Laborator 5: Minimax

Obiective laborator

- Insusirea unor cunostinte de baza despre teoria jocurilor precum si despre jocurile de tip zero-sum;
- Insusirea abilitatii de rezolvare a problemelor ce presupun cunoasterea si exploatarea conceptului de zero-sum;
- Insusirea unor cunostinte elementare despre algoritmii necesari rezolvarii unor probleme de tip zero-sum.

Importanţă – aplicaţii practice

Algoritmul Minimax si variantele sale imbunatatite (Negamax, Alpha-Beta, Negascout etc) sunt folosite in diverse domenii precum teoria jocurilor (Game Theory), teoria jocurilor combinatorice (Combinatorial Game Theory – CGT), teoria deciziei (Decision Theory) si statistica. Astfel, diferite variante ale algoritmului sunt necesare in proiectarea si implementarea de aplicatii legate de inteligenta artificiala, economie, dar si in domenii precum stiinte politice sau biologie.

Descrierea problemei și a rezolvărilor

Algoritmii Minimax permit abordarea unor probleme ce tin de teoria jocurilor combinatorice. CGT este o ramura a matematicii ce se ocupa cu studierea jocurilor in doi (two-player games), in care participantii isi modifica rand pe rand pozitiile in diferite moduri, prestabilite de regulile jocului, pentru a indeplini una sau mai multe conditii de castig. Exemple de astfel de jocuri sunt: sah, go, dame (checkers), X si O (tic-tac-toe) etc. CGT nu studiaza jocuri ce presupun implicarea unui element aleator (sansa) in derularea jocului precum poker, blackjack, zaruri etc. Astfel decizia abordarii unor probleme rezolvabile prin metode de tip Minimax se datoreaza in principal simplitatii atat conceptuale, cat si raportat la implementarea propriu-zisa.

Minimax

Strategia pe care se bazeaza ideea algoritmului este ca jucatorii implicati adopta urmatoarele strategii:

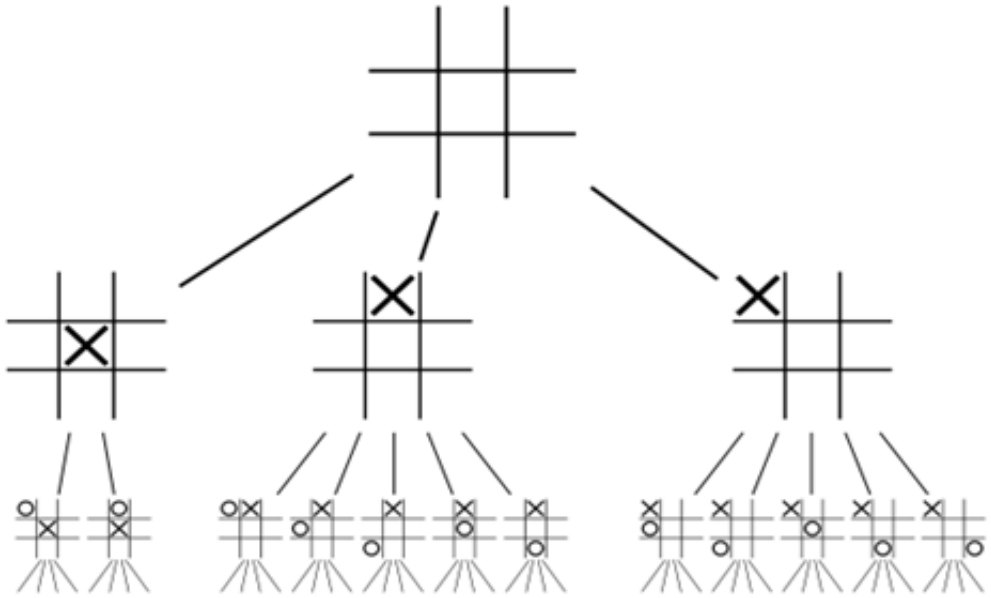
- Jucatorul 1 (maxi) va incerca mereu sa-si maximizeze propriul castig prin mutarea pe care o are de facut;
- Jucatorul 2 (mini) va incerca mereu sa minimizeze castigul jucatorului 1 la fiecare mutare.

De ce merge o astfel de abordare? Dupa cum se preciza la inceput, discutia se axeaza pe jocuri zero-sum. Acest lucru garanteaza, printre altele, ca orice castig al Jucatorului 1 este egal cu modulul sumei pierdute de Jucatorul 2. Cu alte cuvinte cat pierde Jucator 2, atat castiga Jucator 1. Invers, cat pierde Jucator 1, atat castiga Jucator 2. Sau

Win_Player_1 = | Loss_Player_2 | si | Loss_Player_1 | = Win_Player_2

Reprezentarii spatiului solutiilor

In general spatiul solutiilor pentru un joc in doi de tip zero-sum se reprezinta ca un arbore, fiecarui nod fiindu-i asociata o stare a jocului in desfasurare (game state). Pentru exemplul nostru de X si O putem considera urmatorul arbore (partial) de solutii, ce corespunde primelor mutari ale lui X, respectiv O:



Metodele de reprezentare a arborelui variaza in functie de paradigma de programare aleasa, de limbaj, precum si de gradul de optimizare avut in vedere.

Avand notiunile de baza asupra strategiei celor doi jucatori, precum si a reprezentarii spatiului solutiilor problemei, putem formula o prima varianta a algoritmului Minimax:

```
        int maxi( int depth )
        {
                if ( depth == 0 ) return evaluate();
                int max = -oo;
                for ( all moves)
                {
                        score = mini( depth - 1 );
                        if( score > max )
                                max = score;
                }
                return max;
        }

        int mini( int depth )
        {
                if ( depth == 0 ) return -evaluate();
                int min = +oo;
                for ( all moves)
                {
                        score = maxi( depth - 1 );
                        if( score < min )
                                min = score;
                }
                return min;
        }
```

Argumentarea utilizarii unei adancimi maxime

Datorita spatiului de solutii mare, de multe ori coplesitor ca volum, o inspectare completa a acestuia nu este fezabila si devine impracticabila din punctul de vedere al timpului consumat sau chiar a memoriei alocate (se vor discuta aceste aspecte in paragraful legat de complexitate). Astfel, de cele mai multe ori este preferata o abordare care parcurge arborele numai pana la o anumita adancime maxima („depth"). Aceasta abordare permite examinarea arborelui destul de mult pentru a putea lua decizii minimalist coerente in desfasurarea jocului. Totusi, dezavantajul major este ca pe termen lung se poate dovedi ca decizia luata la adancimea depth nu este global favorabila jucatorului in cauza. De asemenea, se observa recursivitatea indirecta. Prin conventie acceptam ca inceputul algoritmului sa fie cu functia maxi. Astfel, se analizeaza succesiv diferite stari ale jocului din punctul de vedere al celor doi jucatori pana la adancimea depth. Rezultatul intors este scorul final al miscarii celei mai bune.

Negamax

Negamax este o varianta a minimax, ce se bazeaza pe urmatoarea observatie: fiind intr-un joc zero-sum in care castigul unui jucator este egal cu modulul sumei pierdute de celalalt jucator si invers, putem deriva remarca ca fiecare jucator incearca sa-si maximizeze propriul castig la fiecare pas. Intr-adevar putem spune ca jucatorul mini incearca de fapt sa maximizeze in modul suma pierduta de maxi. Astfel putem formula urmatoarea implementare ce profita de observatia de mai sus (Nota: putem exprima aceasta observatie si pe baza formulei **max(a, b) = -min(-a, -b))**:

```
        int negaMax( int depth )
        {
                if ( depth == 0 ) return evaluate();
                int max = -oo;
                for ( all moves)
                {
                        score = -negaMax( depth - 1 );
                        if( score > max )
                                max = score;
                }
                return max;
        }
```

Se observa direct avantajele acestei formulari fata de Minimax-ul standard prezentat anterior:

- Claritatea sporita a codului
- Eleganta implementarii
- Usurinta in intretinere si extindere a functionalitatii

Din punctul de vedere al complexitatii temporale, Negamax nu difera absolut deloc de Minimax (ambele examineaza acelasi numar de stari in arborele de solutii). Putem concluziona ca este de preferat o implementare ce foloseste negamax fata de una bazata pe minimax in rezolvarea unor probleme ce tin de aceasta tehnica.

Alpha-beta pruning

Pana acum s-a discutat despre algoritmii Minimax si Negamax. Acestia sunt algoritmi exhaustivi (exhausting search algorithms). Cu alte cuvinte, ei gasesc solutia optima examinand intreg spatiul de solutii al problemei. Acest mod de abordare este extrem de ineficient in ceea ce priveste efortul de calcul necesar, mai ales considerand ca extrem de multe stari de joc inutile sunt explorate (este vorba de acele stari care nu pot fi atinse datorita incalcarii principiului de maximizare a castigului la fiecare runda).

O imbunatatire substantiala a minimax/negamax este Alpha-beta pruning. Acest algoritm incearca sa optimizeze mini/nega-max profitand de o observatie importanta: pe parcursul examinarii arborelui de

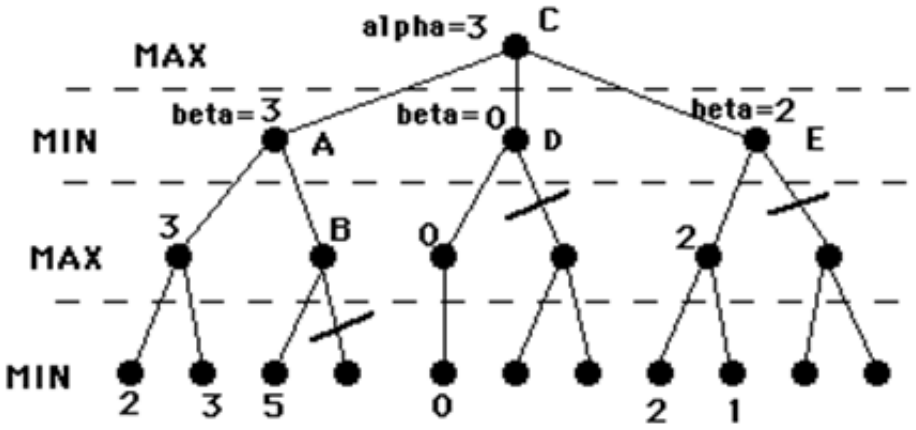
solutii se pot elimina intregi subarbori, corespunzatori unei miscari m, daca pe parcursul analizei gasim ca miscarea m este mai slaba calitativ decat cea mai buna miscare curenta.

Astfel, consideram ca pornim cu o prima miscare M1. Dupa ce analizam aceasta miscare in totalitate si ii atribuim un scor, continuam sa analizam miscarea M2. Daca in analiza ulterioara gasim ca adversarul are cel putin o miscare care transforma M2 intr-o miscare mai slaba decat M1 atunci orice alte variante ce corespund miscarii M2 (subarbori) nu mai trebuie analizate.

De ce? Pentru ca stim ca exista *cel putin* o varianta in care adversarul obtine un castig mai bun decat daca am fi jucat miscarea M1. Nu conteaza exact cat de slaba poate fi miscarea M2 fata de M1. O analiza amanuntita ar putea releva ca poate fi si mai slaba decat am constatat initial, insa acest lucru este irelevant. De ce insa ignoram intregi subarbori si miscari potential bune numai pentru o miscare slaba gasita? Pentru ca, in conformitate cu principiul de maximizare al castigului folosit de fiecare jucator, adeversarul va alege exact acea miscare ce ii va da un castig maximal. Daca exista o varianta si mai buna pentru el este irelevant, deoarece noi suntem interesati daca cea mai slaba miscare buna a lui este mai buna decat miscarea noastra curent analizata.

O observatie foarte importanta se poate face analizand modul de functionare al acestui algoritm: este extrem de importanta ordonarea miscarilor dupa valoarea castigului. In cazul ideal in care cea mai buna miscare a jucatorului curent este analizata prima, toate celelalte miscari, fiind mai slabe, vor fi eliminate din cautare timpuriu. In cel mai defavorabil caz insa, in care miscarile sunt ordonate crescator dupa castigul furnizat, Alpha-beta are aceeasi compelxitate cu Mini/Nega-max, neobtinandu-se nicio imbunatatire. In medie se constata o imbunatatire vizibila a algoritmului Alpha-beta fata de Mini/Nega-max.

Rolul miscarilor analizate la inceput presupune stabilirea unor plafoane de minim si maxim legate de cat de bune/slabe pot fi miscarile. Astfel, plafonul de minim (Lower Bound), numit alpha stabileste ca o miscare nu poate fi mai slaba decat valoarea acestui plafon. Plafonul de maxim (Upper Bound), numit beta, este important doarece el foloseste la a stabili daca o miscare este prea buna pentru a fi luata in considerare. Depasirea plafonului de maxim inseamna ca o miscare este atat de buna incat adversarul nu ar fi permis-o, adica mai sus in arbore exista o miscare pe care ar fi putut s-o joace pentru a nu ajunge in situatia curent analizata. Astfel alpha si beta furnizeaza o fereastra folosita pentru a filtra miscarile posibile pentru cei doi jucatori. Evident aceasta fereastra se poate actualiza pe masura ce se analizeaza mai multe miscari. De exemplul plafonul minim alpha se mareste pe masura ce gasim anumite tipuri de miscari mai bune (better worst best moves). Asadar, in implementare tinem seama si de aceste doua plafoane. In conformitate cu principiul Minimax, plafonul de minim al unui jocator (alpha-ul) este plafonul de maxim al celuilalt (beta-ul) si invers. Prezentam in continuare o descriere grafica a algoritmului Alpha-beta:



1. Start at C. Descend to full-ply depth and assign the heuristic to a state and all siblings (MIN 2, 3). Back up these values to their parent node (MAX 3).

2. Offer this value to the grandparent (A), as its beta value. So, **A has beta=3.** A will be no larger than 3

3. Descend to A's other grandchildren. Terminate the search of their parent if any grandchildren is >= A's beta. **Node B is beta-pruned**, as shown, because its value must be at least 5.

4. Once A's value is known, offer it to its parent (C) as its alpha value. So **C has alpha=3**. C will be no smaller than 3.

5. Repeat this process, descending to C's great grandchildren (0) in a depth-first fashion. **D is alpha-pruned**, because no matter what happens on its right branch, it cannot be greater than 0.

6. Repeating on E, **E is alpha-pruned** because its beta value (2) is less than its parent's alpha value (3). So no matter what happens on its right branch, E cannot have a value greater than 2.

7. Therefore **C is 3.**

O animație foarte bună pentru execuția Alpha-beta prunning găsiți la [11].

In continuare prezentam o implementare conceptuala a Alpha-beta, atat pentru Minimax, cat si pentru Negamax:

Varianta Minimax:

```
        int alphaBetaMax( int alpha, int beta, int depthleft )
        {
                if ( depthleft == 0 ) return evaluate();
                for ( all moves)
                {
                        score = alphaBetaMin( alpha, beta, depthleft - 1 );
                        if( score >= beta )
                                return beta;   // beta-cutoff
                        if( score > alpha )
                                alpha = score;
                }
                return alpha;
        }

        int alphaBetaMin( int alpha, int beta, int depthleft )
        {
                if ( depthleft == 0 ) return -evaluate();
                for ( all moves)
                {
```

```
                        score = alphaBetaMax( alpha, beta, depthleft - 1 );
                        if( score <= alpha )
                                return alpha; // alpha-cutoff
                        if( score < beta )
                                beta = score;
                }
                return beta;
        }
```

Varianta Negamax:

```
        int alphaBeta( int alpha, int beta, int depthleft )
        {
                if( depthleft == 0 ) return evaluate();
                for ( all moves)
                {
                        score = -alphaBeta( -beta, -alpha, depthleft - 1 );
                        if( score >= beta )
                                return beta;   // beta cutoff
                        if( score > alpha )
                                alpha = score;
                }
                return alpha;
        }
```

Din nou remarcam claritatea si coerenta sporita a variantei negamax.

Complexitate

In continuare prezentam complexitatile asociate algoritmilor prezentati anterior. Pentru aceasta vom introduce cateva notiuni ce tin de terminologia folosita in descrierile de specialitate, dupa cum urmeaza:

- Branch factor – notat cu b, reprezinta in medie numarul de fii ai unui nod oarecare, neterminal, al arborelui de solutii
- Depth – notat cu d, reprezinta adancimea pana la care se face cautarea in arborele de solutii. Orice nod de adancime d va fi considerat terminal
- Ply (-pl plies) – reprezinta un nivel al arborelui

Folosind termenii de mai sus putem spune ca un arbore cu un branching factor b, care va fi examinat pana la un nivel d va furniza b^d noduri ce vor trebui procesate. Un algoritm mini/nega-max glasic care analizeaza toate starile posibile, deci fiecare nod va avea complexitatea $O(b^d)$, deci exponentiala. Cat de bun este insa Alpha-beta fata de un mini/nega-max naiv? Dupa cum s-a mentionat anterior, in functie de ordonarea miscarilor ce vor fi evaluate putem avea un caz cel mai favorabil si un caz cel mai defavorabil. Le vom examina separat:

Cazul cel mai favorabil, in care miscarile sunt ordonate descrescator dupa castig (deci ordonate optim), rezulta o complexitate O(b*1*b*1*b*1…de d ori…b*1) pentru d par sau O(b*1*b*1*b*1…de d ori…b) pentru d impar. Restrangand ambele expresii rezulta o complexitate $O(b^{d/2})$, sau $O(\sqrt{b^d})$. Asadar complexitatea este radical din complexitatea obtinuta cu un algoritm mini/nega-max naiv. Explicatia este ca pentru jucatorul 1 trebuie examinate toate miscarile posibile pentru a putea gasi miscarea optima. Insa, pentru fiecare miscare examinata, nu este necesara decat cea mai buna miscare a jucatorului 2 pentru a trunchia restul de miscari ale jucatorului 1, in afara de prima (prima fiind si cea mai buna).

Prin urmare, intr-un caz ideal, algoritmul Alpha-beta poate explora de 2 ori mai multe nivele in arborele de solutii fata de un algoritm mini/nega-max naiv.

Cazul cel mai defavorabil a fost deja discutat, in prezentarea Alpha-beta. El apare atunci cand miscarie sunt ordonate crescator dupa castigul furnizat unui jucator, astfel fiind necesara o examinare a tuturor nodurilor pentru gasirea celei mai bune miscari. In consecinta complexitatea devine egala cu cea a unui algoritm mini/nega-max naiv.

Concluzii si observatii

- Minimax este un algoritm ce analizeaza spatiul solutiilor unui joc de tip zero-sum, dar nu numai;
- Complexitatea Mini/Nega-max este una prohibitiva: $O(b^d)$, facandu-l impractic pentru examinarea unui volum mare de noduri; limitele sale sunt undeva in jurul a 3-4 nivele in arborele de solutii (pe masini standard);
- Este de preferat folosirea unei variante mai clare de implementare a minimax, si anume Negamax;
- Exista mai multe optimizari posibile pentru reducerea complexitatii, precum Alpha-Beta Pruning, Negascout, Transposition Tables

Referinte

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

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

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

[4] http://starbase.trincoll.edu/~ram/cpsc352/notes/minimax.html
[http://starbase.trincoll.edu/~ram/cpsc352/notes/minimax.html]

[5] https://chessprogramming.wikispaces.com/Negamax
[https://chessprogramming.wikispaces.com/Negamax]

[6] https://chessprogramming.wikispaces.com/Minimax
[https://chessprogramming.wikispaces.com/Minimax]

[7] https://chessprogramming.wikispaces.com/Alpha-Beta
[https://chessprogramming.wikispaces.com/Alpha-Beta]

[8] http://en.wikipedia.org/wiki/Zero-sum [http://en.wikipedia.org/wiki/Zero-sum]

[9] http://en.wikipedia.org/wiki/Game_theory [http://en.wikipedia.org/wiki/Game_theory]

[10] http://en.wikipedia.org/wiki/Combinatorial_game_theory
[http://en.wikipedia.org/wiki/Combinatorial_game_theory]

[11] http://www.emunix.emich.edu/~evett/AI/AlphaBeta_movie/index_movie.htm
[http://www.emunix.emich.edu/~evett/AI/AlphaBeta_movie/index_movie.htm]

Probleme

1. Tic-Tac-Toe și Nim

A sosit timpul să ne jucăm puțin, iar aplicatia de laborator presupune aplicarea unor algoritmi de tip

Minimax (Negamax si Alpha-Beta Pruning) în scopul rezolvării următoarelor 2 jocului:

1. **X și O (tic-tac-toe)**: pe un grid 3×3 în care câștigătorul este jucătorul care reușește prima linie, coloană sau diagonală completă.

2. **Nim**: fiind date N bile, grupate inițial într-o singură mulțime, fiecare jucător trebuie să spargă una din submulțimile create pe parcurs în 2 alte sub-mulțimi. Pot fi sparte mulțimi cu minim 3 elemente, iar jucătorul care nu mai poate efectua mutări pierde. Ex.: pt 7 bile inițiale, putem avea următoare secvență: J1: 6 – 1; J2: 4 – 2 – 1; J1: 2 – 2 – 1 – 1. J2 pierde.

Se doreste implementarea algoritmului minimax sau negamax astfel încât calculatorul să poată juca împotriva unui jucător uman. [4 pct pentru unul dintre jocuri, la alegere, 2p pentru celălalt]

Adițional, trebuie extins algoritmul minimax/negamax anterior într-un algoritm de tip alpha-beta pruning. [3 pct pentru unul dintre jocuri, la alegere, 1 pct pentru celălalt]

pa/laboratoare/laborator-05.txt · Last modified: 2013/03/24 15:15 by andrei.sfrent