

IA pour les Jeux

Licence Informatique et Vidéoludisme

Recherche arborescente

Nicolas Jouandeau

n@up8.edu

2024

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
 - $f_{\text{mini}}()$ = valeur minimum de f
 - $f_{\text{maxi}}()$ = 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    $\mathcal{M} \leftarrow \text{nextMoves} ( s )$  ;  
4   if  $|\mathcal{M}| == 0$  then return f ( s ) ;  
5    $\text{best\_v} \leftarrow f_{\text{mini}} ( ) - 1$  ;  
6   for each m  $\in \mathcal{M}$  do  
7     v  $\leftarrow \text{mini} ( \text{applyMove} ( s , m ) , d + 1 )$  ;  
8     if v >  $\text{best\_v}$  then  $\text{best\_v} \leftarrow v$  ;  
9   return  $\text{best\_v}$  ;
```

fonction `mini`

```
1 fonction mini ( s , d ) :  
2   if d == MINIMAX_MAX_DEPTH then return f ( s ) ;  
3    $\mathcal{M} \leftarrow \text{nextMoves} ( s )$  ;  
4   if  $|\mathcal{M}| == 0$  then return f ( s ) ;  
5    $\text{best\_v} \leftarrow f_{\text{maxi}} ( ) + 1$  ;  
6   for each m  $\in \mathcal{M}$  do  
7     v  $\leftarrow \text{maxi} ( \text{applyMove} ( s , m ) , d + 1 )$  ;  
8     if v <  $\text{best\_v}$  then  $\text{best\_v} \leftarrow v$  ;  
9   return  $\text{best\_v}$  ;
```

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    $\mathcal{M} \leftarrow \text{nextMoves} ( s )$  ;  
4   if  $|\mathcal{M}| == 0$  then return f ( s ) ;  
5   best_v  $\leftarrow \emptyset$  ;  
6   for each m  $\in \mathcal{M}$  do  
7     v  $\leftarrow$  negamax ( applyMove ( s , m ), d + 1 ) ;  
8     if v > best_v then best_v  $\leftarrow$  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*)

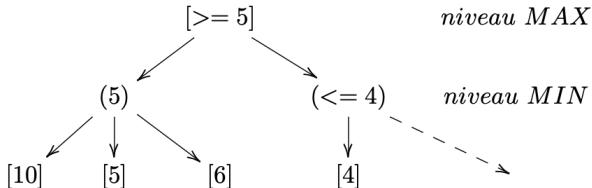
fonction *alphabeta*

```
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 \Rightarrow nombre de coupes

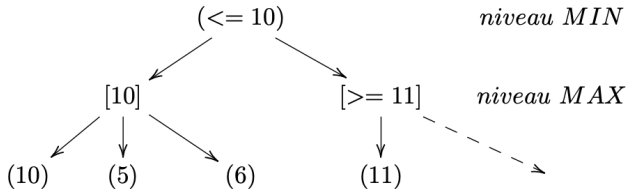
coupe α



Alpha-beta

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

coupe β



Proof Number Search (1988, V. Allis)

- ▶ prouver qu'une position est perdue ou gagnée
- ▶ une valeur DN (Disproof Number) = estimation du coût de la preuve de la défaite
- ▶ une valeur PN (Proof Number) = estimation du coût de la preuve de la victoire
- ▶ un booléen exp vrai si tous les fils sont dans l'arbre
- ▶ construction itérative en 3 étapes : sélection, expansion-évaluation, rétropropagation

PNS

- ▶ exp est initialisé à faux
- ▶ DN et PN sont initialisés à 1
- ▶ \mathcal{H} contient des triplets $\{DN, PN, exp\}$
- ▶ la recherche s'arrête quand le nœud racine est prouvé perdu ou gagné

```
1 fonction PNS ( s ) :  
2    $\mathcal{H} \leftarrow \emptyset$  ;  
3    $\mathcal{H}[s] \leftarrow \{1, 1, false\}$  ;  
4   while not-interrupted do  
5      $s' \leftarrow selection(s)$  ;  
6     expansion (  $s'$  ) ;  
7     backpropagate (  $s'$  ) ;  
8     if  $DN_s == 0$  then return LOSS ;  
9     if  $PN_s == 0$  then return WIN ;
```

sélection PNS

- ▶ la sélection s'arrête au premier nœud dont les fils ne sont pas tous dans \mathcal{T}
- ▶ sélection du nœud avec le PN le plus petit

```
1 fonction selection ( s ) :  
2   if  $exp_s == false$  then return s ;  
3    $\mathcal{M} \leftarrow \text{nextMoves} ( s )$  ;  
4    $\{min, best\} \leftarrow \{\infty, \emptyset\}$  ;  
5   for each  $m \in \mathcal{M}$  do  
6      $s' \leftarrow \text{applyMove} ( s, m )$  ;  
7     if  $PN_{s'} > 0$  and  $PN_{s'} < min$  then  $\{min, best\} \leftarrow \{PN_{s'}, s'\}$  ;  
8   return selection ( best ) ;
```

variante

- ▶ ajouter un seuil min pour ne pas faire les évaluations considérées comme trop coûteuses

expansion-évaluation PNS

- ▶ ajouter de tous les nœuds fils dans \mathcal{T}
- ▶ noter les nœuds perdus ou gagnés

```
1 fonction expansion ( s ) :  
2    $exp_s \leftarrow true$  ;  
3    $\mathcal{M} \leftarrow nextMoves ( s )$  ;  
4   for each  $m \in \mathcal{M}$  do  
5      $s' \leftarrow applyMove ( s , m )$  ;  
6     if  $s' \notin \mathcal{H}$  then  
7        $\mathcal{H}[s'] \leftarrow \{1, 1, false\}$  ;  
8       if  $s' == WIN$  then  $\{DN_{s'}, PN_{s'}\} \leftarrow \{\infty, 0\}$  ;  
9       if  $s' == LOSS$  then  $\{DN_{s'}, PN_{s'}\} \leftarrow \{0, \infty\}$  ;
```

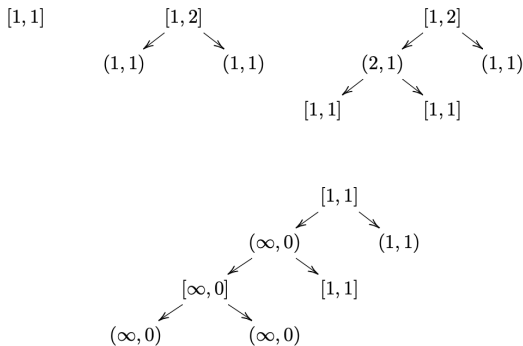
rétropropagation PNS

- ▶ coups root = nœuds-OU ou nœuds-MAX
- ▶ coups adverse = nœuds-ET ou nœuds-MIN

```
1 fonction backpropagate ( s ) :  
2    $\mathcal{M} \leftarrow \text{nextMoves} ( s )$  ;  
3    $\{min, sum\} \leftarrow \{\infty, 0\}$  ;  
4   if  $turn_s \% 2 == 0$  then  
5     for each  $m \in \mathcal{M}$  do  
6        $s' \leftarrow \text{applyMove} ( s, m )$  ;  
7       if  $min > DN_{s'}$  then  $min \leftarrow DN_{s'}$  ;  
8        $sum \leftarrow sum + PN_{s'}$  ;  
9      $\{DN_s, PN_s\} \leftarrow \{min, sum\}$  ;  
10  else  
11    for each  $m \in \mathcal{M}$  do  
12       $s' \leftarrow \text{applyMove} ( s, m )$  ;  
13       $sum \leftarrow sum + DN_{s'}$  ;  
14      if  $min > PN_{s'}$  then  $min \leftarrow PN_{s'}$  ;  
15     $\{DN_s, PN_s\} \leftarrow \{sum, min\}$  ;  
16  if  $s == \text{ROOT}$  then return ;  
17   $p \leftarrow \text{getParent} ( s )$  ;  
18  return backpropagate ( p ) ;
```

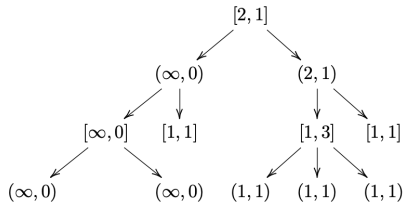
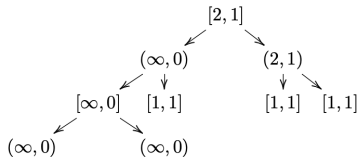
exemple

- ▶ nœud-OU (*i.e.* $[\min, \Sigma]$)
- ▶ nœud-ET (*i.e.* (Σ, \min))
- ▶ défaut $\langle 1, 1 \rangle$ | défaite $\langle 0, \infty \rangle$ | victoire $\langle \infty, 0 \rangle$



exemple (suite)

► résultat après 5 itérations



nombreuses variantes de PNS de 1988 à 2013

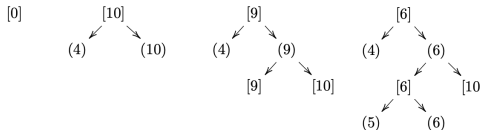
- ▶ PN^2 : évaluation = un autre PNS (1988, Breuker)
- ▶ PN^* : profondeur itérative récursive (1995-2001, Seo et al)
- ▶ DF-PN : best-first itératif (1995-2001, Seo et al)
- ▶ PDS : best-first (1998, Nagai)
- ▶ PPNS : PNS parallèle (1999, Kishimoto et Kotani)
- ▶ PDS-PN : PDS suivi de PNS (2002, Winands et al)
- ▶ JL-PNS : PNS distribué (2010, Wu et al)
- ▶ RPPNS : PNS parallèle randomisé (2010, Saito et al)
- ▶ PPN^2 : PN^2 parallèle (2011, Saffidine et al)
- ▶ SPDF-PN : DF-PN parallèle scalable (2013, Pawlewicz et Hayward)

Unbounded Best First Minimax (1998, R.E. Korf et D.M. Chiskering)

- ▶ Minimax à profondeur itérative en meilleur d'abord
- ▶ avec évaluation heuristique à chaque expansion
- ▶ \mathcal{H} contient les évaluations des états

```
1 fonction UBFM ( s ) :  
2    $\mathcal{H} \leftarrow \emptyset$  ;  
3    $\mathcal{H}[s] \leftarrow 0$  ;  
4   while not-interrupted do  
5      $s' \leftarrow \text{selection} ( s )$  ;  
6     expansion (  $s'$  ) ;  
7     backpropagate (  $s'$  ) ;
```

UBFM exemple



sélection UBFM

► descente minimax dans \mathcal{T}

```
1 fonction selection ( s ) :  
2    $\mathcal{M} \leftarrow \text{nextMoves} ( s )$  ;  
3    $\{min, max, best\} \leftarrow \{\infty, -\infty, \emptyset\}$  ;  
4   for each  $m \in \mathcal{M}$  do  
5      $s' \leftarrow \text{applyMove} ( s, m )$  ;  
6     if  $s' \notin \mathcal{H}$  then return  $s'$  ;  
7     if  $\text{turn}_s \% 2 == 0$  then  
8       if  $max < \mathcal{H}[s']$  then  $\{max, best\} \leftarrow \{\mathcal{H}[s'], s'\}$  ;  
9     else  
10      if  $min > \mathcal{H}[s']$  then  $\{min, best\} \leftarrow \{\mathcal{H}[s'], s'\}$  ;  
11  return selection ( best ) ;
```

expansion UBFM

► ajouter tous les fils de s dans \mathcal{T} par évaluation heuristique

```
1 fonction expansion ( s ) :  
2    $\mathcal{M} \leftarrow \text{nextMoves} ( s )$  ;  
3   for each  $m \in \mathcal{M}$  do  
4      $s' \leftarrow \text{applyMove} ( s, m )$  ;  
5     if  $s' \notin \mathcal{H}$  then  
6        $\mathcal{H}[s'] \leftarrow \text{eval} ( s' )$  ;
```

retropropagation UBFM

- MAJ des valeurs pour améliorer la prochaine descente

```
1 fonction backpropagate ( s ) :  
2    $\mathcal{M} \leftarrow \text{nextMoves} ( s )$  ;  
3    $\{min, max\} \leftarrow \{\infty, -\infty\}$  ;  
4   if  $turn_s \% 2 == 0$  then  
5     for each  $m \in \mathcal{M}$  do  
6        $s' \leftarrow \text{applyMove} ( s, m )$  ;  
7       if  $max < \mathcal{H}[s']$  then  $max \leftarrow \mathcal{H}[s']$  ;  
8      $\mathcal{H}[s] \leftarrow max$  ;  
9   else  
10    for each  $m \in \mathcal{M}$  do  
11       $s' \leftarrow \text{applyMove} ( s, m )$  ;  
12      if  $min > \mathcal{H}[s']$  then  $min \leftarrow \mathcal{H}[s']$  ;  
13     $\mathcal{H}[s] \leftarrow min$  ;  
14    if  $s == \text{ROOT}$  then return ;  
15     $p \leftarrow \text{getParent} ( s )$  ;  
16    return backpropagate ( p ) ;
```

évaluation Monte Carlo (1993, B. Brügmann)

- ▶ principe (1948, Fermi et Richtmyer) (1949, Ulam)
- ▶ faire une moyenne de parties aléatoires (1993, Brügmann)
- ▶ une partie aléatoire = *rollout* ou *playout*

évaluation MC

```
1  $sum \leftarrow 0$  ;  
2 for  $N$  times do  
3    $r \leftarrow \text{playout}(s)$  ;  
4    $sum \leftarrow r + sum$  ;  
5  $r \leftarrow sum/N$  ;
```

évaluation MC avec écart-type

```
1  $sum \leftarrow 0$  ;  
2  $sum2 \leftarrow 0$  ;  
3 for  $N$  times do  
4    $r \leftarrow \text{playout}(p)$  ;  
5    $sum \leftarrow r + sum$  ;  
6    $sum2 \leftarrow (r * r) + sum2$  ;  
7  $r_p \leftarrow sum/N$  ;  
8  $\sigma_{r_p} \leftarrow \sqrt{sum2/N - (r_p * r_p)}$  ;
```

playout

- ▶ jouer une partie sans la mémoriser
- ▶ s est terminal = fin de partie
- ▶ score = score de fin de partie
 - victoire/défaite = 1/0
 - victoire/nul/défaite = 1/0/-1
 - score de la victoire ou de la défaite

```
1 fonction playout ( s ) :  
2   while not terminal ( s ) do  
3      $\mathcal{M} \leftarrow \text{nextMoves} ( s ) ;$   
4      $m \leftarrow \text{random} ( \mathcal{M} ) ;$   
5      $s \leftarrow \text{applyMove} ( s, m ) ;$   
6   return score ( s ) ;
```

évaluation MC

- ▶ N évaluation MC à chaque tour
- ▶ fin de partie
 - quand on trouve une solution (jeu à 1 joueur)
 - quand un joueur gagne (jeu à 2 joueurs)
 - ou quand les 2 joueurs passent (jeu à 2 joueurs)

```
1  $\mathcal{M} \leftarrow \text{nextMoves} ( s ) ;$   
2 if  $|\mathcal{M}| == 0$  then return PASS ;  
3  $best \leftarrow \text{first} ( \mathcal{M} ) ; max \leftarrow 0 ;$   
4 for each  $m \in \mathcal{M}$  do  
5    $s' \leftarrow \text{applyMove} ( s, m ) ;$   
6    $w_i \leftarrow 0 ;$   
7   for N times do  
8      $r \leftarrow \text{playout} ( s' ) ;$   
9     if  $r == \text{WIN}$  then  $w_i \leftarrow w_i + 1 ;$   
10  if  $w_i > max$  then  $\{ best, max \} \leftarrow \{ m, w_i \} ;$   
11 return best ;
```

évaluation MCTS (2006, R. Coulom)

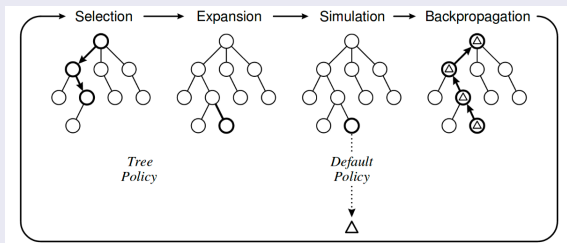


Figure extraite de A Survey of Monte Carlo Tree Search Methods par C. Browne et al (IEEE Trans. On Computational Intelligence and AI in Games, Vol. 4, No. 1, 2012).

```
1 fonction MCTS ( s ) :  
2   while not-interrupted do  
3      $s' \leftarrow \text{selection} ( s )$  ;  
4      $\mathcal{H}[s'] \leftarrow \{0,0\}$  ;  
5      $r \leftarrow \text{playout} ( s' )$  ;  
6     backpropagate (  $s', r$  ) ;  
7   return bestNext ( s ) ;
```

évaluation MCTS : sélection (2006, L. Kocsis et C. Szepesvári)

- ▶ résoud le dylemme entre exploration et exploitation
- ▶ formule $uct : (W_i/N_i) + K\sqrt{\log(N)/N_i}$
 - W_i le nombre de victoires au nœud i
 - N_i le nombre de playouts au nœud i
 - K la constante UCT souvent fixée à proximité de 0.4
 - N le nombre de playouts réalisés au nœud parent
- ▶ \mathcal{H} contient des couples $\{W, N\}$

```
1 fonction selection ( s ) :  
2   if terminal ( s ) then return s ;  
3    $\mathcal{M} \leftarrow \text{nextMoves} ( s )$  ;  
4    $\{max, best\} \leftarrow \{-1, \emptyset\}$  ;  
5   for each  $m \in \mathcal{M}$  do  
6      $s' \leftarrow \text{applyMove} ( s, m )$  ;  
7     if  $s' \notin \mathcal{H}$  then return  $s'$  ;  
8      $new\_eval \leftarrow uct ( s, s' )$  ;  
9     if  $new\_eval > max$  then  $\{max, best\} \leftarrow \{new\_eval, m\}$  ;  
10  return selection ( best ) ;
```

évaluation MCTS : expansion et retropropagation

- ▶ \mathcal{H} contient des couples $\{W, N\}$
- ▶ pour une victoire, le score est $\{1, 1\}$
- ▶ pour une défaite, le score est $\{0, 1\}$
- ▶ K la constante *UCT* souvent fixée à proximité de 0.4
- ▶ N le nombre de playouts réalisés au nœud parent

```
1 fonction backpropagate ( s, score ) :  
2   if parent ( s ) ==  $\emptyset$  then return ;  
3    $\mathcal{H}[s] \leftarrow \mathcal{H}[s] + score$  ;  
4   return backpropagate ( parent ( s ), score ) ;
```


variantes parallèles de MCTS

- ▶ sans partage de \mathcal{T} (2007, Cazenave et Jouandeau)
- ▶ aux feuilles (2008, Cazenave et Jouandeau)
- ▶ avec partage de \mathcal{T} (2008, Cazenave et Jouandeau)
- ▶ avec mutex local (2008, Chaslot et al)
- ▶ avec mutex global (2009, Enzenberger et Muller)
- ▶ distribuée (2011, Yoshizoe et al)
- ▶ pour processeurs many-core (2015-17, Miroleimani et al)
- ▶ avec pipeline de motifs (2018, Miroleimani et al)

playout policy adaptation (2015, T. Cazenave)

- ▶ adapter la politique des playouts selon les résultats
- ▶ pendant un playout
 - utiliser une probabilité de sélection de coups \mathcal{P}
- ▶ après le playout
 - ajouter une fonction `adapt` selon le joueur victorieux

```
1 fonction MCTS_with_PPA ( s ) :  
2   while not-interrupted do  
3     s' ← selection ( s ) ;  
4      $\mathcal{H}[s'] \leftarrow \{0,0\}$  ;  
5     {r, seq} ← playout_with_PPA ( s',  $\mathcal{P}$  ) ;  
6     if r == WIN then  
7       adapt_PPA ( s',  $\mathcal{P}$ , seq, player ) ;  
8     else  
9       adapt_PPA ( s',  $\mathcal{P}$ , seq, opp ) ;  
10    backpropagate ( s', r ) ;  
11  return bestNext ( s ) ;
```

adaptation simple d'une politique \mathcal{P}

- ▶ renforcer les coups d'une séquence choisie
- ▶ => avoir une meilleure politique \mathcal{P}
- ▶ => s'adapter pour mieux jouer face à un adversaire

```
1 fonction adapt_simple (  $\mathcal{P}$ , seq ) :  
2   for each  $m \in seq$  do  
3     if  $m \notin \mathcal{P}$  then  $\mathcal{P}[m] \leftarrow 1$  ;  
4     else if  $\mathcal{P}[m] < 10$  then  $\mathcal{P}[m] \leftarrow \mathcal{P}[m] + 2$  ;  
5     else if  $\mathcal{P}[m] < 100$  then  $\mathcal{P}[m] \leftarrow \mathcal{P}[m] + 4$  ;  
6     else  $\mathcal{P}[m] \leftarrow \mathcal{P}[m] + 1$  ;
```

fonction adapt de PPA

- ▶ $K = 0.4$ et $\alpha = 1$
- ▶ rejoue le playout et adapte la politique de coups \mathcal{P}

```
1 fonction adapt_PPA ( s ,  $\mathcal{P}$  , seq , player ) :  
2    $\mathcal{P}_2 \leftarrow \mathcal{P}$  ;  
3   for each m  $\in$  seq do  
4     if winner == player then  
5       if m  $\notin \mathcal{P}_2$  then  $\mathcal{P}_2[m] \leftarrow \alpha$  ;  
6       else  $\mathcal{P}_2[m] \leftarrow \mathcal{P}_2[m] + \alpha$  ;  
7        $z \leftarrow \sum_{m \in s} e^{\mathcal{P}_2[m]}$  ;  
8       for each m from s do  
9          $\mathcal{P}_2[m] \leftarrow \mathcal{P}_2[m] - \frac{\alpha}{z} e^{\mathcal{P}_2[m]}$  ;  
10      s  $\leftarrow$  applyMove ( s , m ) ;  
11      player  $\leftarrow$  opponent ( player ) ;  
12     $\mathcal{P} \leftarrow \mathcal{P}_2$  ;
```

playout avec PPA

- ▶ sélection des coups selon \mathcal{P}
- ▶ \mathcal{P} peut changer à chaque playout
- ▶ probabilité de sélection définie par $e^{\mathcal{P}[m]} / \sum_{m \in \mathcal{M}} e^{\mathcal{P}[m]}$

```
1 fonction playout_with_PPA ( s,  $\mathcal{P}$  ) :  
2   seq  $\leftarrow \emptyset$  ;  
3   while not terminal ( s ) do  
4      $\mathcal{M} \leftarrow \text{nextMoves} ( s )$  ;  
5      $r \leftarrow \text{random} ( \sum_{m \in \mathcal{M}} e^{\mathcal{P}[m]} )$  ;  
6     for each  $m \in \mathcal{M}$  do  
7       if  $r \leq e^{\mathcal{P}[m]}$  then  
8         s  $\leftarrow \text{applyMove} ( s, m )$  ;  
9         seq  $\leftarrow \text{seq} + m$  ;  
10        break ;  
11    r  $\leftarrow r - e^{\mathcal{P}[m]}$  ;  
12  return { score ( s ), seq } ;
```

Nested Monte Carlo Search (2009, T. Cazenave)

- ▶ recherche MC enroulée
- ▶ pour guider la recherche sans heuristique
- ▶ mémorisation de la meilleur séquence de coups

```
1 fonction nested ( s, level ) :  
2   best_score ← -1 ;  
3   best_seq ← ∅ ;  
4   while not terminal ( s ) do  
5     if level = 1 then  
6       {m,seq} ← argmaxm ∈ M payout ( s, m ) ;  
7     else  
8       s' ← applyMove ( s, m ) ;  
9       {m,seq} ← argmaxm ∈ M nested ( s', level - 1 ) ;  
10    new_score ← score ( s, m ) ;  
11    if new_score > best_score then  
12      best_score ← new_score ;  
13      best_seq ← seq ;  
14    best_move ← best ( best_seq ) ;  
15    s ← applyMove ( s, best_move ) ;  
16  return score ( s ) ;
```

Nested Rollout Policy Adaptation (2011, C.D. Rosin)

- ▶ après N playouts, adapter $level$ fois N playouts
- ▶ $level \times N$ playouts OU $level \times (N \text{ playouts} + adapt)$
- ▶ i.e. N avec $\mathcal{P} \rightarrow N$ avec $\mathcal{P}' \rightarrow \dots$

avec $level = 2$ et $N = 3$

```
1 for 3 times do // appel NRPA avec level=1
2   for 3 times do // appel NRPA avec level=0
3     seq ← playout (  $\mathcal{P}$  );
4     adapt (  $\mathcal{P}$  );
5 return best_seq ;
```

avec $level = 3$ et $N = 10$

```
1 for 10 times do // appel NRPA avec level=2
2   for 10 times do // appel NRPA avec level=1
3     for 10 times do // appel NRPA avec level=0
4       seq ← playout (  $\mathcal{P}$  );
5       adapt (  $\mathcal{P}$  );
6     adapt (  $\mathcal{P}$  );
7 return best_seq ;
```

sélection et playout avec \mathcal{P}

- sélection basée sur n valeurs de \mathcal{P}

```
1 fonction select_NRPA (  $\mathcal{P}$ ,  $\mathcal{M}$  ) :  
2    $sum \leftarrow 0$  ;  
3   for each  $m \in \mathcal{M}$  do  
4      $sum \leftarrow sum + \mathcal{P}[m]$  ;  
5    $r \leftarrow \text{random} ( sum )$  ;  
6   for each  $m \in \mathcal{M}$  do  
7     if  $r \leq \mathcal{P}[m]$  then return  $m$  ;  
8      $r \leftarrow r - \mathcal{P}[m]$  ;
```

- mémoriser la séquence correspondant au playout

```
1 fonction playout_NRPA (  $s$ ,  $\mathcal{P}$  ) :  
2    $seq \leftarrow \emptyset$  ;  
3   while not terminal (  $s$  ) do  
4      $s \leftarrow \text{applyMove} ( s, \text{select\_NRPA} ( \mathcal{P}, \text{nextMoves} ( s ) ) )$  ;  
5      $seq \leftarrow seq + s$  ;  
6   return { score (  $s$  ),  $seq$  } ;
```


adaptation de \mathcal{P} aux résultats

► adaptation exponentielle aux valeurs de \mathcal{P}

```
1 fonction adapt_NRPA (  $\mathcal{P}$ , seq ) :  
2    $\{s, \mathcal{P}_2\} \leftarrow \{root, \mathcal{P}\}$  ;  
3   for each  $m \in seq$  do  
4     if  $m \notin \mathcal{P}_2$  then  $\mathcal{P}_2[m] \leftarrow \alpha$  ;  
5     else  $\mathcal{P}_2[m] \leftarrow \mathcal{P}_2[m] + \alpha$  ;  
6      $z \leftarrow 0$  ;  
7     for each  $m$  from  $s$  do  
8        $z \leftarrow z + e^{\mathcal{P}_2[m]}$  ;  
9     for each  $m$  from  $s$  do  
10       $\mathcal{P}_2[m] \leftarrow \mathcal{P}_2[m] - \frac{\alpha}{z} e^{\mathcal{P}_2[m]}$  ;  
11     $s \leftarrow \text{applyMove} ( s, m )$  ;  
12   $\mathcal{P} \leftarrow \mathcal{P}_2$  ;
```

modification de NMCS en NRPA

► playout et adapt avec \mathcal{P}

```

1 fonction nrpa ( s, level ,  $\mathcal{P}$  ) :
2   best_score  $\leftarrow$  -1 ;
3   best_seq  $\leftarrow$   $\emptyset$  ;
4   while not terminal ( s ) do
5     if level = 1 then
6       {m,seq}  $\leftarrow$  argmaxm $\in$ M playout_NRPA ( s, m ,  $\mathcal{P}$  ) ;
7     else
8       s'  $\leftarrow$  applyMove ( s, m ) ;
9       {m,seq}  $\leftarrow$  argmaxm $\in$ M nrpa ( s', level - 1 ) ;
10    new_score  $\leftarrow$  score ( s, m ) ;
11    if new_score > best_score then
12      best_score  $\leftarrow$  new_score ;
13      best_seq  $\leftarrow$  seq ;
14    adapt_NRPA (  $\mathcal{P}$  , best_seq ) ;
15    best_move  $\leftarrow$  best ( best_seq ) ;
16    s  $\leftarrow$  applyMove ( s, best_move ) ;
17  return score ( s ) ;

```

résoudre un problème par recherche arborescente

- ① définir des fonctions d'évaluation heuristique
- ② choisir un algorithme de référence (au pire random)
- ③ essayer un autre algorithme
- ④ comparer les résultats des fonctions d'évaluation
- ⑤ comparer les résultats des algorithmes
- ⑥ (éventuellement revenir au point ③)

améliorer la recherche arborescente

- ▶ adapter fonction de sélection et évaluation heuristique
- ▶ apprendre fonction de sélection et évaluation heuristique
- ▶ utiliser des bases de finales
- ▶ utiliser des bases d'ouverture
- ▶ paralléliser la recherche arborescente