Proiectarea Algoritmilor 2011-2012

Laborator 5

Minimax

Cuprins

1 Obiective laborator

- Însușirea unor cunoștințe de bază despre teoria jocurilor precum și despre jocurile de tip zero-sum;
- Însușirea abilității de rezolvare a problemelor ce presupun cunoașterea și exploatarea conceptului de zero-sum;
- Însușirea unor cunoștințe elementare despre algoritmii necesari rezolvării unor probleme de tip zero-sum.

2 Importanţă – aplicaţii practice

Algoritmul Minimax și variantele sale îmbunătățite (Negamax, Alpha-Beta, Negascout etc) sunt folosite în diverse domenii precum teoria jocurilor (Game Theory), teoria jocurilor combinatorice (Combinatorial Game Theory – CGT), teoria deciziei (Decision Theory) și statistică. Astfel,

diferite variante ale algoritmului sunt necesare în proiectarea și implementarea de aplicații legate de inteligență artificială, economie, dar și în domenii precum științe politice sau biologie.

3 Descrierea problemei și a rezolvărilor

Algoritmii Minimax permit abordarea unor probleme ce țin de teoria jocurilor combinatorice. CGT este o ramură a matematicii ce se ocupă cu studierea jocurilor în doi (two-player games), în care participanții își modifica rând pe rând pozițiile în diferite moduri, prestabilite de regulile jocului, pentru a îndeplini una sau mai multe condiții de câștig. Exemple de astfel de jocuri sunt: șah, go, dame (checkers), X și O (tic-tac-toe) etc. CGT nu studiază jocuri ce presupun implicarea unui element aleator (șansa) în derularea jocului precum poker, blackjack, zaruri etc. Astfel decizia abordării unor probleme rezolvabile prin metode de tip Minimax se datorează în principal simplității atât conceptuale, cat și raportat la implementarea propriu-zisa.

3.1 Minimax

Strategia pe care se bazează ideea algoritmului este ca jucătorii implicați adoptă următoarele strategii:

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

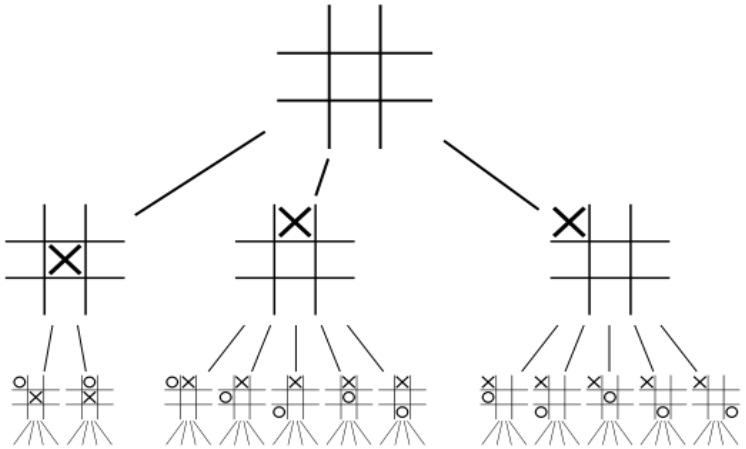
De ce merge o astfel de abordare? După cum se preciza la început, discuția se axează pe jocuri zero-sum. Acest lucru garantează, printre altele, ca orice câștig al Jucătorului 1 este egal cu modulul sumei pierdute de Jucătorul 2. Cu alte cuvinte cat pierde Jucător 2, atât câștiga Jucător 1. Invers, cat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

$$Win\_Player\_1 = |\ Loss\_Player\_2\ |$$

$$|\ Loss\_Player\_1\ | = Win\_Player\_2$$

3.1.1 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 asociata o stare a jocului în desfășurare (game state). Pentru exemplul nostru de X și O putem considera următorul arbore (parțial) de soluții, ce corespunde primelor mutări ale lui X, respectiv O:



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

Având noțiunile de bază asupra strategiei celor doi jucători, precum și a reprezentării spațiului soluțiilor 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;
}
```

3.1.2 Argumentarea utilizării unei adâncimi maxime

Datorita spațiului de soluții mare, de multe ori copleșitor ca volum, o inspectare completa a acestuia nu este fezabila și devine impracticabila 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 preferata o abordare care parcurge arborele numai pana la o anumita adâncime maxima („depth”). Aceasta abordare permite examinarea arborelui destul de mult pentru a putea lua decizii minimalist coerente în desfășurarea jocului. Totuși, dezavantajul major este că

pe termen lung se poate dovedi ca decizia luata la adâncimea depth nu este global favorabilă jucătorului în cauza.

De asemenea, se observă recursivitatea indirectă. Prin convenție acceptăm că începutul algoritmului sa fie cu funcția maxi. Astfel, se analizează succesiv diferite stări ale jocului din punctul de vedere al celor doi jucători până la adâncimea depth. Rezultatul intors este scorul final al mișcării celei mai bune.

3.2 Negamax

Negamax este o variantă a minimax, care se bazează pe următoarea observație: fiind intr-un joc zero-sum în care câștigul unui jucător este egal cu modulul sumei pierdute de celalalt jucător și invers, putem deriva remarca ca fiecare jucător încearcă să-și maximizeze propriul câștig la fiecare pas. Intr-adevăr putem spune că jucătorul mini încearcă de fapt sa maximizeze în modul suma pierdută de maxi. Astfel putem formula următoarea implementare ce profita de observația de mai sus (Nota: putem exprima această observație și 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 formulări fata de Minimax-ul standard prezentat anterior:

- Claritatea sporită a codului
- Eleganța implementării
- Ușurința în întreținere și extindere a funcționalității

Din punctul de vedere al complexității temporale, Negamax nu diferă absolut deloc de Minimax (ambele examinează același număr de stări în arborele de soluții). Putem concluziona ca este de preferat o implementare ce folosește negamax fata de una bazată pe minimax în rezolvarea unor probleme ce țin de aceasta tehnica.

3.3 Alpha-beta pruning

Pana acum s-a discutat despre algoritmii Minimax și Negamax. Aceștia sunt algoritmi exhaustivi (exhausting search algorithms). Cu alte cuvinte, ei găsesc soluția optimă 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 ca extrem de multe stări de joc inutile sunt explorate (este vorba de acele stări care nu pot fi atinse datorita încălcării principiului de maximizare a câștigului la fiecare rundă).

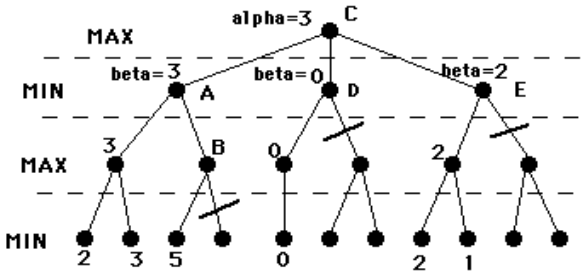
O îmbunătățire substanțială a minimax/negamax este Alpha-beta pruning. Acest algoritm încearcă sa optimizeze mini/nega-max profitând de o observație importantă: pe parcursul examinării arborelui de soluții se pot elimina întregi subarbori, corespunzători unei mișcări m, dacă pe parcursul analizei găsim că mișcarea m este mai slaba calitativ decât cea mai buna mișcare curentă.

Astfel, considerăm că pornim cu o prima mișcare M1. După ce analizăm această mișcare în totalitate și îi atribuim un scor, continuăm să analizăm mișcarea M2. Daca 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 daca am fi jucat mișcarea M1. Nu contează *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 ignoram 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 jucător, 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 daca cea mai slaba mișcare bună a lui este mai bună decât mișcarea noastră curent analizata.

O observație foarte importantă se poate face analizând modul de funcționare al acestui algoritm: este extrem de importanta ordonarea mișcărilor după valoarea câștigului. In cazul ideal în care cea mai buna mișcare a jucătorului curent este analizata prima, toate celelalte mișcări, fiind mai slabe, vor fi eliminate din căutare timpuriu. În cel mai defavorabil caz însă, în care mișcările sunt ordonate crescător după câștigul furnizat, Alpha-beta are aceeași complexitate cu Mini/Nega-max, neobținându-se nicio îmbunătățire. În medie se constată o îmbunătățire vizibilă a algoritmului Alpha-beta fata de Mini/Nega-max.

Rolul mișcărilor analizate la început presupune stabilirea unor plafoane de minim și maxim legate de cat de bune/slabe pot fi mișcările . Astfel, plafonul de minim (Lower Bound), numit alpha stabilește că o mișcare nu poate fi mai slabă decât valoarea acestui plafon. Plafonul de maxim (Upper Bound), numit beta, este important deoarece el folosește la a stabili daca o mișcare este prea buna pentru a fi luata în considerare. Depășirea plafonului de maxim înseamnă că o mișcare este atât de buna încât adversarul nu ar fi permis-o, adică mai sus în arbore exista o mișcare pe care ar fi putut s-o joace pentru a nu ajunge în situația curent analizată. Astfel alpha și beta furnizează o fereastra folosita pentru a filtra mișcările posibile pentru cei doi jucători. Evident aceasta fereastra se poate actualiza pe măsură ce se analizează mai multe mișcări. De exemplul plafonul minim alpha se mărește pe măsură ce găsim anumite tipuri de mișcări mai bune (better worst best moves). Așadar, în implementare ținem seama și de aceste doua plafoane. In conformitate cu principiul Minimax, plafonul de minim al unui jucător (alpha-ul) este plafonul de maxim al celuilalt (beta-ul) și invers. Prezentăm în continuare o descriere grafică a algoritmului Alpha-beta:



In continuare prezentam o implementare conceptuala a Alpha-beta, atât pentru Minimax, cât și 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 remarcăm claritatea și coerenta sporita a variantei negamax.

3.4 Complexitate

In continuare prezentam complexitățile asociate algoritmilor prezentați anterior. Pentru aceasta vom introduce câteva noțiuni ce țin de terminologia folosita în descrierile de specialitate, după cum urmează:

- Branch factor – notat cu ***b***, reprezintă în medie numărul de fii ai unui nod oarecare, neterminal, al arborelui de soluții
- Depth – notat cu ***d***, reprezintă adâncimea pana la care se face căutarea în arborele de soluții. Orice nod de adâncime ***d*** va fi considerat terminal
- Ply (-pl plies) – reprezintă 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 analizează toate stările posibile, deci fiecare nod va avea complexitatea **O(**b^d**)**, deci exponențială. Cât de bun este însă Alpha-beta fata de un mini/nega-max naiv? După 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.

Cazul cel mai favorabil, în care mișcările sunt ordonate descrescător după câștig (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. Restrângând ambele expresii rezulta o complexitate O($b^{d/2}$), sau O($\sqrt{b^d}$). Așadar complexitatea este radical din complexitatea obținută cu un algoritm mini/nega-max naiv. Explicația este că pentru jucătorul 1 trebuie examinate toate mișcările posibile pentru a putea găsi mișcarea optima. Însă, pentru fiecare mișcare examinată, nu este necesară decât cea mai bună mișcare a jucătorului 2 pentru a trunchia restul de mișcări ale jucătorului 1, în afara de prima (prima fiind și cea mai buna).

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

Cazul cel mai defavorabil a fost deja discutat, în prezentarea Alpha-beta. El apare atunci când mișcare sunt ordonate crescător după câștigul furnizat unui jucător, astfel fiind necesara o examinare a tuturor nodurilor pentru găsirea celei mai bune mișcări. În consecință complexitatea devine egală cu cea a unui algoritm mini/nega-max naiv.

4 Concluzii și observații

- Minimax este un algoritm ce analizează spațiul soluțiilor unui joc de tip zero-sum, dar nu numai;
- Complexitatea Mini/Nega-max este una prohibitiva: **O(**b^d**)**, făcându-l puțin practic pentru examinarea unui volum mare de noduri; limitele sale sunt undeva în jurul a 3-4 nivele în arborele de soluții (pe mașini standard);
- Este de preferat folosirea unei variante mai clare de implementare a minimax, și anume Negamax;

- Există mai multe optimizări posibile pentru reducerea complexității, precum Alpha-Beta Pruning, Negascout, Transposition Tables

5 Referințe

[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