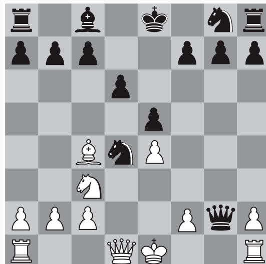
Zéróösszegű, kétszemélyes játékok

Kontextus

Játék típusa:

- kétszemélyes
- zeróösszegű (konstans összegű)
- determinisztikus
- teljes információjú

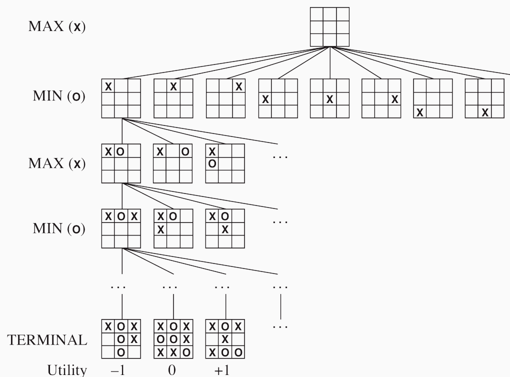


- példa: sakk, connect-4, tic-tac-toe, go, gomoku, reversi, quoridor
- ellenpélda: fogoly dilemma, quake, póker

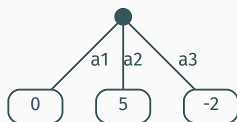
A \ B	B stays silent	B betrays
A stays silent	-1 -1	0 -3
A betrays	-3 0	-2 -2

Fogalmak

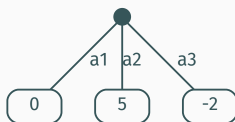
1. játékosok: MIN, MAX
2. állapot, kiinduló állapot
3. lépés, féllépés (ply)
4. játékfa
5. végteszt
6. hasznosságfüggvény
7. stratégia
8. játékfa mérete



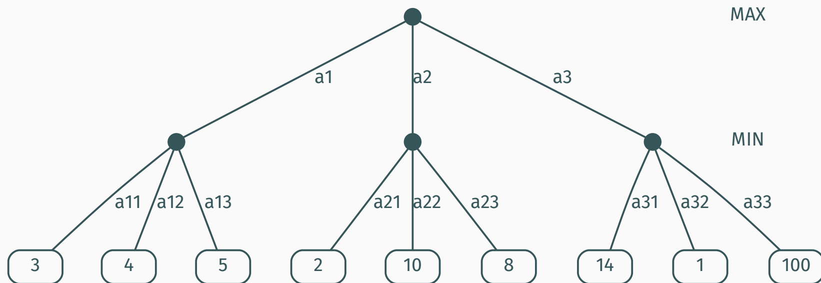
Minimax prelúdium



MAX



MIN



Optimális döntés játékokban

- egy s csomópont minimax értéke:
$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \end{cases}$$
- MAX stratégiája: azt lépést választani, amely maximális minimax értékű csomópontba vezet
- mi történik, ha MIN nem játszik optimálisan?

Minimax, algoritmus

```
int minimax_value(STATE s) {
  if (is_terminal(s)) return utility(s);
  int v = is_max_node(s) ? -INFINITY: +INFINITY;
  if (is_max_node(s)) {
    for (STATE child : children(s)) {
      v = max(v, minimax_value(child));
    }
  } else {
    for (STATE child : children(s)) {
      v = min(v, minimax_value(child));
    }
  }
  return v;
}
```

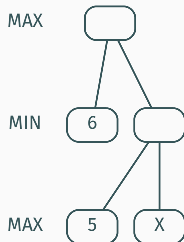
Alfa-béta nyesés

Mire értékelődnek ki az alábbi kifejezések?

- `max(6, min(5, X))`
- `min(7, max(9, Y))`

Alfa-béta nyesés

```
max(6, min(5, X))
```



- α biztos nyereség MAX számára
- β biztos nyereség MIN számára
- nyesési feltétel: $\alpha \geq \beta$

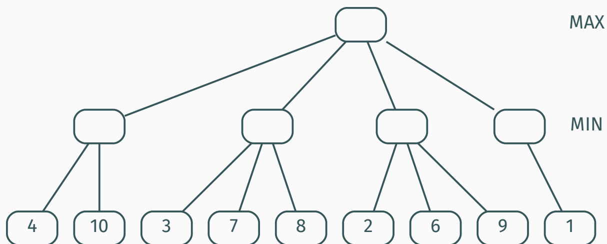
Alfa-béta nyesés (fail-hard verzió)

```
1  int alpha_beta(STATE s, int alpha, int beta) {
2    if (is_terminal(s)) return utility(s);
3    if (max_node(s)) {
4      for (STATE child : children(s)) {
5        alpha = max(alpha, alpha_beta(child, alpha, beta));
6        if (alpha >= beta) return alpha;
7      }
8      return alpha;
9    } else {
10     for (STATE child : children(s)) {
11       beta = min(beta, alpha_beta(child, alpha, beta));
12       if (alpha >= beta) return beta;
13     }
14     return beta;
15   }
16 }
```

kezdeti hívás:

```
alpha_beta(s, -INFINITY, INFINITY);
```

Alfa-béta nyesés



Nem tökéletes döntések

- a játékfa nem triviális játékoknál valós időben nem bontható le a terminális állapotokig
- **heurisztikus kiértékelő függvény**: egy állapot becsült hasznossága

Heurisztikus kiértékelő függvények

Kritériumok:

- ugyanúgy rendezze a terminális állapotokat, mint a hasznosságfüggvény
- korreláljon a nyerés esélyével
- legyen gyors

Heurisztikus kiértékelő függvény tic-tac-toe-ra

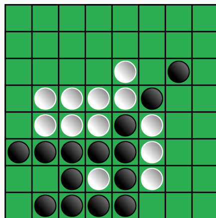
$$h(s) \equiv \begin{cases} \infty, & \text{ha } s \text{ nyerő állapot MAX számára} \\ -\infty, & \text{ha } s \text{ nyerő állapot MIN számára} \\ \mathrm{f}_3(s, MAX) - \mathrm{f}_3(s, MIN), & \text{különben} \end{cases}$$

$f_3(s,q) \equiv$ szabad hármasok q számára az s állapotban

Heurisztikus kiértékelő függvény amőbára

- fontos minták:
  - $p_1($ `*` $)$ = `*****`, $p_2($ `*` $)$ = `_****_`
  - $p_3($ `*` $)$ = `****_`, $p_4($ `*` $)$ = `_****`
  - $p_5($ `*` $)$ = `_***_`, $p_6($ `*` $)$ = `_**_`
  - stb
- rendeljünk minden mintához egy súlyzót, pl:
  $w_1 = \infty, w_2 = \infty, w_3 = 10^3, w_4 = 10^3, w_5 = 10^2, w_6 = 10^1$
- képezzük az előforduló minták súlyzott átlagainak különbségét:
  $h(s) \equiv \sum_i w_i \left(\#p_i(\text{MAX}) - \#p_i(\text{MIN})\right),$
  ahol $\#$ az előfordulások számát jelenti

Heurisztikus kiértékelő függvény Reversi-re



- elfoglalt pozíciók száma: $\#(\cdot)$
- $h(s) \equiv \#(\text{MAX}) - \#(\text{MIN})$
- további szempontok: elfoglalt sarkok száma, szabad pozíciók száma, stabil (nem felfordítható) kövek száma

Heurisztikus függvény kiértékelő sakkra

- paraszt=1, futó=3, ló=3, bástya=5, királynő=9
- gyenge (parasztok által nem védett) négyzetek száma
- parasztok pozíciója: izolált, duplázott...
- a király védve van? stb

Mélységkorlátozott alfa-béta

```
1  int alpha_beta(STATE s, int alpha, int beta, int d) {
2   if (is_terminal(s) || d==0) return heuristic_eval(s);
3   if (max_node(s)) {
4     for (STATE child : children(s)) {
5       alpha = max(alpha, alpha_beta(child, alpha, beta, d-1));
6       if (alpha >= beta) return alpha;
7     }
8     return alpha;
9   } else {
10    for (STATE child : children(s)) {
11      beta = min(beta, alpha_beta(child, alpha, beta, d-1));
12      if (alpha >= beta) return beta;
13    }
14    return beta;
15  }
16 }
```

kezdeti hívás:

```
alpha_beta(s, -INFINITY, INFINITY, MAX_DEPTH);
```

A rendelkezésre álló idő kihasználása

- iteratívan mélyülő alfa-béta keresést használjunk
- annak a mélységnek a kiértékelésével térjünk vissza, amelyet teljes szélességben sikerült bejárni az idő lejárta előtt
- amíg az ellenfél gondolkozik, (külön szálon) folytassuk az iteratívan mélyülő alfa-bétát

Összefoglaló

- zéróösszegű, kétszemélyes, determinisztikus, teljes információjú játékok
- optimális döntés: minimax döntés
- fölösleges kiértékelések elkerülése: alfa-béta nyesés
- mélységkorlátozott minimax, heurisztikus kiértékelés, iteratívan mélyülő keresés

II. rész

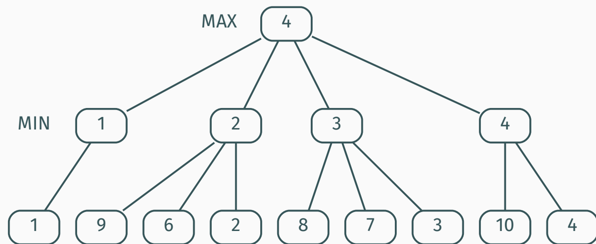
Ismételt állapotok elkerülése

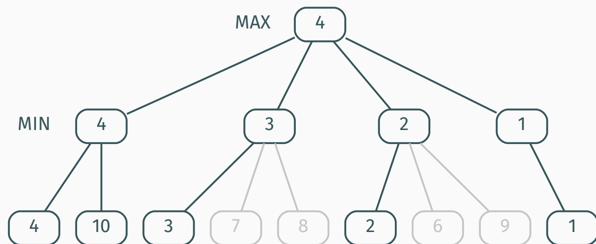
- különböző lépéssorozatok ugyanazt az állapotot —**transzpozíciót**— eredményezhetik
- transzpozíciós tábla: hasító tábla $\{\text{hash}(s_i), \text{minimax_value}(s_i)\}_i \approx$ "zárt lista"
- további elemek egy transzpozíciós csomópontban:
  - flagek: pontos minimax érték / alsó határ / felső határ
  - kiértékelési mélység

Zobrist kulcsok

- a hash inkrementális frissítését lehetővé tevő séma
- példa tic-tac-toe-ra:
  - játék előtt: $r_{i,j,k} \leftarrow$ `rand_uint()` $\forall i,j = 1\ldots3, k = 1,2$
  - üres állapot hash értéke: $h \leftarrow 0$
  - a k játékos az (i,j) pozícióra lép: $h \leftarrow \mathrm{XOR}(h, r_{i,j,k})$
  - a k játékos ellép (?!!) az (i,j) pozícióról: $h \leftarrow \mathrm{XOR}(h, r_{i,j,k})$

Csomópontok rendezése





Csomópontok rendezése (2)

- az alfa-béta nyesés érzékeny a csomópontok sorrendjére
- legjobb esetben, ugyanannyi idő alatt kb kétszer olyan mélyre tudja bontani a játékfát mint a Minimax
- a csomópontok rendezésére használjunk iteratívan mélyülő keresést

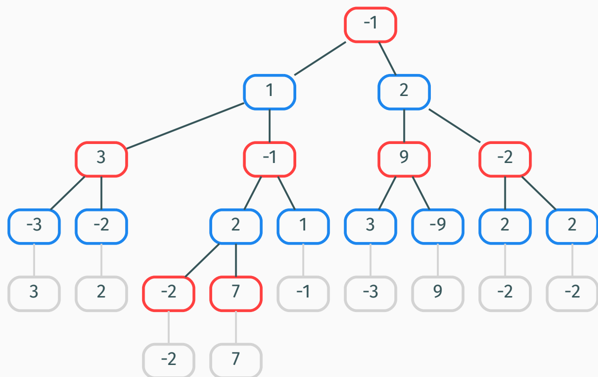
Negamax

- `min(5,6) == -max(-5,-6)`?

```
1 int negamax(STATE s, int color) {
2   if (is_terminal(s)) return color*utility(s);
3   int v = -∞;
4     for (STATE child : children(s)) {
5       v = max(v, -negamax(child, -color));
6     }
7   }
8   return v;
9 }
```

- kezdő hívás MAX-nak: `negamax(s, 1)`
- milyen értékek kerülnek a MAX csomópontokba a negamaxszal a minimaxhoz képest? és a MIN csomópontokba?

Negamax (2)



- szürke: terminális / hasznosság, kék: `color=-1`, piros: `color=1`

Alfa-béta, negamax verzió

```
1  int alphabeta(STATE s, int depth, int alpha, int beta, int color) {
2    if (depth == 0 || is_terminal(s))
3      return color * heuristic_eval(s);
4
5    foreach (STATE child : children(s) {
6      alpha = max(alpha,
7                  -alphabeta(child, depth-1, -beta, -alpha, -color));
8      if (alpha >= beta) return alpha;
9    }
10   return alpha;
11 }
```

- miért van a 7. sorban az `alpha` és a `beta` ”felcserélve”?

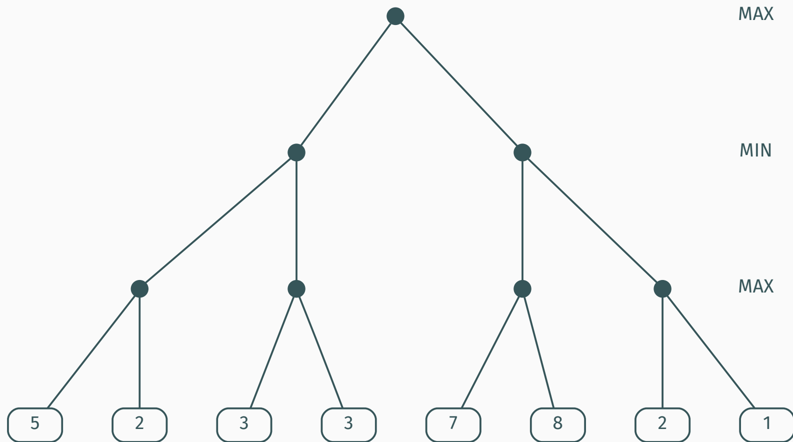
Fail-soft alpha-beta

```
int alphabeta(STATE s, int depth, int alpha, int beta, int color) {
  if (depth == 0 || is_terminal(s))
    return color * heuristic_eval(s);

  int score = −∞;

  foreach (STATE child : children(s) {
    score = max(score,
                -alphabeta(child, depth-1, -beta, -alpha, -color));

    alpha = max(alpha, score);
    if (alpha >= beta) break;
  }
  return score;
}
```

- alphabeta(s, $-\infty$, $+\infty$) = minimax(s)
- alphabeta(s, α, β) = minimax(s), ha $\alpha <$ minimax(s) $< \beta$
- alphabeta(s, α, β) $\leq \alpha$, ha minimax(s) $\leq \alpha$
- alphabeta(s, α, β) $\geq \beta$, ha minimax(s) $\geq \beta$

Fail-soft alpha-beta (2)

- minimax(s) = 3
- alphabeta(s, alpha=6, beta=8) = 5 (fail-soft verzió!)



Az ablak szűkítése – aspiration search

- az `alphabeta(state, -∞, ∞)` csak a `state` első gyerekének kiértékelése után tud csomópontokat levágni
- kisebb ablak több levágást eredményez, de rizikós

```
1  int aspiration_search(STATE state, int psv, int window) {
2    int alpha = psv - window;
3    int beta = psv + window
4    int v;
5    for (;;) {
6      v = alphabeta(state, alpha, beta);  //fail-soft verzió
7      if (v < alpha) alpha = −∞;      //vagy: alpha -= window
8      else if (v > beta) beta = ∞;    //vagy: beta  += window
9      else break;
10   }
11   return v;
12 }
```

- `psv` — a `state` egy korábbi, kisebb mélységű kiértékelésből származó minimax értéke

Null ablak – null window search

- extrém eset, **null ablak**[1]: $\beta = \alpha + 1$
- null ablakkal gyorsan ki tudjuk értékelni, hogy egy S állapot minimax értéke kisebb, vagy nagyobb mint α
- ha $v(S)$ egy S állapot kiértékelése null ablakkal és...
  - $v(S) \leq \alpha \Rightarrow v(S)$ nem jobb mint α, nem kell tennünk semmit (**fail low**)
  - $v(S) > \alpha \Rightarrow v(S)$ jobb, mint α, de nem tudjuk mennyivel, így S-et egy nagyobb ablakkal újra meg kell vizsgálni (**fail high**)

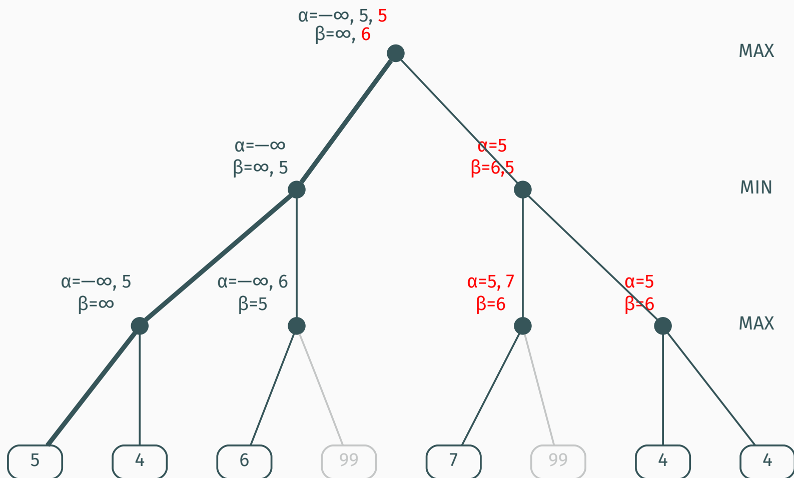
[1] egész értékű kiértékelő függvényt feltételezünk

Principal variation (PV)

- az a lépéssorozat, amely mentén mindkét játékos optimálisan játszik
- a PV-n minden csomópontnak ugyanaz a minimax értéke
- bár több PV is létezhet, a **legelsőre** fogunk így hivatkozni

Principal variation + null window search

- az első gyerek kiértékelése után null ablakos keresést végzünk a második gyereken (5,6)-al



Principal variation search / NegaScout

- ötlet:
  - feltételezi, az első gyerek a PV-ben van
  - majd ellenőrzi a feltételezést egy null-ablakkal
  - ha a feltételezés elbukik, normál alfa-bétával folytatja
- akkor hatékony, ha a gyerekek jól vannak rendezve

Principal variation search (3)

```
1  int pvs(STATE s, int depth, int a, int b, int color) {
2    if (depth == 0 || is_terminal(s))
3        return color * heuristic_eval(s);
4
5    foreach (STATE child : order(children(s)) {
6        if (is_first_child(child)
7            score = -pvs(child, depth-1, -b, -a, -color);
8        else {
9            score = -pvs(child, depth-1, -a-1, -a, -color);
10           if (a < score < b) {
11              score = -pvs(child, depth-1, -b, -score, -color);
12           }
13           a = max(a, score);
14           if (a >= b)  return a;
15       }
16   }
17   return a;
18 }
```

Összefoglaló

- egyszerűsített implementáció: negamax
- ismételt állapotok elkerülése: transzpozíciós tábla
- alfabéta optimizáció: csomópontok rendezése, az ablak szűkítése, principal variation search

Könyvészet, ábrák

- Russel & Norvig: Artificial Intelligence, a Modern Approach (2. és 3. kiadás)
- Csákány, Dr. Vajda: Játékok számítógéppel
- J. Pearl: SCOUT. A simple game-searching algorithm With proven optimal properties
- A. Reinefeld: An improvement of the scout tree search algorithm