IA pour les Jeux
Licence Informatique et Vidéoludisme

Recherche arborescente

Nicolas Jouandeau

n@up8.edu

2024

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 iterative 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

fonction `PNS` (s) :
- $\mathcal{H} \leftarrow \emptyset$;
- $\mathcal{H}[s] \leftarrow \{1\ ,\ 1\ ,\ false\}$;
- **while** *not-interrupted* **do**
  - $s' \leftarrow$ `selection` (s) ;
  - `expansion` (s') ;
  - `backpropagate` (s') ;
  - **if** $DN_s == 0$ **then return** $LOSS$;
  - **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

fonction `selection` $(\ s\ )$:
 if $exp_s == false$ **then return** s ;
 $\mathcal{M} \leftarrow$ `nextMoves` $(\ s\ )$;
 $\{min\ ,\ best\} \leftarrow \{\infty\ ,\ \emptyset\}$;
 for $each\ m \in \mathcal{M}$ **do**
  $s' \leftarrow$ `applyMove` $(\ s\ ,\ m\ )$;
  if $PN_{s'} > 0$ **and** $PN_{s'} < min$ **then** $\{min\ ,\ best\} \leftarrow \{PN_{s'}\ ,\ s'\}$;
 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

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

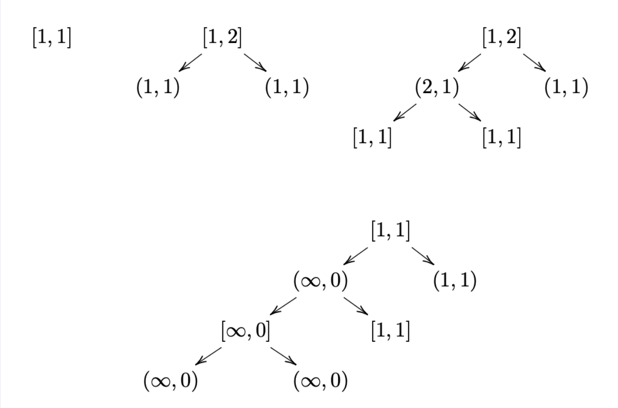
rétropropagation PNS

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

fonction `backpropagate` (s) :
- $\mathcal{M} \leftarrow$ `nextMoves` (s) ;
- $\{min\ ,\ sum\} \leftarrow \{\infty\ ,\ 0\}$;
- **if** $turn_s \% 2 == 0$ **then**
  - **for** *each* $m \in \mathcal{M}$ **do**
    - $s' \leftarrow$ `applyMove` (s , m) ;
    - **if** $min > DN_{s'}$ **then** $min \leftarrow DN_{s'}$;
    - $sum \leftarrow sum + PN_{s'}$;
  - $\{DN_s\ ,\ PN_s\} \leftarrow \{min\ ,\ sum\}$;
- **else**
  - **for** *each* $m \in \mathcal{M}$ **do**
    - $s' \leftarrow$ `applyMove` (s , m) ;
    - $sum \leftarrow sum + DN_{s'}$;
    - **if** $min > PN_{s'}$ **then** $min \leftarrow PN_{s'}$;
  - $\{DN_s\ ,\ PN_s\} \leftarrow \{sum\ ,\ min\}$;
- **if** $s == ROOT$ **then return** ;
- $p \leftarrow$ `getParent` (s) ;
- **return** `backpropagate` (p) ;

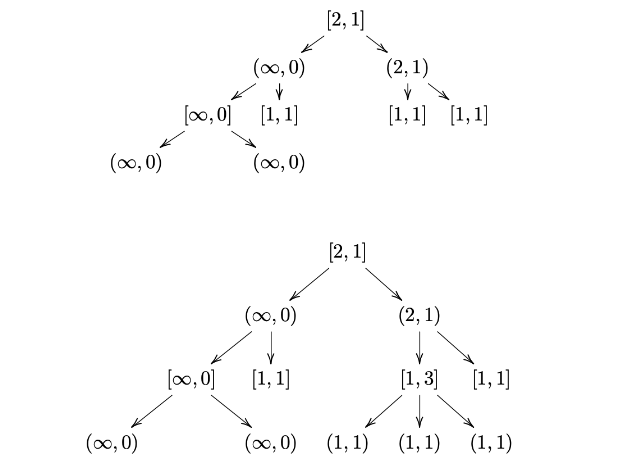
exemple

- nœud-OU (*i.e.* [min,Σ])
- nœud-ET (*i.e.* (Σ,min))
- défaut <1,1> | défaite <0,∞> | victoire <∞,0>



exemple (suite)

- résultat après 5 itérations



nombreuses variantes de PNS de 1988 à 2013

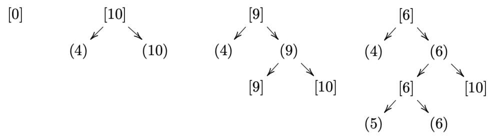
- PN^2 : évaluation = un autre PNS (1988, Breuker)
- PN* : profondeur itérative récursive (1995-2001, Seo et al)
- DF-PN : best-first itératif (1995-2001, Seo et al)
- PDS : best-first (1998, Nagai)
- PPNS : PNS parallèle (1999, Kishimoto et Kotani)
- PDS-PN : PDS suivi de PNS (2002, Winands et al)
- 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

fonction `UBFM` (s) :
    $\mathcal{H} \leftarrow \emptyset$;
    $\mathcal{H}[s] \leftarrow 0$;
    while *not-interrupted* **do**
        $s' \leftarrow$ `selection` (s) ;
        `expansion` (s') ;
        `backpropagate` (s') ;

UBFM exemple



sélection UBFM

- descente minimax dans $\mathcal{T}$

```
fonction selection ( s ) :
    M ← nextMoves ( s ) ;
    {min, max , best} ← {∞ , −∞ , ∅} ;
    for each m ∈ M do
        s′ ← applyMove ( s , m ) ;
        if s′ ∉ H then return s ;
        if turn_s %2 == 0 then
            if max < H[s′] then {max , best} ← {H[s′] , s′} ;
        else
            if min > H[s′] then {min , best} ← {H[s′] , s′} ;
    return selection ( best ) ;
```

expansion UBFM

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

```
fonction expansion ( s ) :
    M ← nextMoves ( s ) ;
    for each m ∈ M do
        s′ ← applyMove ( s , m ) ;
        if s′ ∉ H then
            H[s′] ← eval ( s′ ) ;
```

rétropropagation UBFM

- MAJ des valeurs pour améliorer la prochaine descente

```
fonction backpropagate ( s ) :
    M ← nextMoves ( s ) ;
    {min, max} ← {∞ , −∞} ;
    if turn_s%2 == 0 then
        for each m ∈ M do
            s′ ← applyMove ( s , m ) ;
            if max < H[s′] then max ← H[s′] ;
        H[s] ← max ;
    else
        for each m ∈ M do
            s′ ← applyMove ( s , m ) ;
            if min > H[s′] then min ← H[s′] ;
        H[s] ← min ;
    if s == ROOT then return ;
    p ← getParent ( s ) ;
    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

```
sum ← 0 ;
for N times do
    r ← playout ( s ) ;
    sum ← r + sum ;
r ← sum/N ;
```

évaluation MC avec écart-type

```
sum ← 0 ;
sum2 ← 0 ;
for N times do
    r ← playout ( p ) ;
    sum ← r + sum ;
    sum2 ← (r * r) + sum2 ;
r_p ← sum/N ;
σ_{r_p} ← sqrt(sum2/N − (r_p * r_p)) ;
```

$$\sigma_{r_p} \leftarrow \sqrt{sum2/N - (r_p * r_p)}$$

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

```
fonction playout ( s ) :
    while not terminal ( s ) do
        M ← nextMoves ( s ) ;
        m ← random ( M ) ;
        s ← applyMove ( s, m ) ;
    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)

$\mathcal{M} \leftarrow \texttt{nextMoves}\ (\ s\ )\ ;$
if $|\mathcal{M}| == 0$ **then return** PASS ;
$\mathit{best} \leftarrow \texttt{first}\ (\ \mathcal{M}\ )\ ;\ \mathit{max} \leftarrow 0\ ;$
for $\mathit{each}\ m \in \mathcal{M}$ **do**
$\quad s' \leftarrow \texttt{applyMove}\ (\ s,\ m\ )\ ;$
$\quad w_i \leftarrow 0\ ;$
$\quad$ **for** $N\ \mathit{times}$ **do**
$\quad\quad r \leftarrow \texttt{playout}\ (\ s'\ )\ ;$
$\quad\quad$ **if** $r == \mathit{WIN}$ **then** $w_i \leftarrow w_i + 1\ ;$
$\quad$ **if** $w_i > \mathit{max}$ **then** $\{\ \mathit{best},\ \mathit{max}\ \} \leftarrow \{\ m,\ w_i\ \}\ ;$
return best ;

évaluation MCTS (2006, R. Coulom)

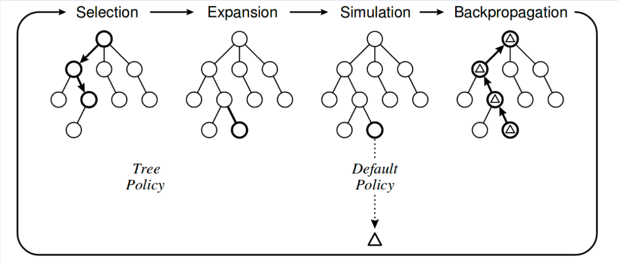


Figure extraite de A Survey of Monte Carlo Tree Search Methods par C. Browne et al (IEEE Trans. On Computational Intelligence and AI in Games, Vol. 4, No. 1, 2012).

fonction `MCTS` (s) :
- **while** *not-interrupted* **do**
  - $s' \leftarrow$ `selection` (s) ;
  - $\mathcal{H}[s'] \leftarrow \{0, 0\}$;
  - $r \leftarrow$ `playout` (s') ;
  - `backpropagate` (s', r) ;
- **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 $\{W, N\}$

```
fonction selection ( s ) :
    if terminal ( s ) then return s ;
    M ← nextMoves ( s ) ;
    {max, best} ← {−1, ∅} ;
    for each m ∈ M do
        s′ ← applyMove ( s, m ) ;
        if s′ ∉ H then return s′ ;
        new_eval ← uct ( s, s′ ) ;
        if new_eval > max then {max, best} ← {new_eval, s′} ;
    return selection ( best) ;
```

évaluation MCTS : expansion et rétropropagation

- $\mathcal{H}$ 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

```
fonction backpropagate ( s, score ) :
    if parent ( s ) == ∅ then return ;
    H[s] ← H[s] + score ;
    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, Yoshizœ 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

fonction `MCTS_with_PPA` (s) :
  while *not-interrupted* **do**
    $s' \leftarrow$ `selection` (s) ;
    $\mathcal{H}[s'] \leftarrow \{0, 0\}$;
    $\{r, seq\} \leftarrow$ `playout_with_PPA` (s', $\mathcal{P}$) ;
    if $r == WIN$ **then**
      `adapt_PPA` (s', $\mathcal{P}$, seq, $player$) ;
    else
      `adapt_PPA` (s', $\mathcal{P}$, seq, opp) ;
    `backpropagate` (s', r) ;
  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

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

fonction `adapt_PPA` (s, $\mathcal{P}$, seq , $player$) :
- $\mathcal{P}_2 \leftarrow \mathcal{P}$;
- **for** *each* $m \in seq$ **do**
  - **if** $winner == player$ **then**
    - **if** $m \notin \mathcal{P}_2$ **then** $\mathcal{P}_2[m] \leftarrow \alpha$;
    - **else** $\mathcal{P}_2[m] \leftarrow \mathcal{P}_2[m] + \alpha$;
    - $z \leftarrow \sum_{m \in s} e^{\mathcal{P}_2[m]}$;
    - **for** *each* m *from* s **do**
      - $\mathcal{P}_2[m] \leftarrow \mathcal{P}_2[m] - \frac{\alpha}{z} e^{\mathcal{P}_2[m]}$;
  - $s \leftarrow$ `applyMove` (s, m) ;
  - $player \leftarrow$ `opponent` ($player$) ;
- $\mathcal{P} \leftarrow \mathcal{P}_2$;

playout avec PPA

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

```
fonction playout_with_PPA ( s, P ) :
    seq ← ∅ ;
    while not terminal ( s ) do
        M ← nextMoves ( s ) ;
        r ← random ( Σ_{m∈M} e^{P[m]} ) ;
        for each m ∈ M do
            if r ≤ e^{P[m]} then
                s ← applyMove ( s, m ) ;
                seq ← seq + m ;
                break ;
            r ← r − e^{P[m]} ;
    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

```
fonction nested ( s, level ) :
    best_score ← −1 ;
    best_seq ← ∅ ;
    while not terminal ( s ) do
        if level = 1 then
            {m,seq} ← argmax_{m∈M} playout ( s, m ) ;
        else
            s′ ← applyMove ( s, m ) ;
            {m,seq} ← argmax_{m∈M} nested ( s′, level − 1 ) ;
        new_score ← score ( s, m ) ;
        if new_score > best_score then
            best_score ← new_score ;
            best_seq ← seq ;
        best_move ← best ( best_seq ) ;
        s ← applyMove ( s, best_move ) ;
    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 playouts $+$*adapt*)
- *i.e.* N avec $\mathcal{P} \rightarrow N$ avec $\mathcal{P}' \rightarrow$ ...

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

```
for 3 times do      // appel NRPA avec level=1
    for 3 times do      // appel NRPA avec level=0
        seq ← playout ( P ) ;
    adapt ( P ) ;
return best_seq ;
```

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

```
for 10 times do      // appel NRPA avec level=2
    for 10 times do      // appel NRPA avec level=1
        for 10 times do      // appel NRPA avec level=0
            seq ← playout ( P ) ;
        adapt ( P ) ;
    adapt ( P ) ;
return best_seq ;
```

sélection et playout avec $\mathcal{P}$

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

```
fonction select_NRPA ( P, M ) :
    sum ← 0 ;
    for each m ∈ M do
        sum ← sum + P[m] ;
    r ← random ( sum ) ;
    for each m ∈ M do
        if r ≤ P[m] then return m ;
        r ← r − P[m] ;
```

- mémoriser la séquence correspondant au playout

```
fonction playout_NRPA ( s, P ) :
    seq ← ∅ ;
    while not terminal ( s ) do
        s ← applyMove ( s, select_NRPA ( P, nextMoves ( s ) ) ) ;
        seq ← seq + s ;
    return { score ( s ), seq } ;
```

adaptation de $\mathcal{P}$ aux résultats

- adaptation exponentielle aux valeurs de $\mathcal{P}$

fonction `adapt_NRPA` ($\mathcal{P}$, *seq*) :
    $\{s, \mathcal{P}_2\} \leftarrow \{root, \mathcal{P}\}$;
    for *each* $m \in seq$ **do**
        if $m \notin \mathcal{P}_2$ **then** $\mathcal{P}_2[m] \leftarrow \alpha$;
        else $\mathcal{P}_2[m] \leftarrow \mathcal{P}_2[m] + \alpha$;
        $z \leftarrow 0$;
        for *each m from s* **do**
            $z \leftarrow z + e^{\mathcal{P}_2[m]}$;
        for *each m from s* **do**
            $\mathcal{P}_2[m] \leftarrow \mathcal{P}_2[m] - \frac{\alpha}{z} e^{\mathcal{P}_2[m]}$;
        $s \leftarrow$ `applyMove` (s, m) ;
    $\mathcal{P} \leftarrow \mathcal{P}_2$;

modification de NMCS en NRPA

- playout et adapt avec $\mathcal{P}$

```
fonction nrpa ( s, level , 𝒫 ) :
    best_score ← −1 ;
    best_seq ← ∅ ;
    while not terminal ( s ) do
        if level = 1 then
            {m,seq} ← argmax_{m∈ℳ} playout_NRPA ( s, m , 𝒫) ;
        else
            s′ ← applyMove ( s, m ) ;
            {m,seq} ← argmax_{m∈ℳ} nrpa ( s′, level − 1 ) ;
        new_score ← score ( s, m ) ;
        if new_score > best_score then
            best_score ← new_score ;
            best_seq ← seq ;
        adapt_NRPA ( 𝒫 , best_seq ) ;
        best_move ← best ( best_seq ) ;
        s ← applyMove ( s, best_move ) ;
    return score ( s );
```

résoudre un problème par recherche arborescente

1. définir des fonctions d'evaluation 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