IA pour les Jeux
Licence Informatique et Vidéoludisme

Nicolas Jouandeau

n@up8.edu

2026

évaluation

- C/C++ Racket Ocaml Prolog Pike Java sh Python
- évaluation : TPs + PRJ
- TP individuel
- PRJ en groupe (2-4)
  - compétition Breakthrough
  - dès que possible
  - un rendu (programme et rapport 1p) par groupe
  - `https://ludii.games/`
- ia orientée jeux de plateaux (échecs-like)
  - 2 joueurs en tour par tour : breakthrough
  - `https://www.chessprogramming.org`
- ia de jeux de stratégie temps-réel (Real Time Battle)
  - bataille de tank en temps réel, 1 tank par personne
  - `https://realtimebattle.sourceforge.net/`

C/C++

- solution impérative avec `struct` et sans `class`
- avec conteneurs C++ (`list`, `vector`, `unordered_map`)
- `https://en.cppreference.com/w/`

Racket et Ocaml

- solution fonctionnelle
- `https://racket-lang.org/`
- `https://ocaml.org/`

Prolog

- solution descriptive
- `https://www.swi-prolog.org/`

Pike

- syntaxe C interprétée garbage-collecté orienté type entier et string
- `https://pike.lysator.liu.se/`

Java

- langage objet compilé pour JVM
- `https://www.java.com/fr/`

sh

- Bourn shell, abrégé `bsh`, lui-même abrégé `sh`
- interpréteur de command shell des environnements UNIX
- `http://www.opengroup.org/unix/online.html`

Python

- apprentissage automatique (machine learning)
- avec NumPy, Keras, PyTorch et TensorFlow
- `https://www.python.org/`
- `https://numpy.org/`
- `https://keras.io/`
- `https://pytorch.org/`
- `https://www.tensorflow.org/?hl=fr`

Définition de l'IA

- compréhension et la construction d'entités intelligentes
- intelligence : processus de la pensée dont l'objectif est de percevoir, comprendre, prévoir, manipuler un monde plus étendu que soi-même
- objectif de l'IA : systématiser et automatiser les tâches intellectuelles

Deux définitions

- Penser et agir rationnellement (i.e. conformément à ses connaissances et à une fonction d'évaluation)
- Penser et agir comme un humain (i.e. conformément au test de Turing)

Domaines concernés

- les jeux
- et également la planification, la programmation d'agents et de systèmes autonomes, le diagnostic, la robotique, la compréhension des langages

Sociétés savantes, conférences et revues

- Association for the Advancement of Artificial Intelligence (AAAI), Special Interest Group on Artificial Intelligence (ACM-SIGAI), Association for Computational Linguistics (ACL), International Computer Games Association (ICGA)
- IJCAI, ECAI, ICML, NeurIPS, IEEE-TAAI, IEEE-CoG, TCIAIG, ACG, CG

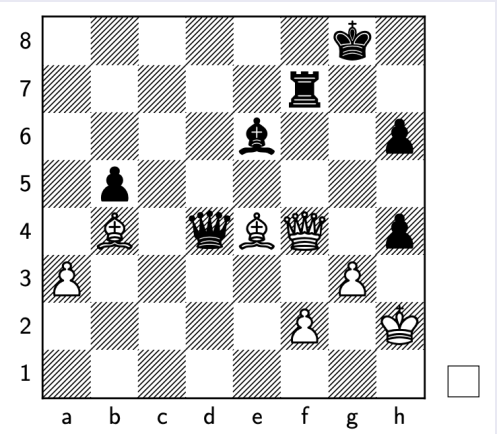
fonction heuristique d'évaluation : les échecs

- King, Queen, Rook, Bishop, kNight et Pawn
- blanc en majuscule et noir en miniscule
- D (respect. d) les pions doublés (2 pions sur une colonne)
- S (respect. s) les pions bloqués (pièce devant)
- I (respect. i) les pions isolés (pas de pions dans les colonnes adj.)
- M (respect. m) la mobilité
- f fonction d'évaluation pour le joueur blanc
- $f = 200(N_K - N_k) + 9(N_Q - N_q) + 5(N_R - N_r) + 3(N_B + N_N - N_b - N_n) + N_P - N_p - (D - d + S - s + I - i)/2 + (M - m)/10$

fonction heuristique d'évaluation : les échecs

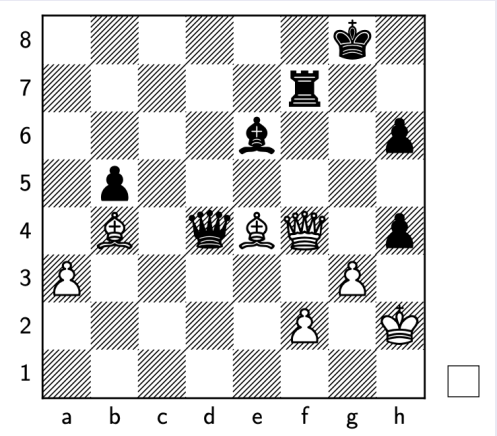
- $f = 200(N_K - N_k) + 9(N_Q - N_q) + 5(N_R - N_r) + 3(N_B + N_N - N_b - N_n) + N_P - N_p - (D - d + S - s + I - i)/2 + (M - m)/10$

exemple de position



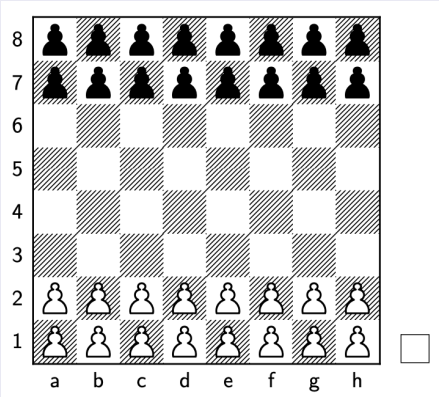
évaluation d'une position

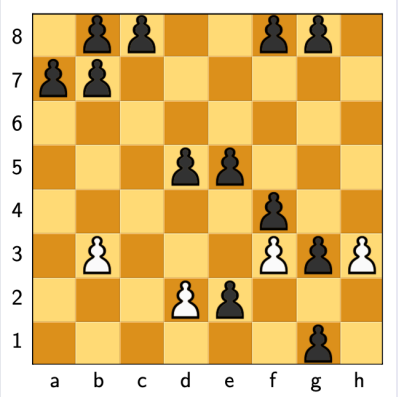
- $f = 200(N_K - N_k) + 9(N_Q - N_q) + 5(N_R - N_r) + 3(N_B + N_N - N_b - N_n) + N_P - N_p - (D - d + S - s + I - i)/2 + (M - m)/10$
  $= 5(0 - 1) + 3(2 - 1) - (0 - 2 + 0 - 1)/2 + (43 - 39)/10$
  $= -0.1$



Breakthrough (usuellement 8x8)

- jouer uniquement avec des pions (2 lignes par coté)[a]
- déplacement devant et en diag de 1 case
- capture uniquement en diag de 1 case
- victoire à la première promotion



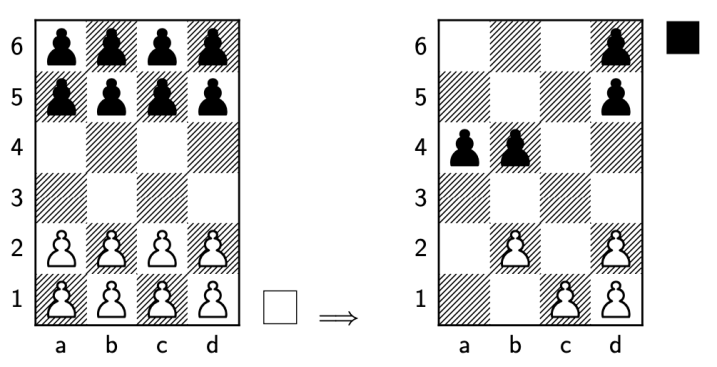


Breakthrough : les variantes

- misère : le joueur qui fait une promotion perd
- à captures forcées : les captures sont obligatoires si elles sont possibles
- à coups simultanés : chaque joueur annonce son coup et en cas de collision, les coups s'annulent et personne ne bouge, un arbitre annonce les coups nuls
- à information imparfaite : chaque joueur ne voit que ses pions, parfois un pion peut disparaitre (ce qui signale une capture adverse), il ne sait pas quand il capture en diagonal, un déplacement devant un pion est annulé si la case n'est pas vide, un arbitre annonce les coups nuls

Breakthrough (2)

- board de 8x8 : 3^{64} positions $\approx 10^{30}$
- board de 6x4 : 3^{24} positions $\approx 10^{11}$
- exemple de séquence : `2a3a-5a4a 2b3b-4a3b 2c3b-6a5a 3a4a-5b4a 3b4a-6b5b 4a5b-6c5b 1a2a-5a4a 2a3a-5b4b 3a4b-5c4b 1b2b-????`
- quel est le meilleur coup pour noir ?



A* : recherche de plus court chemin

- un état s = une position = un ensemble de pions sur un plateau
- p_i la position initiale
- p_f la position finale
- n itérations
- `free`(s) retourne vrai si s n'est pas en collision
- `nextMoves`(s) retourne la liste des mouvements en s
- `applyMove`(s, m) applique m sur s
- `dist`(a, b, c) estime la distance de a à c passant par b
- $\mathcal{H}$ la table de hashage des positions déjà évaluées
  $\mathcal{H}[s'] = s$ ssi $s' \leftarrow$ `applyMove` (s, m)
- $\mathcal{L}$ la liste des positions à évaluer

A* : recherche de plus court chemin

```
1  fonction astar ( p_i, p_f, n ) :
2      L ← { p_i } ;
3      H ← ∅ ;
4      for n times do
5          if |L| == 0 then break ;
6          s ← L.pop_front ( ) ;
7          if s == p_f then
8              return mkSolution ( H, s, p_i ) ;
9          M ← nextMoves ( s ) ;
10         for each m ∈ M do
11             s' ← applyMove ( s, m ) ;
12             if free (s') then
13                 if s' ∉ H then
14                     H[s'] ← s ;
15                     L ← L + s' ;                    // by distance order
16                 else
17                     if dist (p_i, H[s'], p_f) +1 > dist (p_i, s', p_f) then
18                         H[s'] ← s ;
19                         L ← L + s' ;                // by distance order
20     return ∅ ;
```

A* : retourner le chemin trouvé

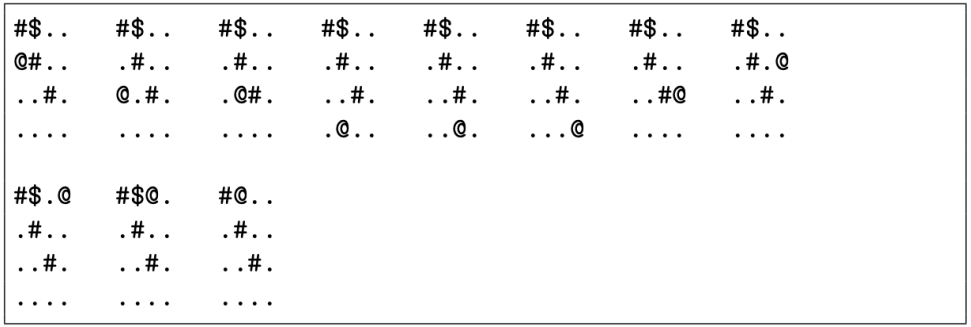
```
1 fonction mkSolution ( H, s, e ) :
2     S ← ∅ ;
3     while not-interrupted do
4         S.push_front ( s ) ;
5         if s == e then break ;
6         s ← H[s] ;
7     return S. ;
```

exemple de problème

- grille de 4x4 avec obstacles notés `#`
- un personnage `@` et un objectif `$`

```
#$..
@#..
..#.
....
```

A* : solution

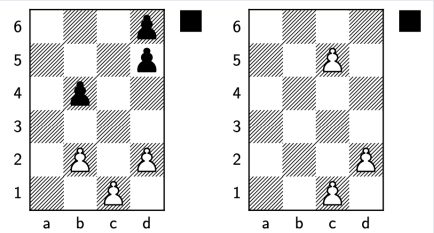


- `drdrruuull`

chemins le plus long possible ?

- dans une grille de 4x4 ? 5x5 ? 6x6 ? ...

séquence la plus courte à Breakthrough à captures forcées



4b3c-2b3c
5d4d-3c4d
6d5c-4d5c#

exemple de problèmes possibles

- séquence la + courte (respect. + longue) pour une position
- positions atteignables avec une longueur de séquence fixée à partir d'une position
- positions à une séquence de longueur fixée de la victoire

Depth Limited Search (DLS)

- recherche en profondeur limitée
  - jusqu'à un état *WIN*
  - interrompue sur un état *LOST*
  - interrompue à la profondeur *DLS_MAX_DEPTH*
- variables globales
  - *current_s* la position courante
  - $\mathcal{H}$ la table de hashage des profondeurs des états évaluées
  - *solved* un booléen à vrai si une solution est trouvée
- `applyMove`(*m*) joue/ajoute *m* sur *current_s*
- `undoMove`(*m*) déjoue/retire *m* sur *current_s*
- `applyMove`(*m*) et `undoMove`(*m*) modifient *current_s*

problème d'horizon de la fonction d'évaluation / profondeur

- évaluation à $DEPTH \neq$ évaluation à $(DEPTH + 1)$

DLS : recherche en profondeur limitée

```
1  fonction DLS ( d ) :
2      if solved then return ;
3      H[current_s] ← d ;
4      if h (current_s) < h (best_s) then
5          best_s ← current_s ;
6      if current_s == WIN then
7          solved ← true ;
8          return ;
9      if current_s == LOST or d == DLS_MAX_DEPTH then
10         return ;
11     M ← nextMoves ( ) ;
12     for each m ∈ M do
13         applyMove ( m ) ;
14         if current_s ∉ H or H[current_s] > d then
15             DLS ( d + 1 ) ;
16         undoMove ( m ) ;
17         if solved then return ;
```

DLS : recherche en profondeur limitée

- sans état global *current_s*
- avec un état local *s*

```
1  fonction DLS ( s, d ) :
2      if solved then return ;
3      H[s] ← d ;
4      if h (best_s) > h (s) then
5          best_s ← s ;
6      if s == WIN then
7          solved ← true ;
8          return ;
9      if s == LOST or d == DLS_MAX_DEPTH then
10         return ;
11     M ← nextMoves ( s ) ;
12     for each m ∈ M do
13         s' ← applyMove ( s, m) ;
14         if s' ∉ H or H[s'] > d then
15             DLS ( s', d + 1 ) ;
16         if solved then break ;
```

DLS : recherche en profondeur limitée

- avec la solution : *solution_size* et *best_s*

```
1  fonction DLS ( s, d ) :
2      if solution_size ≠ 0 then return ;
3      H[s] ← d ;
4      if h (best_s) > h (s) then
5          best_s ← s ;
6      if s == WIN then
7          solution_size ← d ;
8          return ;
9      if s == LOST or d == DLS_MAX_DEPTH then
10         return ;
11     M ← nextMoves ( s ) ;
12     for each m ∈ M do
13         s' ← applyMove ( s, m) ;
14         if s' ∉ H or H[s'] > d then
15             solution [ d ] ← m ;
16             DLS ( s', d + 1 ) ;
17         if solution_size ≠ 0 then break ;
```

DLS : initialiser la recherche

- fixer la profondeur max
- vider la table des états déjà évalués
- initialiser la taille de la solution
- définir *best_s* avec l'état courant
- lancer la recherche en profondeur limitée

```
1 DLS_MAX_DEPTH ← 10 ;
2 H ← ∅ ;
3 solution_size ← 0 ;
4 best_s ← p ;
5 DLS ( p, 0 ) ;
```

Breadth First Search (BFS)

- recherche en largeur d'abord (limitée en profondeur)
  - jusqu'à un état *WIN*
  - interrompue sur un état *LOST*
  - interrompue à la profondeur *BFS_MAX_DEPTH*
- variables globales
  - *solution_state* l'état correspondant à la victoire trouvée
  - *best_s* la meilleure solution trouvée
- variables locales
  - $\mathcal{L}$ la liste des états à évaluer
  - d la profondeur courante
  - $\mathcal{L}'$ la liste des fils de $\mathcal{L}$

Breadth First Search (BFS)

```
1  fonction BFS ( L, d ) :
2      if solution_state ≠ 0 then return ;
3      S ← ∅ ;
4      for each s ∈ L do
5          M ← nextMoves ( s ) ;
6          for each m ∈ M do
7              s' ← applyMove ( s, m ) ;
8              S ← S + (s', m) ;
9      L' ← ∅ ;
10     for each (s, m) ∈ S do
11         if s ∉ H then
12             H[s] ← m ;
13             if h (best_s) > h (s) then best_s ← s ;
14             if s == WIN then
15                 solution_state ← s ;
16                 return ;
17             if s == LOST or d == BFS_MAX_DEPTH then
18                 return ;
19             L' ← L' + s ;
20     BFS ( L', d + 1 ) ;
```

défaut de DLS

- première solution possiblement sous-optimale
- rechercher toutes les solutions pour obtenir la solution optimale

défaut de BFS

- première solution = solution optimale
- coût du calcul et du stockage de $\mathcal{L}$

Iterative Deepening Search (IDS)

- recherche DLS à profondeur itérative
- première solution = solution optimale
- pas de calcul et de stockage de $\mathcal{L}$

```
1 fonction IDS ( p ) :
2     solution_size ← 0 ;
3     best_s ← p ;
4     for depth ∈ [1; IDS_MAX_DEPTH ] do
5         H ← ∅ ;
6         DLS_MAX_DEPTH ← depth ;
7         DLS ( p , 0 ) ;
8         if solution_size ≠ 0 then break ;
```

représentation courante du matériel (jeu de plateau)

- un état du jeu (*i.e.* une position du jeu)
  - des constantes identifiant les types de pièces
  - une grille 1D ou 2D (pour le plateau)
  - une taille de grille
  - une valeur de tour
- une pièce
  - un type
  - une position sur la grille
- un coup
  - une position intiale
  - une position finale

fonctions utiles (jeu de 1 à N joueurs)

- initialiser l'état du jeu
- afficher l'état courant
- lister les coups possibles
- jouer un coup choisi (MAJ l'état courant)
- jouer un coup aléatoire (MAJ l'état courant)
- déjouer un coup (MAJ l'état courant)
- dire si un état est terminal
- donner le score d'un état terminal
- donner une évaluation heuristique d'un état non terminal
- donner une clé identifiant un état
- jouer un playout (partie aléatoire)

fonctions utiles (jeu à 2 joueurs)

- joueur random
- joueur minimax à profondeur fixée
- jouer 1 partie avec deux joueurs
- jouer N parties (joueurA, joueurB)
- jouer N parties (joueurB, joueurA)
- afficher le nombre de victoires de chaque joueur sur N parties

premiers tests

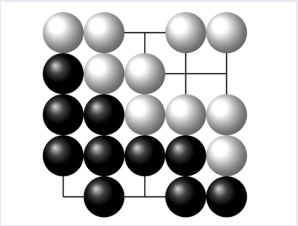
- réduction du problème (plateau 5x4)
  - évaluation de la profondeur pour une résolution minimax complète
  - test du joueur aléatoire contre le joueur minimax

développement

- revenir sur un plateau + grand
  - test de la fonction heuristique pour l'évaluatiuon des états non terminaux
  - ajout de nouvelles fonctions heuristiques
  - ajout de nouveaux joueurs (*i.e.* de nouveaux algorithmes)

représentation "bitboard" pour jouer plus vite

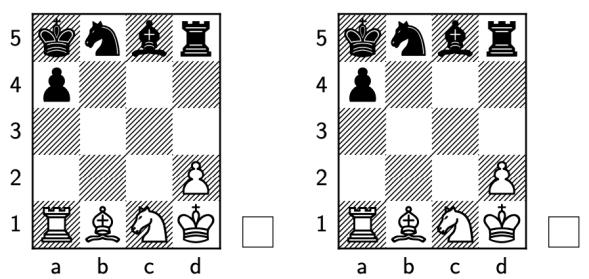
- conversion des pions en représentation binaire
- 1 position => - de mémoire => - de swap => + rapide
- exemple sur un plateau de go



- 11011011000011100001000000 (pour blanc)
- 00000100000110001111001011 (pour noir)
- plateau 5x5 -> 2x25 bits par position
- appliquer un coup = appliquer 2 opérations 32bits
- obtenir les coups possibles avec `not(blanc xor noir)` (`~(blanc ^ noir)` en C avec les entiers non signés)

cycles (une particularité de certains jeux)

- cycle = séquence de coups revenant sur une position précédente
- exemple sur un plateau de Microchess

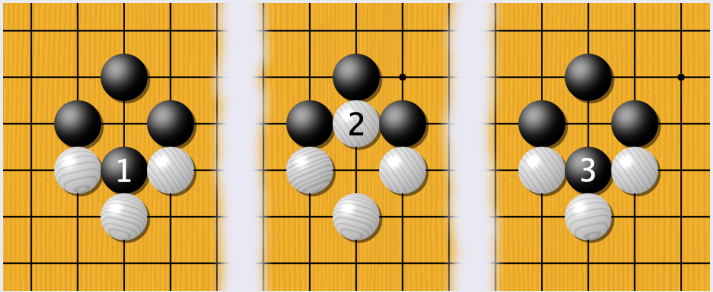


1b2b-5c4d
2b1b-4d5c

- considérer le tour de jeu dans l'état produit des positions différentes, ce qui fait que deux positions identiques pourrait ne pas avoir les mêmes statistiques
- les cycles produisent des boucles infinies dans les descentes

éviter les cycles (la règle du ko au go)

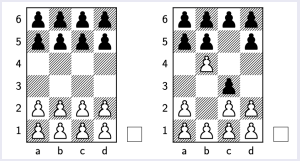
- certains jeux interdisent les cycles dans leurs règles
- règle du ko : interdiction de jouer un coup revenant sur la position précédent la position courante
- exemple sur un plateau de go



- 1 et 2 sont des coups légaux
- 3 est interdit par la règle du ko

transpositions

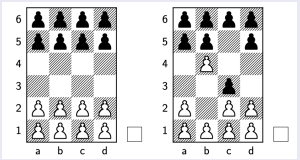
- variation d'une séquence qui mène à la même position
  - deux positions distinctes A et B
  - deux séquences distinctes 1 et 2
  - partant de A, appliquer la séquence 1 mène à B
  - partant de A, appliquer la séquence 2 mène à B
  - A est une transposition de B
  - B est une transposition de A
- exemple à breakthrough



- séquences de coups pour blanc (sans considérer noir)
  - `2b3b` puis `3b4b` (abrégée `f` puis `f`)
  - `2b3a` puis `3a4b` (abrégée `l` puis `r`)
  - `2b3c` puis `3c4b` (abrégée `r` puis `l`)

transpositions

- coups de blanc en majuscule et coups de noir en minuscule
- `f` pour forward, `l` pour left et `r` pour right



- 7 transpositions considérant les pions partant de `b2` et `c5`
  - `FfFf`, `FlFr`, `FrFl`
  - `LfRf`, `LrRl`
  - `RfLf`, `RrLl`

en pratique

- ne pas considérer les transpositions
  - crée des positions identiques avec des statistiques incomplets
  - multiplie les positions inutilement en mémoire
- considérer les transpositions
  - stocker les positions dans $\mathcal{H}$
    - une table de hashage (*i.e.* `unordered_map`)
    - un arbre (*i.e.* `map`)
      arbre binaire de recherche ABR
      arbre à équilibrage automatique AVL
      arbre bicolore
  - identifier une position par une clé
    - rapidité de la fonction de hashage
    - taille en mémoire de la table des positions
    - question des collisions (une clé pour deux positions)

pour minimiser l'espace mémoire nécessaire

- une position -> une clé
- une clé -> une position
- $\mathcal{H}$[clé] contient les statistiques de la position
- recherche, ajout et suppression en $O(log(n))$ au pire
- partant d'une position
  - calculer la liste des coups possibles
  - jouer chaque coup pour obtenir la nouvelle position
  - rechercher les statistiques de chaque nouvelle position
  - décider du meilleur coup

résoudre un jeu

- prédire le résultat (victoire/défaite) à partir de n'importe quelle position en supposant que les joueurs jouent parfaitement
- résolution ultra-faible
  - démontrer le résultat pour le premier joueur à partir de la position initiale
  - par application d'un principe
- résolution faible
  - avoir un algorithme qui donne les coups à jouer à partir de la position initiale
  - prédit la victoire pour un des joueurs
  - pas obligatoirement optimal contre un joueur qui joue mal
- résolution forte
  - avoir un algorithme qui donne les meilleurs coups à jouer à partir de n'importe quelle position

quelques jeux résolus

- tic tac toe (résolution triviale)
  - chaque joueur peut forcer un match nul
- tic tac toe 3D
  - achevée en 1980 par O. Patashnik et V. Allis
  - résolution faible
  - le premier joueur gagne
- awalé (variante Oware)
  - achevée en 2002 par J.W. Romein et H.E. Bal
  - résolution forte en 51h de calcul avec 144 cpu
  - évaluation de 889 milliards d'états
  - la partie parfaite se solde par un match nul
- dames
  - achevée en 2007 par J. Schaeffer
  - déplacements courts de dames uniquement
  - résolution faible en 18 ans de calcul
  - évaluation de 10^{14} états
  - la partie parfaite se solde par un match nul

mieux jouer avec une base de fins de partie

- objectif : évaluer plus rapidement (et espérer mieux jouer)
- issue : victoire, défaite, égalité (selon les jeux et les règles)
- certaines combinaisons de matériel et certaines configurations ont toujours la même issue
- sont elles atteignables ? est il intéressant de les mémoriser ? lesquelles mémoriser ?
  - quelle est la longueur min/max/moyenne d'une partie (contre random, contre IDS, ...) ?
  - quel est le facteur de branchement min/max/moyen au cours d'une partie ?
  - quelle est la probabilité de retrouver une configuration à profondeur 10 ?
  - faut il jouer off-line avec bcp de simulations et stocker les positions ambiguës avec peu de simulations ?
  - faut il partir des fins de parties gagnées et rechercher des configurations avec une issue similaire ?

Minimax (J. V. Neumann 1928)

- un état s
- un jeu à deux joueurs
- le joueur *root* maximise une évaluation `f`
- le joueur adverse minimise `f`
- une imbrication mutuelle des fonctions `maxi` et `mini`
- une profondeur d
- une fonction d'évaluation `f` pour le joueur *root*
  - `f`(s) = valeur de la position s
  - $\texttt{f}_{mini}()$ = valeur minimum de `f`
  - $\texttt{f}_{maxi}()$ = valeur maximum de `f`
- itérer l'évaluation en profondeur tant que
  - s n'est pas terminal
  - la profondeur d n'est pas *MINIMAX_MAX_DEPTH*

fonction `maxi`

```
1 fonction maxi ( s, d ) :
2     if d == MINIMAX_MAX_DEPTH then return f ( s ) ;
3     M ← nextMoves ( s ) ;
4     if |M| == 0 then return f ( s ) ;
5     best_v ← f_mini ( ) −1 ;
6     for each m ∈ M do
7         v ← mini ( applyMove ( s , m ) , d + 1) ;
8         if v > best_v then best_v ← v ;
9     return best_v ;
```

fonction `mini`

```
1 fonction mini ( s, d ) :
2     if d == MINIMAX_MAX_DEPTH then return f ( s ) ;
3     M ← nextMoves ( s ) ;
4     if |M| == 0 then return f ( s ) ;
5     best_v ← f_maxi ( ) +1 ;
6     for each m ∈ M do
7         v ← maxi ( applyMove ( s , m ) , d + 1) ;
8         if v < best_v then best_v ← v ;
9     return best_v ;
```

Negamax

- une évaluation `f` dépendant du tour de jeu
- une profondeur max *NEGAMAX_MAX_DEPTH*

fonction `negamax`

```
1 fonction negamax ( s, d ) :
2     if d == NEGAMAX_MAX_DEPTH then return f ( s ) ;
3     M ← nextMoves ( s ) ;
4     if |M| == 0 then return f ( s ) ;
5     best_v ← ∅ ;
6     for each m ∈ M do
7         v ← negamax ( applyMove ( s , m ), d + 1) ;
8         if v > best_v then best_v ← v ;
9     return best_v ;
```

Alpha-beta (1956-75, sur une idée de J. McCarthy)

- Negamax avec des coupes α et β
- une profondeur max *ALPHABETA_MAX_DEPTH*
- coupe α (définie par *a*) et β (définie par *b*)

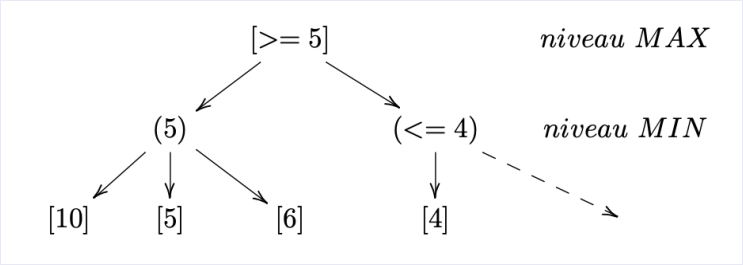
fonction `alphabeta`

```
1  fonction alphabeta (s, d, a, b) :
2      if d == ALPHABETA_MAX_DEPTH then return f ( s ) ;
3      M ← nextMoves ( s ) ;
4      if |M| == 0 then return f ( s ) ;
5      { best , mval } ← { ∅ , ∅ } ;
6      for each m ∈ M do
7          mval ← alphabeta ( applyMove ( s , m ) , d + 1 , a, b ) ;
8          if isMaxNode ( s ) then
9              best ← max ( mval, best ) ;
10             if best >= b then return best ;
11             a ← max ( a , best ) ;
12         else
13             best ← min ( mval, best ) ;
14             if best <= a then return best ;
15             b ← min ( b , best ) ;
16     return best ;
```

Alpha-beta

- α est initialisé à $+\infty$
- les nœuds les plus à gauche sont toujours évalués
- l'ordre d'énumération des fils => nombre de coupes

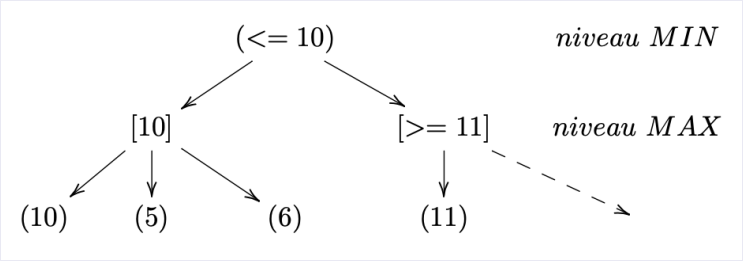
coupe α



Alpha-beta

- β est initialisé à $-\infty$
- les nœuds les plus à gauche sont toujours évalués
- l'ordre d'énumération des fils => nombre de coupes

coupe β



Conclusion

- DLS
- BFS (solution optimale)
- IDS (solution équivalente à BFS dans un arbre)
- fonction d'évaluation heuristique et problème d'horizon
- Minimax
- Alpha-beta (coupes alpha et coupes beta)
- cycles, transpositions (réduire les calculs)
- représentation bitboard (réduire l'espace mémoire)
- base d'ouvertures, bases de finales et précalculs