Proiectarea Algoritmilor 2009-2010

Laborator 5

Minimax

Cuprins:

1. Obiective laborator

a. Insusirea unor cunostinte elemenatre legate de teoria jocurilor precum si a jocurilor de tip zero-sum

b. Insusirea abilitatii de rezolvare a problemelor ce presupun cunoasterea si exploatarea conceptului de zero-sum

c. Insusirea unor cunostinte elementare despre algoritmii necesari rezolavrii unor probleme de tip zero-sum

2. Aplicatii practice

Algoritmul minimax si variantele imbunatatite ale sale (Negamax, Alpha-Beta, Negascout etc) sunt folosite in diferite domenii precum teoria jocurilor (Game Theory), teoria jocurilor combinatorice (Combinatorial Game Theory – CGT), teoria deciziei (Decision Theory) si statistica, daca e sa numim doar cateva. Asadar diferite variante ale algoritmului sunt necesare in proiectarea si implementarea diverselor aplicatii legate de inteligenta artificiala, economie, dar si in domenii precum stiinta politica si biologie.

3. Descrierea problemei si a rezolvarilor

In cadrul laboratorului 5 se va pune accentul pe varianta Minimax ce permite abordarea unor probleme ce tin de teoria jocurilor combinatorice. CGT este o ramura a matematicii ce se ocupa cu studierea diferitelor jocuri 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 legate de implementare.

Aplicatia de laborator va presupune aplicarea unor algoritmi de tip Minimax (Negamax si Alpha-Beta Pruning) in scopul rezolvarii jocului de X si O (tic-tac-toe). (Nota: acest joc, desi rezolvabil prin metode mai putin costisitoare din punct de vedere al efortului de calcul, este ales in scop didactic). In continuare este prezentat pas cu pas algoritmul minimax, insotit de pseudocod ajutator.

3.1. Minimax

Strategia pe care se bazeaza ideea algoritmului este ca jucatorii participanti sa adopte urmatoarele strategii:

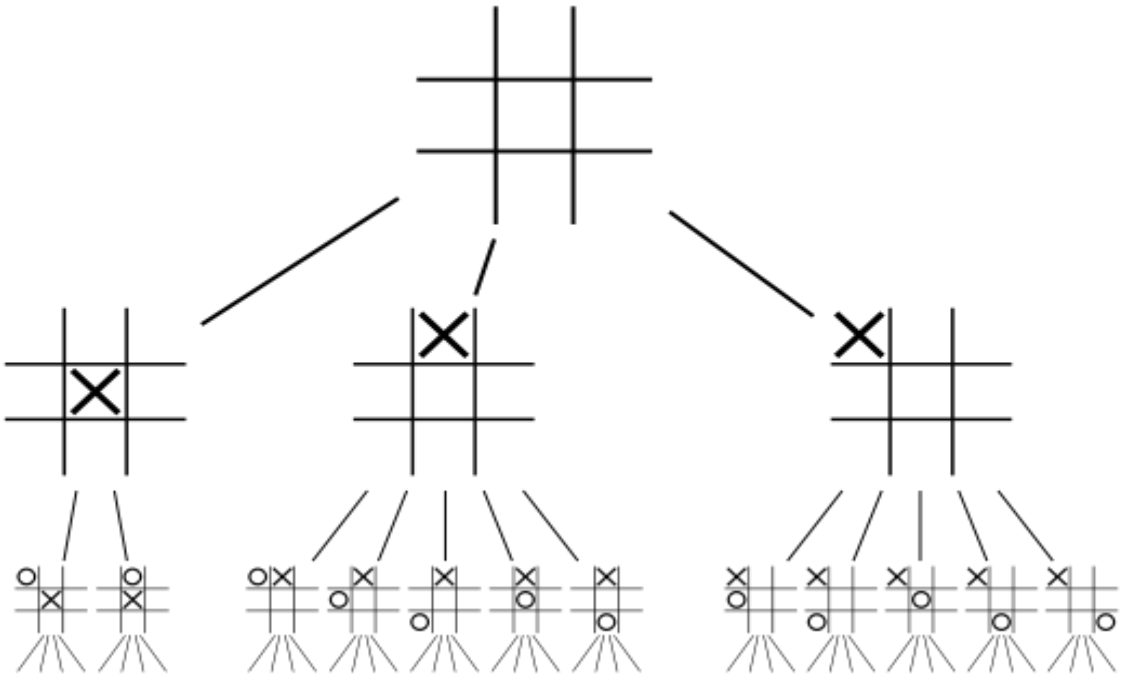
- Jucatorul 1 (maxi) va incerca mereu sa-si maximizeze propriul castig prin mutarea pe care o are de facut
- Jucatorul 2 (mini) va incerca mereu sa minimizeze castigul jucatorului 1 la fiecare mutare

De ce merge o astfel de abordare? Dupa cum se preciza la inceput discutia se axeaza pe jocuri de tip 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 |

| Loss_Player_1 | = Win_Player_2

Punem in continuare problema reprezentarii problemei, in special a 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, grad de optimziare etc. si nu se va insista asupra lor momentan.

Avand notiunile de baza asupra strategiei celor doi jucatori, precum si a reprezentarii spatiului solutiilor problemei, prezentam in continuare o formulare incipienta a algoritmului Minimax, in pseudo-C-cod:

```
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;
}
```

Ce reprezinta variabila `depth`? Datorita spatiului de solutii mare, de multe ori coplesitor ca volum, o inspecatare completa a acestuia (sau chiar una in amanunt) este nefezabila si impractica din punct de vedere al timpului consumat sau chiar a memoriei consumate (se vor discuta aceste aspecte in paragraful legat de complexitate). Asadar de cele mai multe ori este preferata o abordare ce parcurge arborele numai pana la o anumita adancime depth. Aceasta abordare permite examinarea arborelui destul pentru luarea unei 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.

Se observa de asemenea recursivitatea indirecta. Prin conventie acceptam ca inceputul algoritmului sa fie cu functia maxi. Astfel succesiv se analizeaza 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.

Mai departe se va prezenta o varianta imbunatatita a algoritmului minimax, numita **Negamax**.

3.2. 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 conchide ca de fapt fiecare incearca sa maximizeze la fiecare pas. Intr-adevar putem spune ca jucatorul mini incearca sa maximizeze in modul suma pierduta de maxi. Astfel putem formula urmatoarea implementare in pseudo-C-cod 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, int type)
{
        if ( depth == 0 )
        {
                if(type) // maxi
                        return evaluate();
                else     // mini
                        return -1*evaluate();
        }
        int max = -oo;
        for ( all moves)
        {
                score = -negaMax( depth - 1 , !type);
                if( score > max )
                        max = score;
        }
        return max;
}
```

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

a. Claritatea sporita a codului
b. Eleganta implementarii
c. Usurinta in intretinere si extindere a functionalitatii

Din punct de vedere al complexitatii temporale, Negamax nu difera absolut deloc de Minimax (ambele examineaza acelasi numar de stari in arborele de solutii), avantajele sale fata de ultimul fiind exclusiv cele prezentate mai sus. Putem conchide ca este de preferat o implementare ce foloseste negamx, fata de una folosind minimax in rezolvarea unor probleme ce tin de aceasta tehnica.

OBSERVATIE!

Este necesar ca evaluearea din algoritmul negamax sa tina seama de tipul jucatorului din punctul caruia se evalueaza starea curenta. Astfel daca analizam din punctul de vedere a jucatorului *mini* este necesara o inmultire a scorului returnat cu -1, astfel incat sa se poata respecta principul minimax.

3.3. 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 (Nota: prin inutilitate intelegem acele stari care nu pot fi atinse datorita incalcarii principiului de maximizare a castigului la fiecare runda).

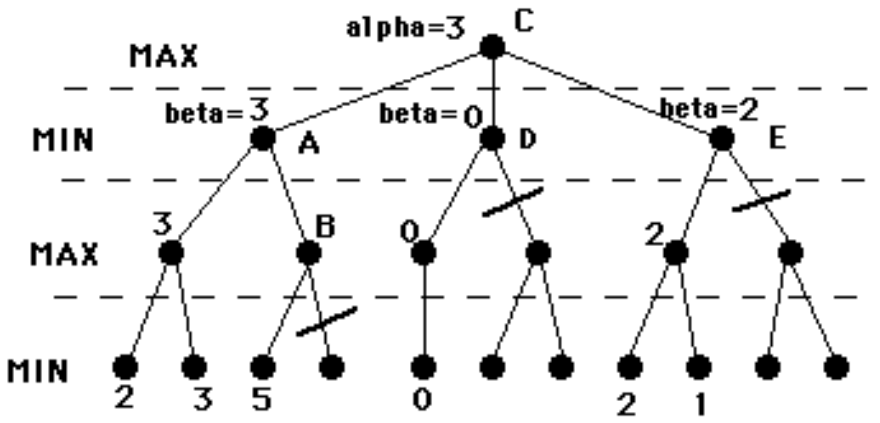
O imbunatatire substantiala a minimax/negamax este Alpha-beta pruning. Acest algoritm incearca sa optimizeze mini/nega-max profitand de o observatie importanta:

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 foarte importanta observatie se poate face analizand modul de functionare al acestui algoritm. Anume ca 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 insa se constata o imbunatatire vizibila a algoritmului Alpha-beta fata de Mini/Nega-max.

Rolul miscarilor analizate la inceput este 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 palfoane. In conformitate cu principiul Minimax, plafonul de minim al unui jucator (alpha-ul) este plafonul de maxim al celuilalt (beta-ul) si invers. Prezentam in continuare o descriere grafica a algrotmului 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.**

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

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.

3.4. Complexitate

In acest moment cunoastem principiul de functionare al algoritmului de baza pentru cautarea in spatiul solutiilor unei probleme de tip zero-sum game: Minimax (sau Negamax). De asemenea s-a discutat si pe seama celei mai de baza (si importante) optimizari a acestui algoritm: Alpha-beta pruning. In continuare prezentam complexitatile asociate acestor algoritmi. 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 clasic 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:

a) 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.

b) 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.

4. 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 fiind undeva pe la 3-4 nivele in arborele de solutii (pe masini standard).
- Este de preferat folosirea unei variante mai clare ca implementare a minimax, anume Negamax
- Exista mai multe optimizari posibile pentru reducerea complexitatii, precum Alpha-Beta Pruning, Negascout, Transposition Tables

4.1 Key words:

Minimax
Negamax
Alpha-beta pruning
game theory
game tree
tic tac toe
Negascout
Transposition tables
Principal Variation Search
zero-sum
combinatorial game theory

5. Referinte

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

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

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

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

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

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

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

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

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

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

Responsabil laborator: Mihail Claudiu Bogdan - claudiu.bogdan.mihail@gmail.com

6. Aplicatii

1. Folosind scheletul de cod pus la dispozitie (sau propria implementare) implementati algoritmul de minimax sau negamax, astfel incat calculatorul sa poata juca impotriva unui jucator uman intr-un joc de X si O (tic-tac-toe).

2. (bonus) Folosind scheletul de cod pus la dispozitie (sau propria implementare), extindeti algoritmul de minimax/negamax de la punctul 1, intr-un algoritm alpha-beta pruning.