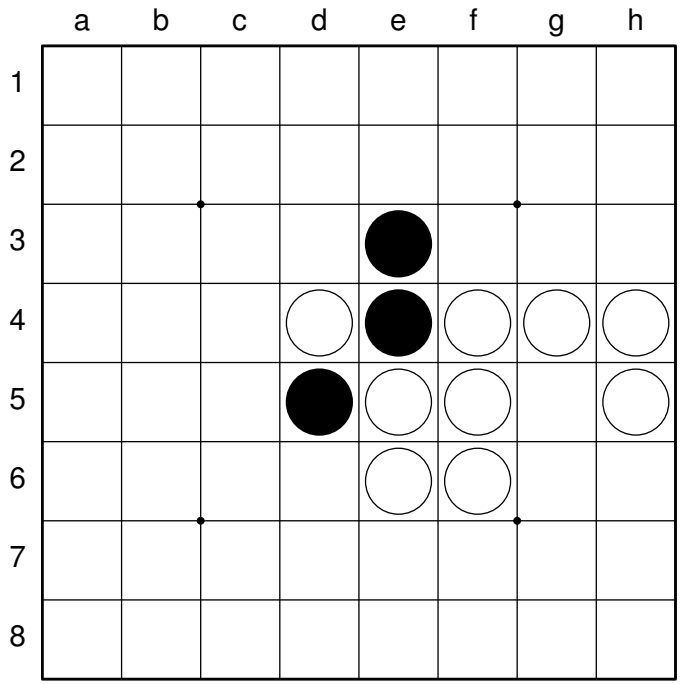
MTD(*root*, f);
Soit $g \leftarrow f$;
Soit $h \leftarrow +\infty$;
Soit $b \leftarrow -\infty$;
tant que $h > b$ **faire**
    si $g = b$ **alors** $\beta \leftarrow g + 1$;
    sinon $\beta \leftarrow g$;
    $g \leftarrow$`Alpha-Béta(`*root*$, \beta - 1, \beta$`)`;
    si $g < \beta$ **alors** $h \leftarrow g$;
    sinon $b \leftarrow g$;
retourner g

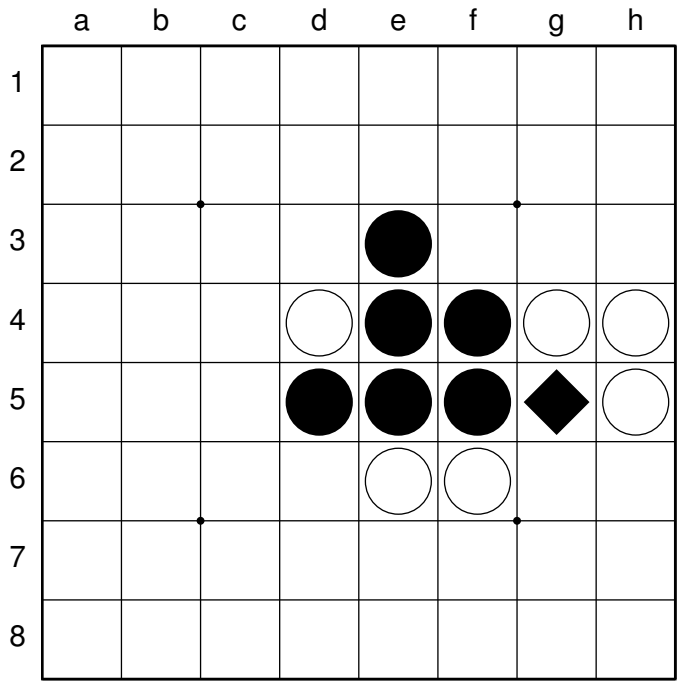
Evaluation, apprentissage

Fonctions d'évaluation

- La fonction d'évaluation représente la "connaissance" que l'on implante dans le programme.
- Cette connaissance peut être codée "en dur", ou le résultat d'un processus d'apprentissage, soit par renforcement (par exemple Logistello dans les années 90), soit plus récemment par réseaux de neurones (le célèbre AlphaGo).
- Il y a un trade-off entre la complexité de la fonction d'évaluation (et donc sa qualité) et la profondeur d'évaluation.

Principe du jeu





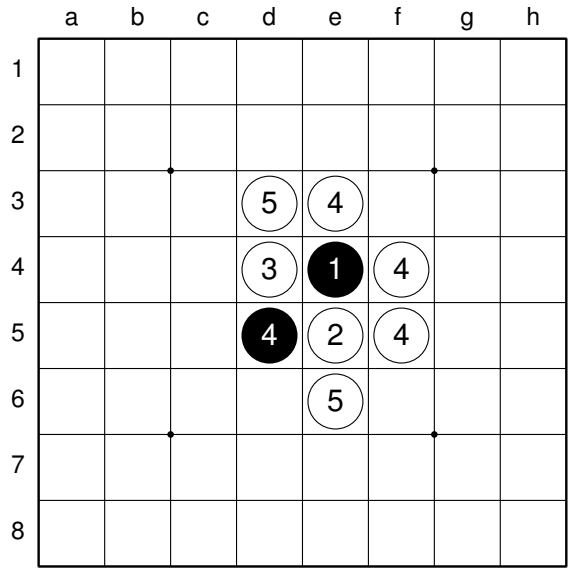
Valuation statique (pauvre)

On associe à chaque case une valeur. Les angles sont les plus importants car imprenables.

	a	b	c	d	e	f	g	h
1	500	-150	30	10	10	30	-150	500
2	-150	-250	0	0	0	0	-250	-150
3	30	0	1	2	2	1	0	30
4	10	0	2	16	16	2	0	10
5	10	0	2	16	16	2	0	10
6	30	0	1	2	2	1	0	30
7	-150	-250	0	0	0	0	-250	-150
8	500	-150	30	10	10	30	-150	500

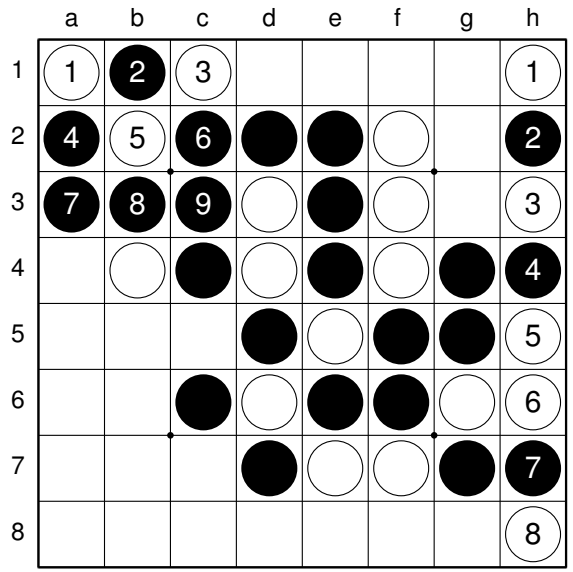
Valuation “dynamique”

- Centraliser ses pions
- Maximiser le nombre de coups jouables
- Plus simplement, minimiser le nombre de cases vides à côté de ses propres pions
- Ici noir a 9 coups jouables, blanc en a 4. Noir a 5 cases voisines vides, blanc en a 27. Même si noir a moins de pions que blanc, la position de noir est meilleure.



Apprentissage par renforcement

- Apprentissage par renforcement de type MENACE
- On observe deux types de patterns :
  - Les cases 3x3 formant chaque coin
  - Les 8 cases de chaque bord
- On utilise deux bits par case (00 vide, 01 blanc, 10 noir)+1 bit pour coder le joueur au trait
- Il faut 19 bits pour les coins, 17 bits pour les bords
- Le ratio $n_{gagnees}/n_{total}$ donne une idée de la qualité du pattern



Evaluation élémentaire

- Valeur des pièces :
  - Dame : 9 points
  - Tour : 5 points
  - Fou : 3 points
  - Cavalier : 3 points
  - Pion : 1 point
- Mettre le roi en sécurité
- Conserver la paire de fous
- Dominer le centre
- Garder un maximum de mobilité
- Centraliser les cavaliers
- Placer les fous sur les diagonales ouvertes
- Doubler les tours sur les colonnes ouvertes
- Conserver une bonne structure de pions

Autres phases

Les ouvertures

- Utilisation de bibliothèques d'ouverture, par exemple aux échecs
- Utilisation de bibliothèques d'anti-ouvertures (coups à éviter), par exemple aux checkers
- Les bibliothèques peuvent être préparées par de forts joueurs (les échecs), ou apprises par le programme
- La force actuelle des programmes limite de plus en plus l'utilité des bibliothèques d'ouverture

Les finales

- Pour les jeux à durée connue (comme Othello) :
  - une recherche exhaustive avec un α-*beta* à fenêtre réduite $[-1, 1]$ (c.a.d D/N/V) est lancée très en amont (une vingtaine de coups à Othello)
  - une recherche exhaustive complète est lancée un ou deux coups plus tard
- Pour certains jeux comme les échecs, on construit des bases de données de fin de partie par analyse rétrograde.

Analyse rétrograde aux échecs

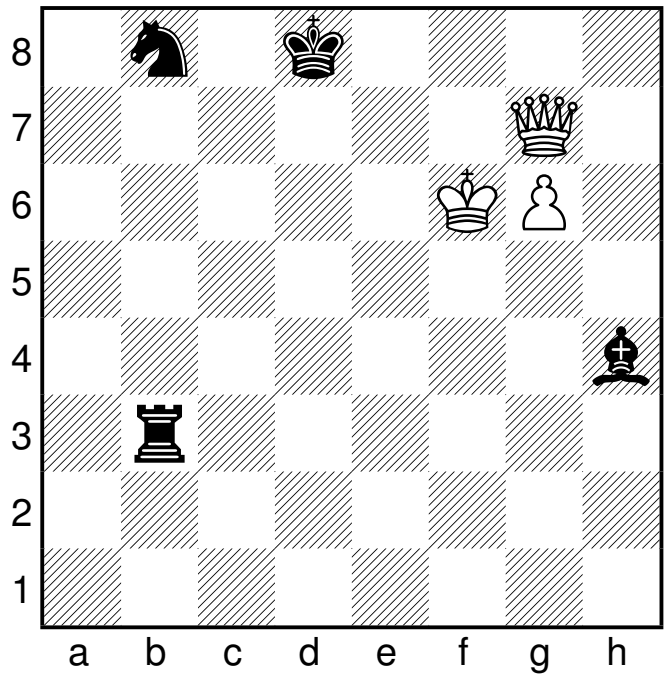
Principe général :

- on recherche toutes les positions où les noirs sont mat avec un ensemble de pièces données
- on détermine les positions de mat en un coup : ce sont les positions qui permettent au blanc d'atteindre en un coup une position de mat.
- on détermine récursivement les positions de mat en $n+1$ coups à partir des positions de mat en n coups.

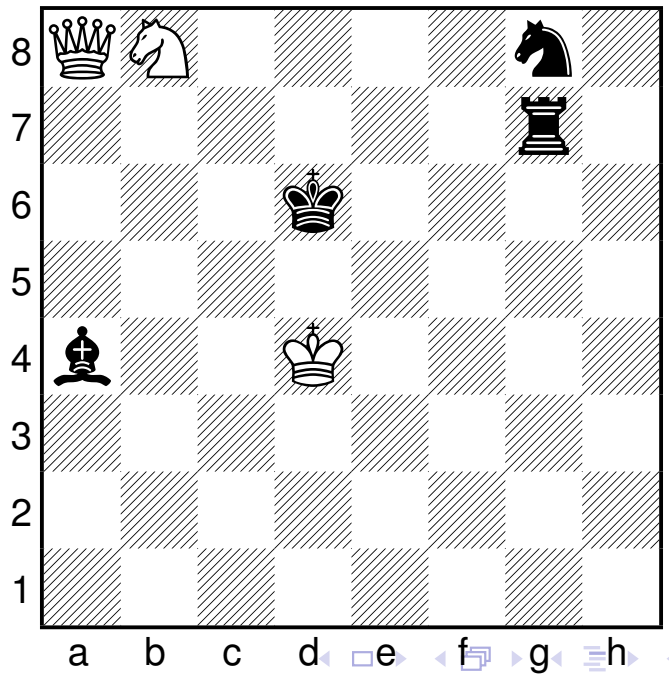
On a ainsi construit successivement les finales pour toutes les positions à 3, puis 4, 5, 6 et 7 pièces. Ces bases de données nécessitent, suivant les formats de stockage, autour de 1GB (5 pièces), 150GB (6 pièces) et 17TB (7 pièces).

Plus longues finales connues aux échecs

Ci-dessous, la plus longue finale connue à l'heure actuelle, où les blancs jouent et gagnent en 549 coups.



Dans la position ci-dessous, les noirs jouent et perdent en 545 coups, mais il n'y a aucune prise avant le 523ème coup.



Play chess with God (Ken Thompson)

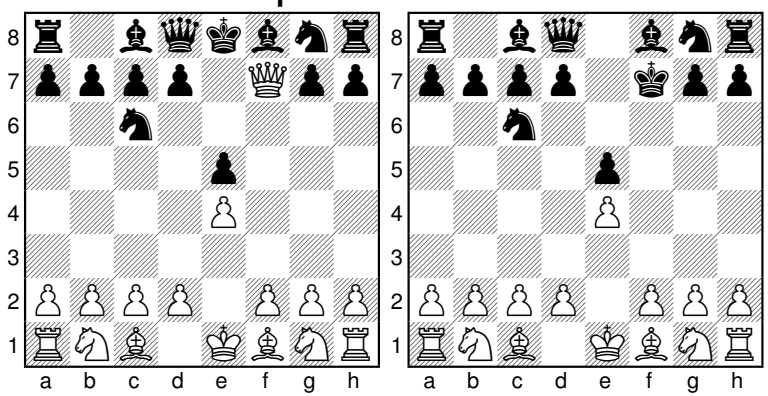
"A grandmaster wouldn't be better at these endgames than someone who had learned chess yesterday. It's a sort of chess that has nothing to do with chess, a chess that we could never have imagined without computers. The Stiller moves are awesome, almost scary, because you know they are the truth, God's Algorithm – it's like being revealed the Meaning of Life, but you don't understand one word." Tim Krabbé

Améliorations

Recherche de quiescence

- Il faut éviter de terminer la recherche sur une feuille “instable”.
- On fait une recherche spécifique à partir de chaque feuille où on restreint les coups aux seules prises ou échecs par exemple.
- Permet d’éviter partiellement l’effet horizon
- Indispensable sur certains jeux (les échecs), moins utile sur d’autres (Othello)

Ici , après **1 e4 e5 2 ♕f3 ♘c6 3 ♕×f7+**, la reine blanche a pris un pion et le roi noir est en échec pourtant la situation est clairement mauvaise après . . . ♔×f7.



Alpha-Béta négamax+quiescence

Algorithme 8 : Alpha-Béta négamax+quiescence

;;; n est le sommet dont on veut calculer la valeur;
;;; α et β valent initialement $-\infty$ et $+\infty$ respectivement;

`Alpha-Béta(`n, α, β, d`);`
si $d = 0$ **alors retourner** `quiescence(`n, α, β`);`
Soit $j \leftarrow 1$;
Soit $(f_1 \ldots f_k)$ *les fils de n*;
tant que *(*$\alpha < \beta$ *et* $j \leq k$*)* **faire**
    $\alpha \leftarrow \max(\alpha, -$`Alpha-Béta(`$f_j, -\beta, -\alpha, d-1$`));`
    $j \leftarrow (j+1)$;

retourner α;

`quiescence(`n, α, β`);`
$\alpha \leftarrow \max(\alpha, h(n))$;
Soit $j \leftarrow 1$;
Soit $(f_1 \ldots f_l)$ *les fils de n qui sont des prises ou des échecs ou des sorties d'échecs*;
tant que *(*$\alpha < \beta$ *et* $j \leq l$*)* **faire**
    $\alpha \leftarrow \max(\alpha, -$`quiescence(`$f_j, -\beta, -\alpha$`));`
    $j \leftarrow (j+1)$;

retourner α;

Coups meurtriers

- Si un coup se révèle décisif dans une position, il y a de fortes chances qu'il soit aussi décisif dans d'autres positions.
- Il faut donc essayer de le traiter parmi les premiers dans l'α-β lorsqu'il est possible de le jouer.
- On conserve ces coups dans des tables spécifiques.
- Heuristique efficace sur certains jeux, inutile sur d'autres.

Elagage de futilité

- Au lieu d'élaguer seulement les positions strictement inférieures, on élague aussi les positions inférieures ou supérieures de "quelques" pourcents
- Accélère la recherche au détriment de l'optimalité
- Plus ou moins efficace (ou dangereux) suivant le jeu concerné

Extension sélective

- Dans un alpha-béta normal, la profondeur est fixé à l'avance
- Dans un système d'extension sélective, la profondeur dépend de l'intérêt de la branche.
- Permet d'explorer plus en profondeur les branches contenant des coups complexes
- A condition de savoir les trouver. . .

Heuristique du coup nul (Null Move Heuristic)

- Si on doit évaluer un nœud en profondeur n, on essaie d'abord de l'évaluer en profondeur $n - 2$ en supposant que l'adversaire passe son tour
- Si un avantage clair n'est pas acquis, le nœud est élagué
- Stratégie apparemment risqué mais extrêmement efficace sur certains jeux
- A éviter absolument dans les positions de Zugzwang

Alpha-Béta négamax avec NMH

Algorithme 9 : Alpha-Béta négamax avec NMH

;;; n est le sommet dont on veut calculer la valeur;
;;; α et β valent initialement $-\infty$ et $+\infty$ respectivement;
;;; R a une valeur comprise entre 1 et 3 suivant l'agressivité choisie

Alpha-Béta(n, α, β, d, *joueur*, *null_ok*) ;
si $d \leq 0$ **alors retourner** $h(n)$;
si *Pas_en_échec(n) et null_ok* **alors**
    $g \leftarrow -$Alpha-Béta(n, $-\beta$, $-\beta+1$, $d-R-1$, *joueur*, *FALSE*) ;
    si $g \geq \beta$ **alors retourner** g;

Soit $j \leftarrow 1$;
Soit $(f_1 \ldots f_k)$ *les fils de n*;
tant que *(*$\alpha < \beta$ *et* $j \leq k$*)* **faire**
    $\alpha \leftarrow \max(\alpha, -$Alpha-Béta($f_j$, $-\beta$, $-\alpha$, $d-1$, $-$*joueur*, *TRUE*)$)$;
    $j \leftarrow (j+1)$;

retourner α;