Laborator 06: Minimax

Responsabili:

- Radu Nichita (2021) [mailto:radunichita99@gmail.com]
- Darius-Florentin Neațu (2017-2021) [mailto:neatudarius@gmail.com]
- Cristian Olaru (2021) [mailto:cristianolaru99@gmail.com]
- Miruna-Elena Banu (2021) [mailto:mirunaelena.banu@gmail.com]
- Mara-Ioana Nicolae (2021) [mailto:maraioana9967@gmail.com]
- Ștefan Popa (2018-2020) [mailto:stefanpopa2209@gmail.com]

Obiective laborator

- Însușirea unor cunoștințe de bază despre **teoria jocurilor** precum și despre **jocurile de tip joc de sumă zero (suma nulă, zero-sum games)**
- Însușirea unor cunostințe elementare despre **algoritmii necesari** rezolvării unor probleme de joc de sumă zero (zero-sum game).
- Găsirea și compararea diverselor euristici pentru jocurile cunoscute, precum **șah**.

Precizări inițiale

Un curs foarte bine explicat este pe canalul de YouTube de la MIT. Vă sfătuim să vizionați integral Search: Games, Minimax, and Alpha-Beta [https://youtu.be/STjW3eH0Cik] înainte să parcurgeți materialul de pe ocw.

Introducere

Jocuri de sumă 0 (wikipedia.org/zero-sum-game [https://en.wikipedia.org/wiki/Zero-sum_game]) sunt jocurile care de obicei se joacă în doi jucători și în care o mutare bună a unui jucător este în dezavantajul celuilalt jucător în mod direct. Aplicațiile din cadrul acestui laborator sunt în orice joc de sumă 0: șah, table, X și 0, dame, go etc. Orice joc în care se poate acorda un scor / punctaj pentru anumite evenimente - de exemplu la șah pentru capturarea unor piese sau la X și 0 pentru plasarea unui X pe o anumite poziție).

Algoritmul Minimax reprezintă una din cele mai cunoscute strategii pentru a juca jocurile de sumă 0. Minimax este un algoritm ce permite minimizarea pierderii de puncte într-o mutare următoare a jucătorului advers. Concret, pentru o alegere într-un joc oarecare se preferă cea care aduce un risc minim dintre viitoarele mutări bune ale adversarului.

De asemenea, algoritmul Minimax este folosit în diverse domenii precum teoria jocurilor (Game Theory), teoria jocurilor combinatorice (Combinatorial Game Theory - CGT), teoria deciziei (Decision Theory) și statistică.

Descrierea algoritmului

Minimax

Ideea pe care se bazează algoritmul este că jucătorii adoptă următoarele strategii:

- Jucătorul 1 (**Maxi**) va încerca mereu să-și **maximizeze** propriul câștig prin mutarea pe care o are de făcut;
- Jucătorul 2 (**Mini**) va încerca mereu să **minimizeze** câștigul jucătorului 1 la fiecare mutare.

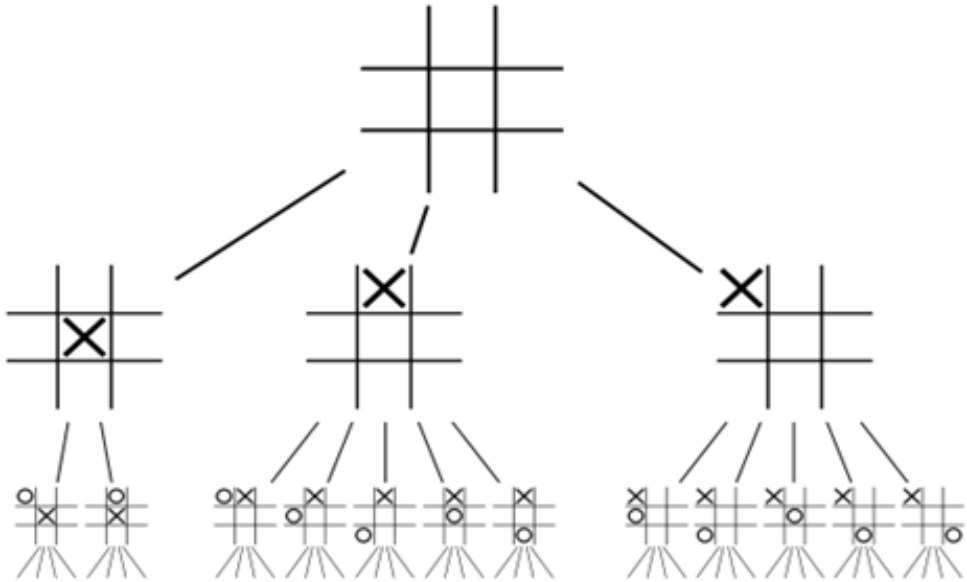
De ce merge o astfel de abordare? După cum se preciza la început, discuția se axează pe jocuri de sumă zero (zero-sum game). Acest lucru garantează, printre altele, că orice câștig al Jucătorului 1 este egal cu modulul sumei pierdute de Jucătorul 2. Cu alte cuvinte, cât pierde Jucătorul 2, atât câștigă Jucător 1. Invers, cât pierde Jucător 1, atât câștigă Jucator 2. Sau:

$$Câștig_{Jucător_1} = |Pierdere_{Jucător_2}|$$

$$Câștig_{Jucător_2} = |Pierdere_{Jucător_1}|$$

Reprezentarea spațiului soluțiilor

În general spațiul soluțiilor pentru un joc în doi de tip zero-sum se reprezintă ca un **arbore**, fiecărui nod fiindu-i asociată o stare a jocului în desfășurare (game state). De exemplul, putem considera jocul de X și O ce are următorul arbore (parțial) de soluții. Acesta corespunde primelor mutări ale lui X, respectiv O:



Metodele de reprezentare a arborelui variază în funcție de paradigma de programare aleasă, de limbaj, precum și de gradul de optimizare avut în vedere.

Implementare

Având noțiunile de bază asupra strategiei celor doi jucători, precum și a reprezentării spațiului soluțiilor problemei, o primă implementare a algoritmului Minimax ar folosi două funcții maxi() și mini(), care ambele calculează cel mai bun scor pe care îl poate obține jucătorul menționat. Intuitiv, cele 2 funcții au implementare aproape identică, astfel că dorim o organizam a codului fără porțiuni duplicate. De aceea singura variantă de implementare pe care o recomandăm, este varianta Negamax:

```
// compute the state score for current player
int evaluate(state, player);

// apply the move on the current state: old_state -> new_state
void apply_move(state, move);
// undo the move and restore previous state: new_state -> old_state
```

```
void undo_move(state, move);

// check if any player won the game
bool game_over(state);

// return the opponent for current player
Player get_opponent(player);

// compute the best score that player can get,
// considering that the opponent also has an optimal strategy
int negamax(State& state, int depth, Player player) {
    // STEP 1: game over or maximum recursion depth was reached
    if (game_over() || depth == 0) {
       return evaluate(state, player);
    }

    // STEP 2: generate all possible moves for player
    all_moves = get_all_moves(state, player);


    // STEP 3: try to apply each move - compute best score
    int best_score = -oo;
    for (move : all_moves) {
        // STEP 3.1: do move
        apply_move(state, move);

        // STEP 3.2: play for the opponent
        int score = -negamax(state, depth - 1, get_opponent(player));
        // opponent allows player to obtain this score if player will do current move.
        // player chooses this move only if it has a better score.
        if (score > best_score) {
            best_score = score;

            // [optional]: the best move can be saved
            // best_move = move;
        }

        // STEP 3.3: undo move
        undo_move(state, move);
    }

    // STEP 4: return best allowed score
    // [optional] also return the best move
    return best_score;
}
```

De ce este nevoie de utilizarea unei adâncimi maxime?

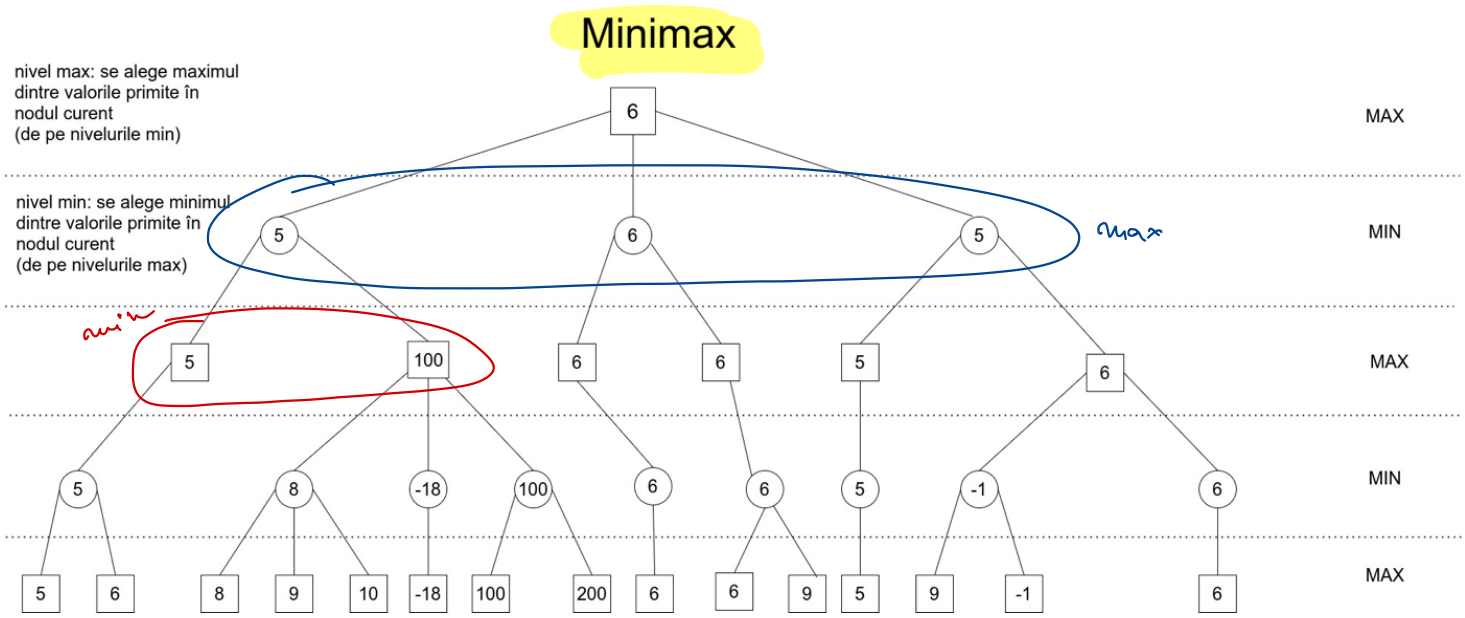
Datorită **spațiului de soluții mare**, de multe ori copleșitor ca volum de date de analizat, o inspectare completă a acestuia nu este fezabilă și devine impracticabilă din punctul de vedere al timpului consumat sau chiar a memoriei alocate (se vor discuta aceste aspecte în paragraful legat de complexitate).

Astfel, de cele mai multe ori este preferată o abordare care parcurge arborele numai până la o anumită **adâncime maximă („depth”)**. Aceasta abordare permite examinarea arborelui destul de mult pentru a putea lua decizii minimalist coerente in desfășurarea jocului.

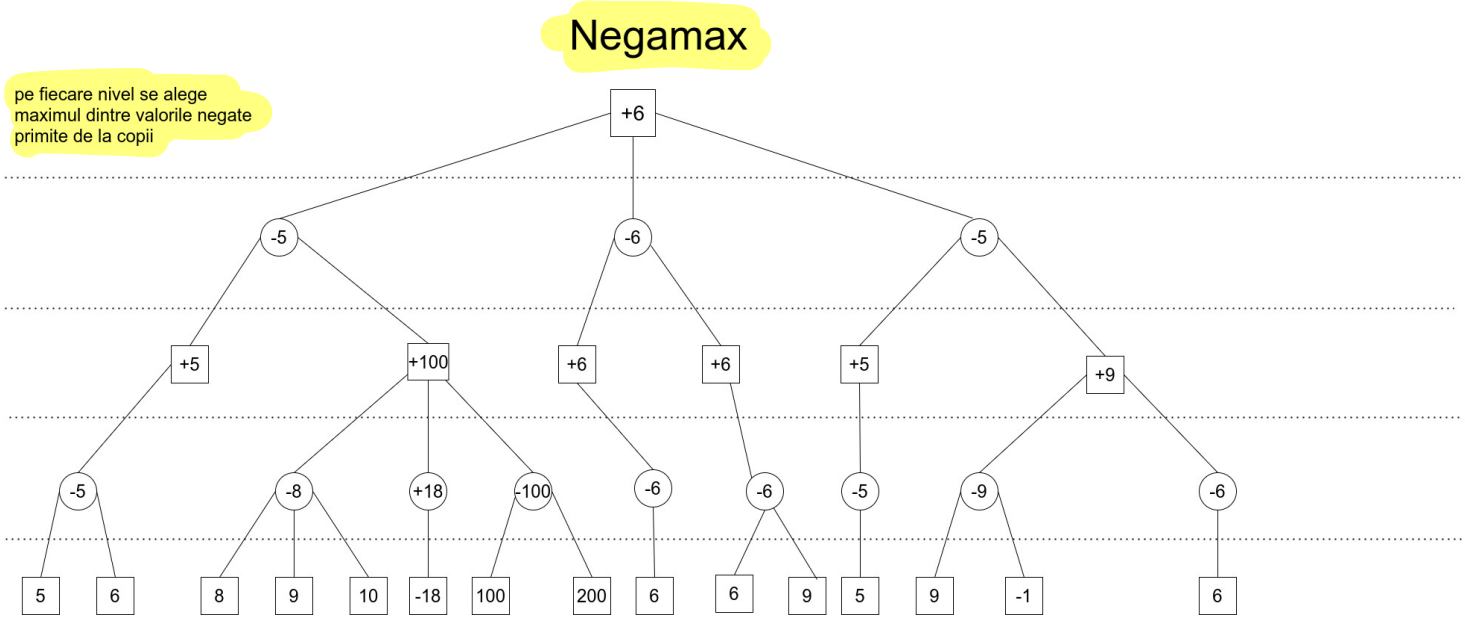
Totuși, **dezavantajul major** este că pe termen lung se poate dovedi ca decizia luată la adâncimea depth nu este global favorabilă jucătorului în cauză (s-a ales o valoare maxim local, iar dacă s-ar fi continuat în arborele de explorare s-ar fi constatat că este o decizie ce avantajează celălălt jucător).

De asemenea, se observă recursivitatea indirectă. Prin convenție acceptăm ca **începutul algoritmului** să fie cu jucătorul max. Astfel, se analizează succesiv diferite stări ale jocului din punctul de vedere al celor doi jucatori până la adâncimea depth. Rezultatul întors este scorul final al mișcării celei mai bune pentru un jucător din perspectiva următoarelor depth mutări în joc.

Exemplu grafic



Se observă că arborele se completează prin parcugere în adâncime (stanga-dreapta, sus-jos). * Pe nivel Max/Maxi se alege maximul dintre valorile primite de la copii. * Pe nivel Min/Mini se alege minimul dintre valorile primite de la copii. Obervație: Toate nodurile sunt evaluate.



Aceleași explicații ca la Minimax, cu câteva observații adiționale: * NU mai este nevoie să numerotăm nivelele. Pe fiecare nivel, jucătorul curent alege maximul dintre scorurile negate ale adeversarului (venite din copii).

Optimizări

Alpha-beta pruning

Până acum s-a discutat despre algoritmii Minimax / Negamax. Aceștia sunt algoritmi exhaustivi (**exhausting search algorithms**). Cu alte cuvinte, ei găsesc soluția optima examinând întreg spațiul de soluții al problemei. Acest mod de abordare este extrem de ineficient în ceea ce privește efortul de calcul necesar, mai ales considerând că extrem de multe stări de joc inutile sunt explorate (este vorba de acele stări care nu pot fi atinse datorită încălcării principului că fiecare jucător joacă optim la fiecare rundă).

O îmbunătățire substanțială a Minimax/Negamax este **Alpha-beta pruning (tăiere alfa-beta)**. Acest algoritm încearcă să optimizeze Minimax/Negamax profitând de o observație importantă: **pe parcursul examinării arborelui de soluții se pot elimina întregi subarbori, corespunzători unei mișcări m, dacă pe parcursul**

analizei găsim că mișcarea m este mai slabă calitativ decât cea mai bună mișcare curentă.

Astfel, considerăm că pornim cu o primă mișcare M1. După ce analizăm această mișcare în totalitate și îi atribuim un scor, continuăm să analizăm mișcarea M2. Dacă în analiza ulterioară găsim că adversarul are cel puțin o mișcare care transformă M2 într-o mișcare mai slabă decât M1 atunci orice alte variante ce corespund mișcării M2 (subarbori) nu mai trebuie analizate.

> De ce? Pentru că știm că există **cel puțin** o variantă în care adversarul obține un câștig mai bun decât dacă am fi jucat mișcarea M1.
>
> Nu conteaza exact cât de slabă poate fi mișcarea M2 față de M1. O analiză amănunțită ar putea releva că poate fi și mai slabă decât am constatat inițial, însă acest lucru este irelevant.
>
> De ce însă ignorăm întregi subarbori și mișcări potențial bune numai pentru o mișcare slabă găsită? Pentru că, în conformitate cu **principiul de maximizare al câștigului** folosit de fiecare jucator, adversarul va alege exact acea mișcare ce îi va da un câștig maximal. Dacă există o variantă și mai bună pentru el este irelevant, deoarece noi suntem interesați dacă cea mai slabă mișcare bună a lui este mai bună decât mișcarea noastră curent analizată.

Un video cu un exemplu detaliat și foarte bine explicat se găsește în tutorialul recomandat de pe YouTube (de la minutul 21:30 la 30:30).

Implementare

În continuare prezentăm o implementare conceptuală a Alpha-beta pentru varianta Negamax:

```
// compute the state score for current player
int evaluate(state, player);

// apply the move on the current state: old_state -> new_state
void apply_move(state, move);
// undo the move and restore previous state: new_state -> old_state
void undo_move(state, move);

// check if any player won the game
bool game_over(state);

// return the opponent for current player
Player get_opponent(player);

// compute the best score that player can get,
// considering that the opponent also has an optimal strategy
int alphabeta_negamax(State& state, int depth, Player player, int alpha, int beta) {
    // STEP 1: game over or maximum recursion depth was reached
    if (game_over() || depth == 0) {
       return evaluate(state, player);
    }

    // STEP 2: generate all possible moves for player
    // Note: sort moves descending by score (if possible) for maximizing the number of cut-off actions
    // (or generete the moves already sorted by a custom criterion)
    all_moves = get_all_moves(state, player);


    // STEP 3: try to apply each move - compute best score
    int best_score = -oo;
    for (move : all_moves) {
        // STEP 3.1: do move
        apply_move(state, move);

        // STEP 3.2: play for the opponent
        int score = -alphabeta_negamax(state, depth - 1, get_opponent(player), -beta, -alpha);
```

```
        // opponent allows player to obtain this score if player will do current move.
        // player chooses this move only if it has a better score.
        if (score > best_score) {
            best_score = score;

            // [optional]: the best move can be saved
            // best_move = move;
        }

        // STEP 3.3: update alpha (found a better move?)
        if (best_score > alpha) {
            alpha = best_score;
        }

        // STEP 3.4: cut-off
        // * already found the best possible score (alpha == beta)
        // OR
        // * on this branch we can obtain a score (alpha) better than the
        // maximum allowed score by the opponent => drop the branch because
        // opponent also plays optimal
        if (alpha >= beta) {
            break
        }

        // STEP 3.4: undo move
        undo_move(state, move);
    }

    // STEP 4: return best allowed score
    // [optional] also return the best move
    return best_score;
}
```

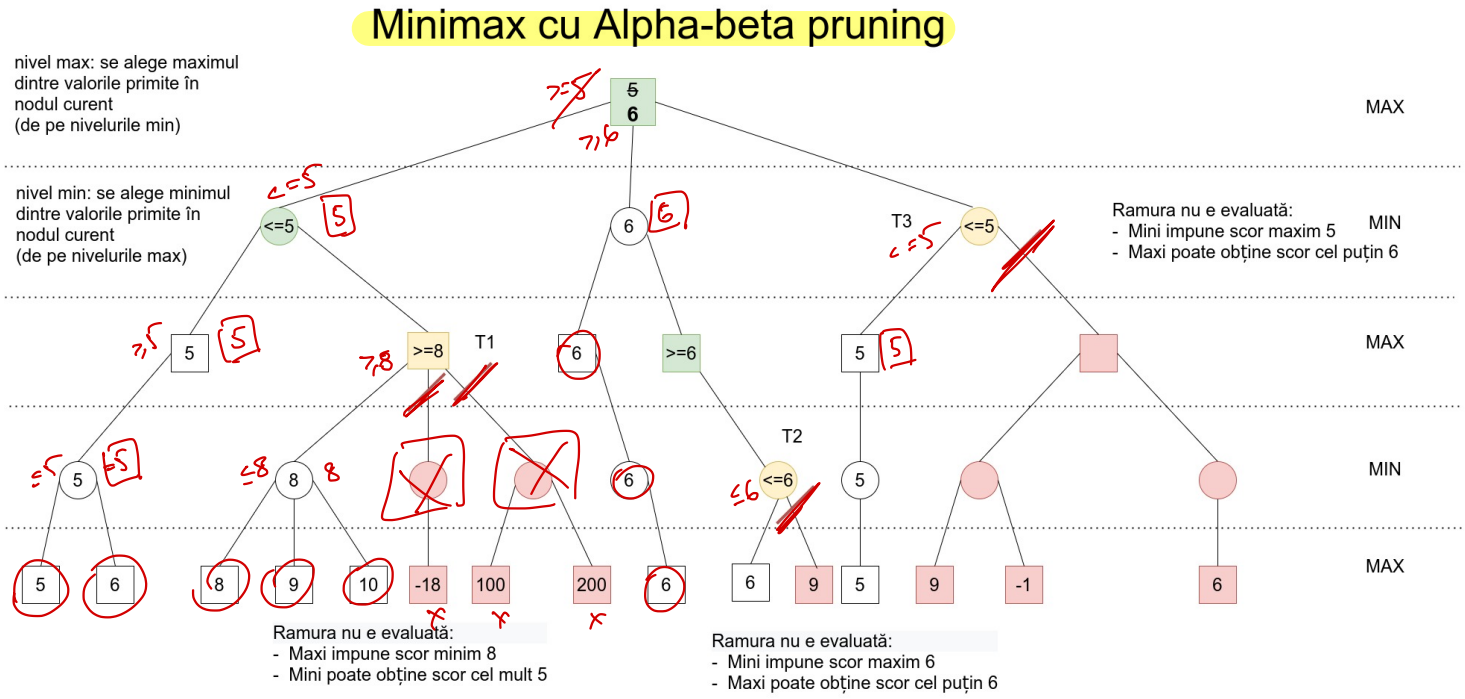
O observație foarte importantă se poate face analizând **modul de funcționare** al acestui algoritm: este extrem de importantă **ordonarea mișcărilor după valoarea câștigului**.

În **cazul ideal** în care cea mai bună mișcare a jucătorului curent este analizată prima, toate celelalte mișcări, fiind mai slabe, vor fi eliminate din căutare timpuriu.

În **cazul cel mai defavorabil** însă, în care mișcările sunt ordonate crescător după câștigul furnizat, Alpha-beta are aceeași complexitate cu Minimax/ Negamax, îmbunătățirea fiind nulă.

În **medie** se constată o eficiență sporită in practică pentru Alpha-beta.

Exemplu grafic



Iterative deepening

Pe scurt, iterative deepening reprezintă o strategie de optimizare a timpului prin căutarea progresivă în adâncime. Uneori, există posibilitatea pentru anumite stări să nu fie explorate complet, iar în acest caz rezultatul este aproximat printr-o euristică. Tehnica presupune salvarea în memorie a rezultatelor neexplorate complet (toate sau cele mai "promițătoare" stări), iar la următoarea rundă se vor refolosi o parte din aceste rezultate (cele care mai sunt posibile din noul moment al jocului). Un video cu un exemplu explicat se găsește pe link-ul youtube.com/iterative_deepening [https://www.youtube.com/watch?v=Y85ECk_H3h4&ab_channel=JohnLevine]

Complexitate

Pentru a vedea complexitatea algoritmilor prezentați anterior, se vor introduce câteva noțiuni:

- **branch factor** : **b** = **numărul mediu de ramificări** pe care le are un nod neterminal (care nu e frunză) din **arborele de soluții**
- **depth** : **d** = **adâncimea maximă** până la care se face căutarea în arborele de soluții
  - orice nod de adâncime d va fi considerat terminal

Un arbore cu un branching factor **b**, care va fi examinat până la un nivel **d** va furniza b^d noduri frunze ce vor trebui procesate (ex. calculăm scorul pentru acele noduri).

Nivelurile sunt notate cu $0, 1, 2, \ldots, d$

- nivel 0: 1 nod (radacină)
- nivel 1: b noduri
- nivel 2: b^2 noduri
- nivel 3: b^3 noduri
- ...
- nivel d: b^d noduri

- **minimax/negamax**
  - Un algoritm **Minimax / Negamax** clasic, care analizează toate stările posibile, va avea complexitatea $O(b^d)$ - deci exponențială.

- **alpha-beta**
  - Cât de bun este însa alpha-beta fața de un Minimax / Negamax naiv? Dupa cum s-a menționat anterior, în funcție de ordonarea mișcărilor ce vor fi evaluate putem avea un caz cel mai favorabil și un caz cel mai defavorabil.
  - **best case** : mișcările sunt ordonate descrescător după câștig (deci ordonate optim), rezultă o complexitate
    - $O(b * 1 * b * 1 * b * 1 ...de\ d\ ori... b * 1)$ pentru d par
    - $O(b * 1 * b * 1 * b * 1 ...de\ d\ ori... b)$ pentru d impar
    - restrângând ambele expresii rezultă o complexitate $O(b^{\frac{d}{2}}) = O(\sqrt{b^d})$
    - prin urmare, într-un caz ideal, algoritmul Alpha-beta poate explora de 2 ori mai multe nivele în arborele de soluții în același timp față de un algoritm Minimax/Negamax naiv.
  - **worst case**: mișcările sunt ordonate crescător după câștigul furnizat unui jucător, astfel fiind necesară o examinare a tuturor nodurilor pentru găsirea celei mai bune mișcări.
    - în acest caz complexitatea devine egală cu cea a unui algoritm Minimax / Negamax naiv.

Exemple

Dintre cele mai importante jocuri în care putem aplica direct strategia minimax, menționăm:

- X și 0 [https://en.wikipedia.org/wiki/Tic-tac-toe]
  - joc foarte simplu/ușor (spațiul stărilor este mic).
  - Prin urmare tot arborele de soluții poate fi generat și explorat într-un timp foarte scurt.
- sah [https://en.wikipedia.org/wiki/Chess]
  - joc foarte greu (spațiul stărilor este foarte mare)
  - minimax/negamax simplu poate merge până la $d = 7$ (nu reușea să bată campionul mondial la șah - campion uman)
  - alpha-beta poate merge până la $d = 14$
  - **Deep Blue [https://en.wikipedia.org/wiki/Deep_Blue_(chess_computer)]** a fost implementarea unui bot cu minimax și alpha-beta care a bătut în 1997 campionul mondial la șah (Gary Kasparov).
- Ultimate tic-tac-toe [https://en.wikipedia.org/wiki/Ultimate_tic-tac-toe]
  - varianta mult mai grea de X și 0 (spațiul stărilor foarte mare)
  - s-a dat la proiect PA 2016 :D
- Go [https://en.wikipedia.org/wiki/Go_(game)]
  - soluțiile se bazează pe Monte Carlo Tree Search (nu pe minimax)
  - AlphaGo [https://en.wikipedia.org/wiki/AlphaGo] este botul cel mai bun pe tabla de 19×19

Exerciții

În lab06 nu există exerciții propriu-zise.

Task-uri:

* Parcugeți teoria din acest laborator împreună cu asistentul.

- Trebuie să înțelegeți și să comparați **Negamax** vs **Negamax cu Alpha-beta pruning**.
- Indicație: Desenați exemplul din videoclipul de pe YouTube din sectiunea de **Precizări inițiale**.

* Discutați și comparați euristici pentru:

- șah
- table

Referințe

[0] wikipedia.org/Minimax [http://en.wikipedia.org/wiki/Minimax]

[1] wikipedia.org/Negamax [http://en.wikipedia.org/wiki/Negamax]

[2] wikipedia.org/Alpa-beta_pruning [http://en.wikipedia.org/wiki/Alpha-beta_pruning]

[4] wikipedia.org/Monte_Carlo_tree_search [https://en.wikipedia.org/wiki/Monte_Carlo_tree_search]

[5] wikipedia.org/MTD-f [https://en.wikipedia.org/wiki/MTD-f]]

[6] chessprogramming.org/Null_Window [https://www.chessprogramming.org/Null_Window]

[7] chessprogramming.org/Principal_Variation [https://www.chessprogramming.org/Principal_Variation]

[8] chessprogramming.org/Iterative_deepening [https://www.chessprogramming.org/Iterative_Deepening]

[9] Monte Carlo Tree Search [https://www.aaai.org/Papers/AIIDE/2008/AIIDE08-036.pdf]

pa/laboratoare/laborator-06.txt · Last modified: 2021/04/16 02:54 by radu.nichita