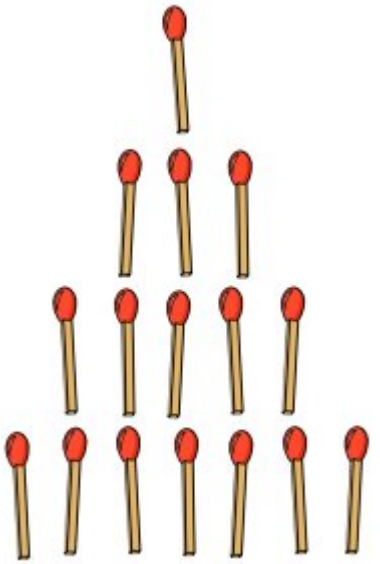
Intelligence Artificielle et Jeux

Tristan Cazenave
LAMSADE
Université Paris-Dauphine

Tristan.Cazenave@dauphine.fr

Le jeu de Nim



Le jeu de Nim

- Pour évaluer une position on effectue la somme binaire :

I	0 0 1
III	0 1 1
IIII	1 0 0
IIIIIII	1 1 1
	0 0 1

Le jeu de Nim

- Une position perdante pour celui qui va jouer a une somme binaire de 0.
- Pour choisir un coup on choisit le coup qui amène à une somme binaire de 0.
- Ecrire un programme de Nim imbattable.

Le jeu de Nim

```
#include <iostream>
using namespace std;
int tas [4] = {7, 5, 3, 1};
int main () {
  int t, nombre;
  while (true) {
    cout << tas [0] << " " << tas [1] << " " <<
      tas [2] << " " << tas [3] << endl;
    if (tas [0] + tas [1] + tas [2] + tas [3] == 0) {
      cout << "J'ai gagne !" << endl;
      break;
    }
```

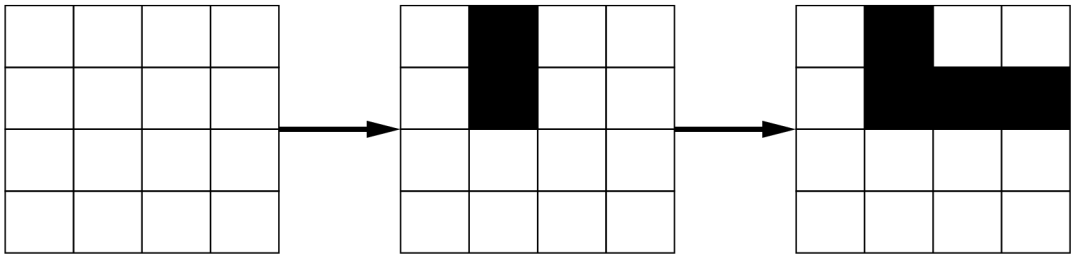
Le jeu de Nim

```
do {
  cout << "Donnez le tas (entre 0 et 3): ";
  cin >> t;
  cout << "Donnez le nombre : ";
  cin >> nombre;
} while ((t < 0) || (t > 3) || (tas [t] < nombre) ||
            (nombre < 1));
tas [t] -= nombre;
cout << tas [0] << " " << tas [1] << " " <<
            tas [2] << " " << tas [3] << endl;
```

```
    for (int i = 0; i < 4; i++)
      for (int j = 1; j <= tas [i]; j++) {
        tas [i] -= j;
        if ((tas [0] ^ tas [1] ^ tas [2] ^ tas [3]) == 0) {
          t = i;
          nombre = j;
        }
        tas [i] += j;
      }
    cout << "Je prend " << nombre <<
                    " dans le tas " << t << endl;
    tas [t] -= nombre;
  }
  return 0;
}
```

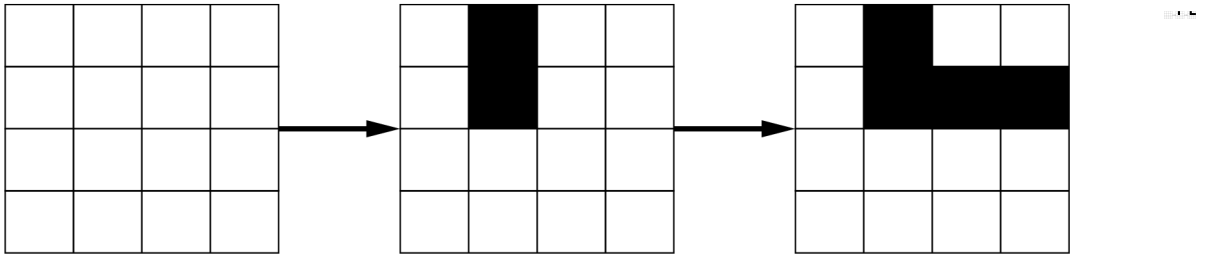
Domineering

- On pose des dominos sur un échiquier :



- Résoudre Domineering 3x3 avec un crayon et un papier.

Domineering



- Ecrire les classes Board et Move pour Domineering 8x8.
- Ecrire la liste des coups possibles.
- Evaluation = coups possibles – coups possibles adversaire
- Ecrire un algorithme qui choisit le coup qui maximise la fonction d'évaluation pour Domineering 8x8.

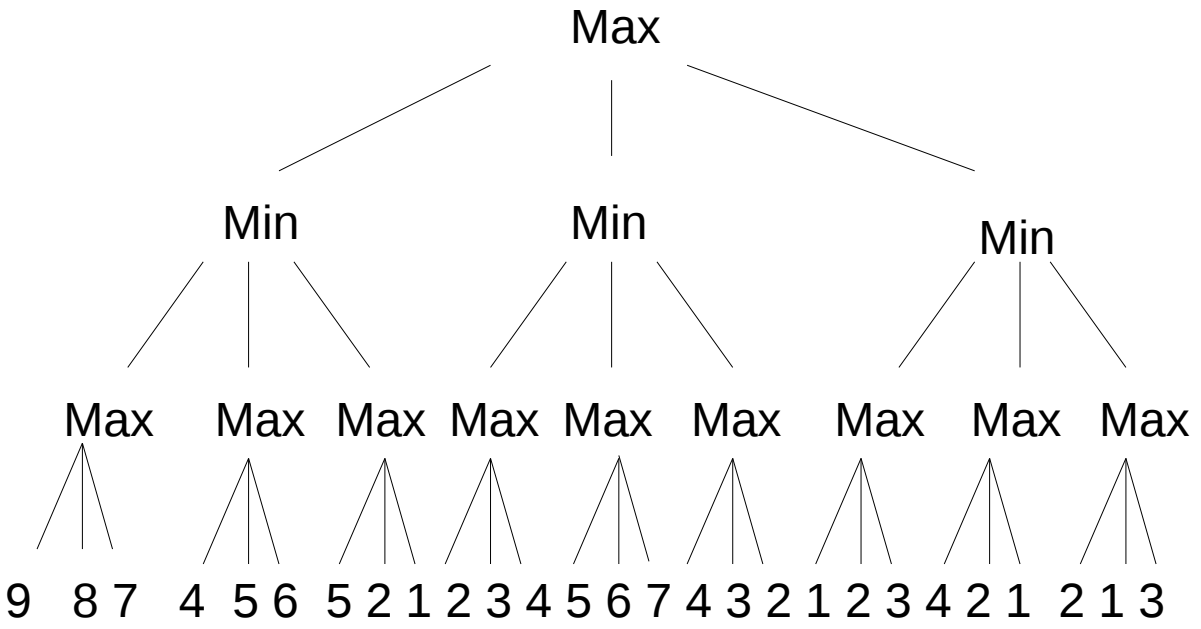
Minimax

- Jeux à somme nulle :
- Le joueur à la racine est en général max.
- Les gains sont ceux de max.
- Il y a une alternance entre min et max.
- On remonte les valeurs à la racine.
- L'algorithme pour résoudre le jeu est le Minimax.

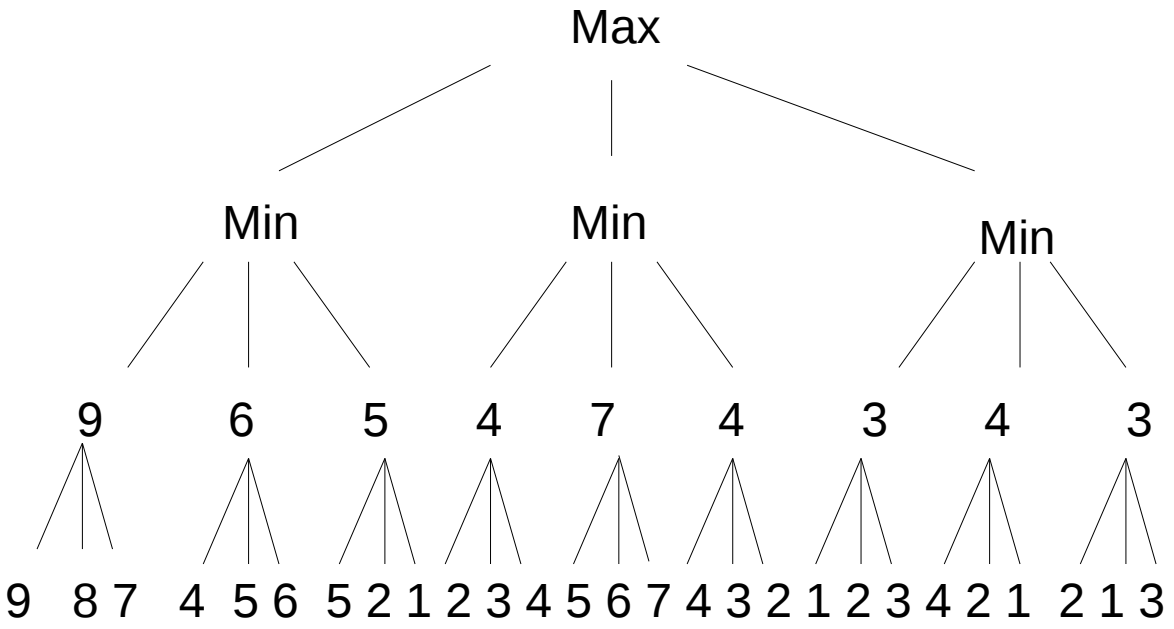
Minimax

- Fonction d'évaluation :
- Une fonction d'évaluation donne des valeurs élevées quand la position est favorable au joueur max et basses quand elle est défavorable.
- La fonction d'évaluation est appelée aux feuilles de l'arbre pour les évaluer.

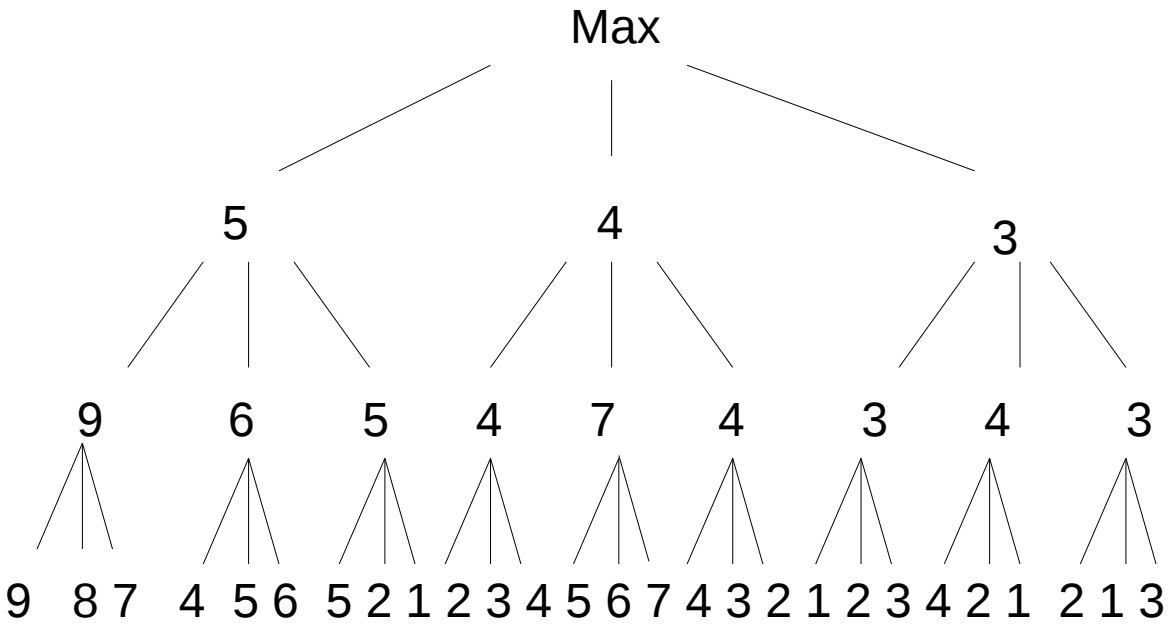
Minimax


Max
Min
Min
Min
Max
Max
Max
Max
Max
Max
Max
Max
Max
9
8
7
4
5
6
5
2
1
2
3
4
5
6
7
4
3
2
1
2
3
4
2
1
2
1
3

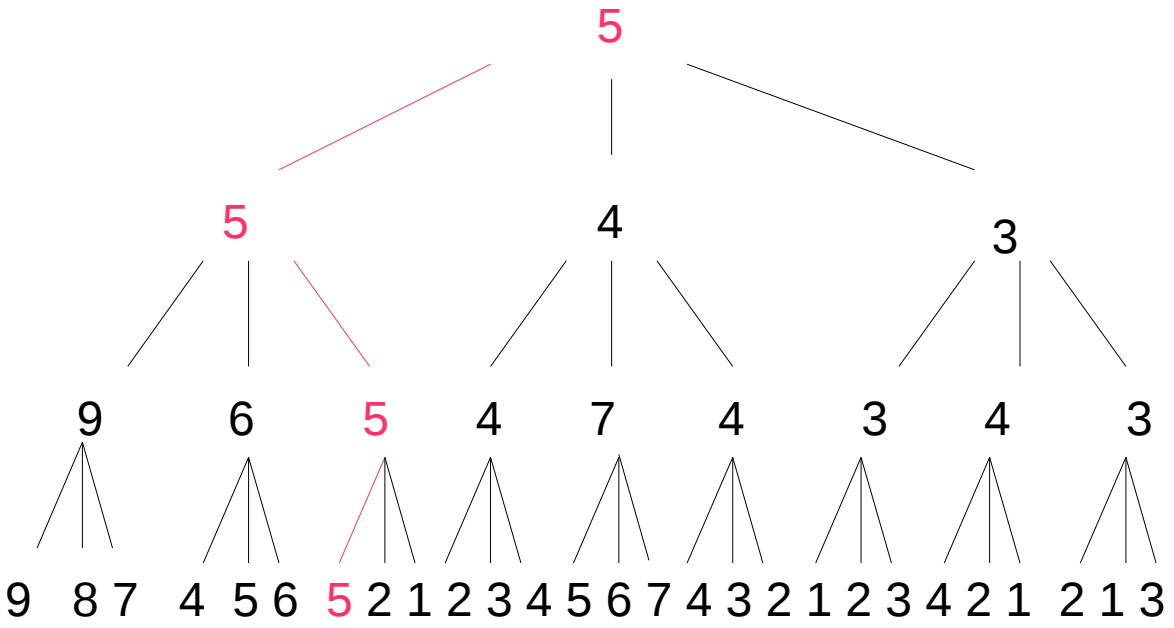
Minimax



Minimax



Minimax



Minimax

- Ecrire l'algorithme Minimax.
- On suppose qu'on dispose d'une fonction d'évaluation qui évalue un damier.
- On dispose aussi de fonctions qui donnent la liste des coups possibles, jouent et déjouent des coups sur le damier.
- L'algorithme recherche toutes les variations possibles jusqu'à une profondeur fixée et renvoie la valeur minimax de l'état.

Minimax

```
max (depth)
  if depth == 0
    return evaluation (maxPlayer)
  eval = -infinity
  for m in possible moves
      play (m)
      e = min (depth – 1)
      undo (m)
      if e > eval
        eval = e
  return eval
```

Minimax

```
min (depth)
  if depth == 0
    return evaluation (maxPlayer)
  eval = infinity
  for m in possible moves
      play (m)
      e = max (depth – 1)
      undo (m)
      if e < eval
        eval = e
  return eval
```

Negamax

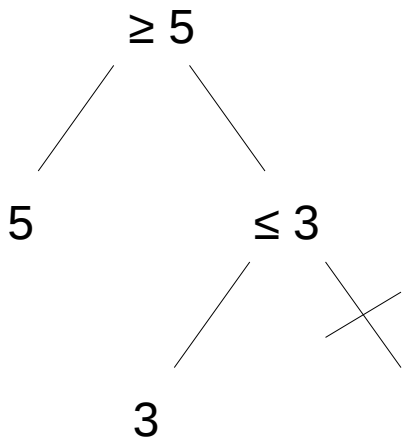
- L'algorithme Negamax effectue le même calcul que le Minimax.
- Il n'utilise qu'une seule fonction récursive au lieu de deux.
- Pour cela il change le signe de l'évaluation à chaque niveau et maximise toujours.
- Ecrire l'algorithme Negamax.

Negamax

```
negamax (depth)
  if depth == 0
    return evaluation (currentPlayer)
  eval = -infinity
  for m in possible moves of currentPlayer
      play (m)
      e = -negamax (depth – 1)
      undo (m)
      if e > eval
        eval = e
  return eval
```

Alpha-Beta

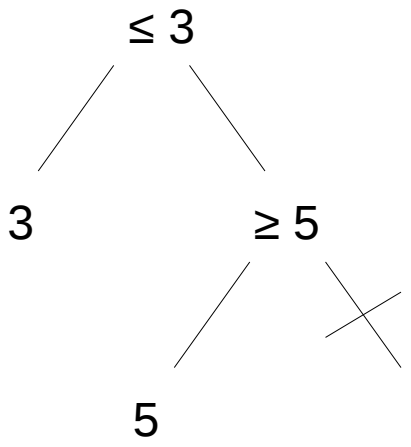
- Alpha-Beta :
- On parcourt l'arbre de gauche à droite.
- Coupe aux nœuds Min :



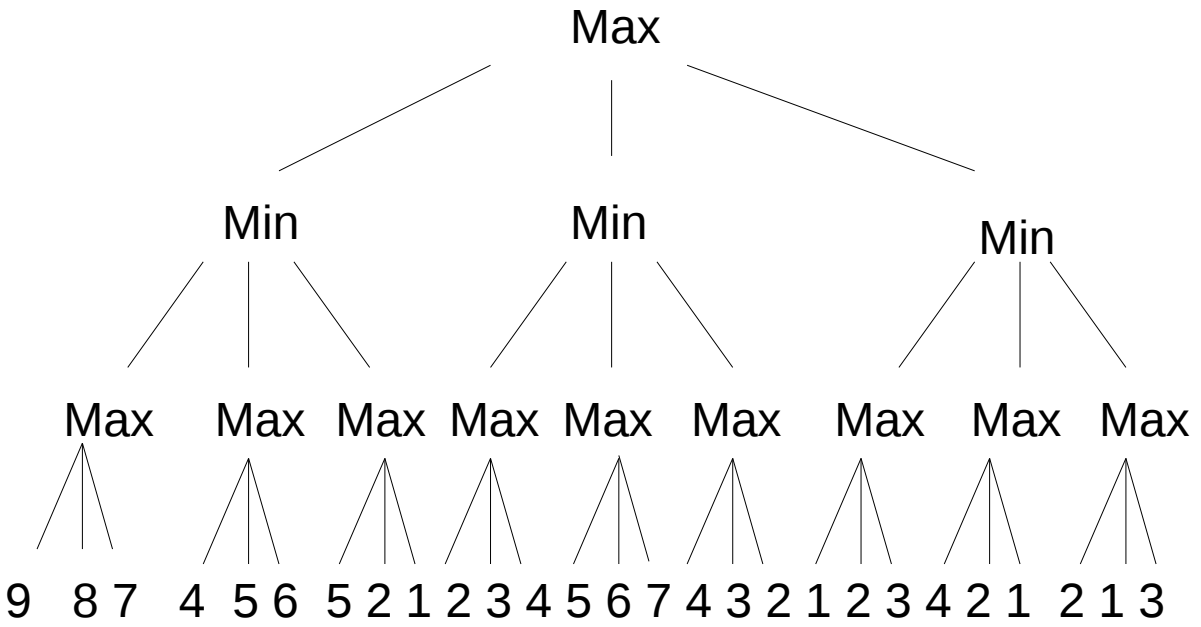
- Trouver un exemple de coupe aux nœuds Max.

Alpha-Beta

- Coupe aux nœuds Max :

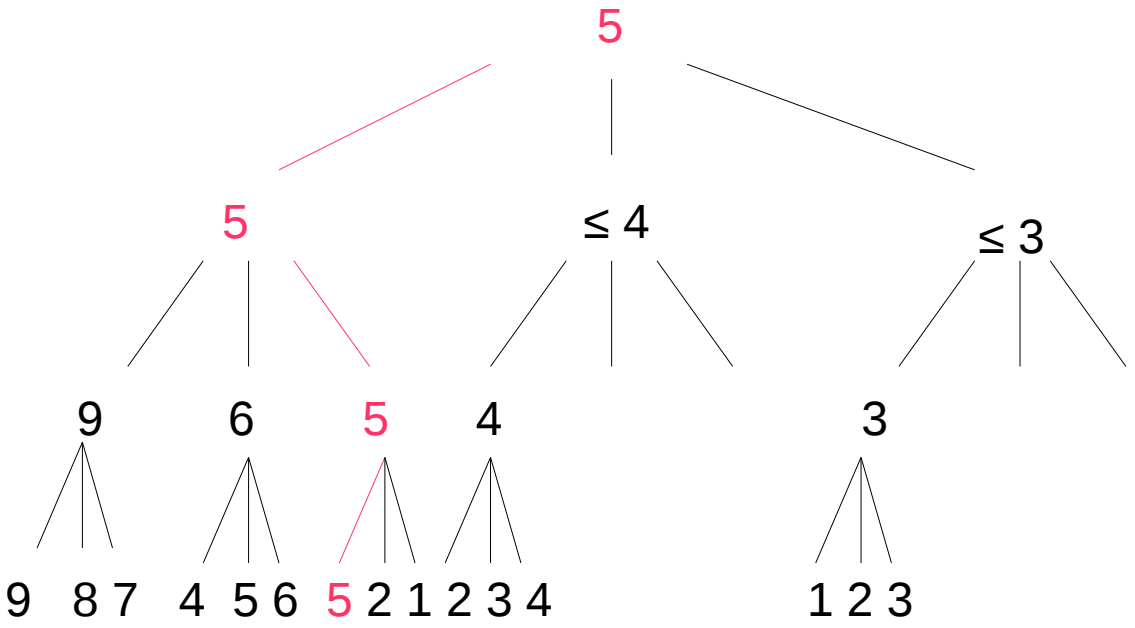


Alpha-Beta



- Effectuer les coupes Alpha-Beta sur cet arbre.

Alpha-Beta



Alpha-Beta

- Pour maximiser le nombre de coupes il faut bien ordonner les coups.
- On utilise de nombreuses heuristiques dans les programmes de jeu pour ordonner les coups.
- Les tables de transposition associées à l'approfondissement itératif.
- Les coups qui tuent.
- L'heuristique de l'historique.
- Grâce à ces heuristiques les programmes d'Echecs et de Dames développent des arbres proches de l'optimal (2 * racine(nombre de noeuds)).

Alpha-Beta

- Ecrire le pseudo code d'un algorithme Alpha-Beta.
- On passe deux entiers alpha et beta en paramètres qui sont les bornes min et max que peuvent prendre les évaluations.
- On appelle l'intervalle entre alpha et beta la fenêtre de l'Alpha-Beta.

Alpha-Beta

```
max (depth, alpha, beta)
  if depth == 0
    return evaluation (maxPlayer)
  for m in possible moves
      play (m)
      e = min (depth – 1, alpha, beta)
      undo (m)
      if e > alpha
        alpha = e
        if alpha ≥ beta
          return beta
  return alpha
```

Alpha-Beta

```
min (depth, alpha, beta)
  if depth == 0
    return evaluation (maxPlayer)
  for m in possible moves
      play (m)
      e = max (depth – 1, alpha, beta)
      undo (m)
      if e < beta
        beta = e
        if alpha ≥ beta
          return alpha
  return beta
```

Alpha-Beta

- Une façon standard d'écrire l'algorithme est d'utiliser un Negamax :
- Il n'y a qu'une seule fonction récursive.
- On inverse le signe de l'évaluation à chaque niveau de façon à maximiser à tous les niveaux.
- Ecrire le pseudo code d'un Negamax avec coupes Alpha-Beta.

Alpha-Beta

```
negamax (depth, alpha, beta)
  if depth == 0
    return evaluation (currentPlayer)
  for m in possible moves
      play (m)
      e = -negamax (depth – 1, -beta, -alpha)
      undo (m)
      if e > alpha
        alpha = e
        if alpha ≥ beta
          return beta
  return alpha
```

Approfondissement itératif

- On commence par un recherche à profondeur 1.
- Puis on fait une recherche à profondeur 2.
- Puis profondeur 3.
- Etc. jusqu'à ce que le temps de réflexion soit épuisé.
- Comportement temps réél.
- Permet de réutiliser l'ordre des coups de l'itération précédente => coupes Alpha-Beta.
- Complexité = $b + b^2 + b^3 + \ldots + b^{depth} = O(b^{depth})$

Coups qui tuent

- L'heuristique des coups qui tuent permet d'envisager un bon coup en premier pour maximiser le nombre de coupes Alpha-Beta.
- On mémorise pour chaque profondeur de recherche le dernier coup qui a permis de faire une coupe Alpha-Beta à cette profondeur.
- Lorsqu'on revient à cette profondeur on essaie le coup qui tue en premier.
- Ecrire un Alpha-Beta avec des coups qui tuent.

Coups qui tuent

```
negamax (depth, alpha, beta)
  if depth == 0
    return evaluation (currentPlayer)
  if killer [rootDepth - depth] is legal
    move killer [rootDepth - depth] in front of the possible moves
  for m in possible moves
      play (m)
      e = -negamax (depth – 1, -beta, -alpha)
      undo (m)
      if e > alpha
        alpha = e
        if alpha ≥ beta
          killer [rootDepth - depth] = m
          return beta
  return alpha
```

Heuristique de l'historique

- L'heuristique de l'historique permet de trier les coups à essayer.
- Chaque coup possible est associé à une note.
- Pour chaque coup ayant permis une coupe Alpha-Beta on ajoute 4^depth à sa note.
- Les coups possibles sont triés par notes décroissantes dans l'Alpha-Beta.
- Ecrire un Alpha-Beta avec l'heuristique de l'historique.

Heuristique de l'historique

```
negamax (depth, alpha, beta)
  if depth == 0
    return evaluation (currentPlayer)
  sort possible moves using history
  for m in possible moves
      play (m)
      e = -negamax (depth – 1, -beta, -alpha)
      undo (m)
      if e > alpha
        alpha = e
        if alpha ≥ beta
          history [code (m)] += 4^depth
          return beta
  return alpha
```