Algoritmul Minimax

Oponenţii din cadrul jocului pe care îl vom trata prin aplicarea Algoritmului Minimax vor fi numiţi, în continuare, MIN şi respectiv MAX. MAX reprezintă jucătorul care încearcă să câştige sau să îşi maximizeze avantajul avut. MIN este oponentul care încearcă să minimizeze scorul lui MAX. Se presupune că MIN foloseşte aceeaşi informaţie şi încearcă întotdeauna să se mute la acea stare care este cea mai nefavorabilă lui MAX.

Algoritmul Minimax este conceput pentru a determina strategia optimă corespunzătoare lui MAX şi, în acest fel, pentru a decide care este cea mai bună primă mutare:

Algoritmul Minimax

1. Generează întregul arbore de joc, până la stările terminale.
2. Aplică funcţia de utilitate fiecărei stări terminale pentru a obţine valoarea corespunzătoare stării.
3. Deplasează-te înapoi în arbore, de la nodurile-frunze spre nodul-rădăcină, determinând, corespunzător fiecărui nivel al arborelui, valorile care reprezintă utilitatea nodurilor aflate la acel nivel. Propagarea acestor valori la niveluri anterioare se face prin intermediul nodurilor- părinte succesive, conform următoarei reguli:
   - dacă starea-părinte este un nod de tip MAX, atribuie-i maximul dintre valorile avute de fiii săi;
   - dacă starea-părinte este un nod de tip MIN, atribuie-i minimul dintre valorile avute de fiii săi.
4. Ajuns în nodul-rădăcină, alege pentru MAX acea mutare care conduce la valoarea maximă.

❑

- Observaţie: Decizia luată la pasul 4 al algoritmului se numeşte *decizia minimax*, întrucât ea maximizează utilitatea, în ipoteza că oponentul joacă perfect cu scopul de a o minimiza.

Un arbore de căutare cu valori minimax determinate conform Algoritmului Minimax este cel din Fig. 3.1:

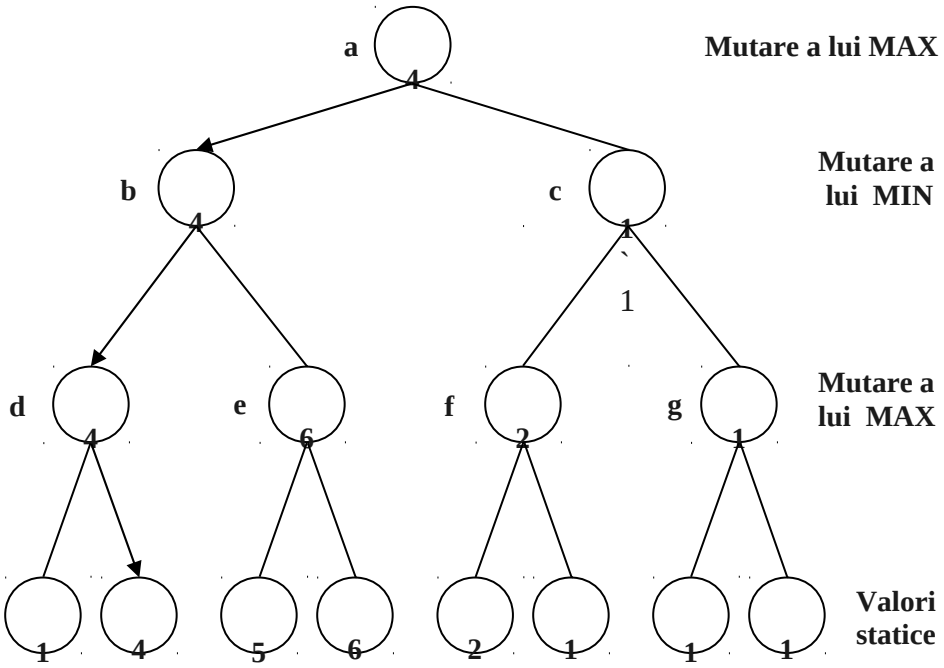

Fig. 3.1

Valorile poziţiilor de la ultimul nivel sunt determinate de către funcţia de utilitate şi se numesc *valori statice*. Valorile minimax ale nodurilor interne sunt calculate în mod dinamic, în manieră bottom-up, nivel cu nivel, până când este atins nodul-rădăcină. Valoarea rezultată, corespunzătoare acestuia, este 4 şi, prin urmare, cea mai bună mutare a lui MAX din poziţia *a* este *a-b*. Cel mai bun răspuns al lui MIN este *b-d*. Această secvenţă a jocului poartă denumirea de *variaţie principală*. Ea defineşte jocul optim de tip minimax pentru ambele părţi. Se observă că valoarea poziţiilor de-a lungul variaţiei principale nu variază. Prin urmare, mutările corecte sunt cele care *conservă valoarea jocului*.

Dacă adâncimea maximă a arborelui este *m* şi dacă există *b* "mutări legale" la fiecare punct, atunci complexitatea de timp a Algoritmului Minimax este $O(b^m)$. Algoritmul reprezintă o căutare de tip depth-first (deşi aici este sugerată o implementare bazată pe recursivitate şi nu una care foloseşte o coadă de noduri), astfel încât cerinţele sale de spaţiu sunt numai liniare în *m* şi *b*.

În cazul jocurilor reale, cerinţele de timp ale algoritmului sunt total nepractice, dar acest algoritm stă la baza atât a unor metode mai realiste, cât şi a analizei matematice a jocurilor.

Întrucât, pentru majoritatea jocurilor interesante, arborele de joc nu poate fi alcătuit în mod exhaustiv, au fost concepute diverse metode care se bazează pe căutarea efectuată numai într-o anumită porţiune a arborelui de joc. Printre acestea se numără şi tehnica Minimax, care, în majoritatea cazurilor, va căuta în arborele de joc numai până la *o anumită adâncime*, de obicei constând în numai câteva mutări. Ideea este de a evalua aceste poziţii terminale ale căutării, fără a mai căuta dincolo de ele, cu scopul de a face economie de timp. Aceste estimări se propagă apoi în sus de-a lungul arborelui, conform principiului Minimax. Mutarea care conduce de la poziţia iniţială, nodul-rădăcină, la cel mai promiţător succesor al său (conform acestor evaluări) este apoi efectuată în cadrul jocului.

Algoritmul general Minimax a fost amendat în două moduri: funcţia de utilitate a fost înlocuită cu o *funcţie de evaluare*, iar testul terminal a fost înlocuit de către un aşa-numit *test de tăiere*.

Cea mai directă abordare a problemei deţinerii controlului asupra cantităţii de căutare care se efectuează este aceea de a fixa o limită a adâncimii, astfel încât testul de tăiere să aibă succes pentru toate nodurile aflate la sau sub adâncimea *d*. Limita de adâncime va fi aleasă astfel încât cantitatea de timp folosită să nu depăşească ceea ce permit regulile jocului. O abordare mai robustă a acestei probleme este aceea care aplică „iterative deepening”. În acest caz, atunci când timpul expiră, programul întoarce mutarea selectată de către cea mai adâncă căutare completă.

Funcţii de evaluare

O *funcţie de evaluare* întoarce o *estimaţie*, realizată dintr-o poziţie dată, a utilităţii aşteptate a jocului. Ea are la bază evaluarea şanselor de câştigare a jocului de către fiecare dintre părţi, pe baza calculării caracteristicilor unei poziţii. Performanţa unui program referitor la jocuri este extrem de dependentă de calitatea funcţiei de evaluare utilizate.

Funcţia de evaluare trebuie să îndeplinească anumite condiţii evidente: ea trebuie să concorde cu funcţia de utilitate în ceea ce priveşte stările terminale, calculele efectuate nu trebuie să dureze prea mult şi ea trebuie să reflecte în mod corect şansele efective de câştig. O valoare a funcţiei de evaluare acoperă mai multe poziţii diferite, grupate laolaltă într-o *categorie* de poziţii etichetată cu o anumită valoare. Spre exemplu, în jocul de şah, fiecare *pion* poate avea valoarea 1, un *nebun* poate avea valoarea 3 şamd.. În poziţia de deschidere evaluarea este 0 şi toate poziţiile până la prima captură vor avea aceeaşi evaluare. Dacă MAX reuşeşte să captureze un nebun fără a pierde o piesă, atunci poziţia rezultată va fi evaluată la valoarea 3. Toate poziţiile de acest fel ale lui MAX vor fi grupate într-

o *categorie* etichetată cu “3”. Funcţia de evaluare trebuie să reflecte şansa ca o poziţie aleasă la întâmplare dintr-o asemenea categorie să conducă la câştig (sau la pierdere sau la remiză) pe baza experienţei anterioare.

Funcţia de evaluare cel mai frecvent utilizată presupune că valoarea unei piese poate fi stabilită independent de celelalte piese existente pe tablă. Un asemenea tip de funcţie de evaluare se numeşte *funcţie liniară ponderată*, întrucât are o expresie de forma

$$w_1f_1 + w_2f_2 + ... + w_nf_n,$$

unde valorile $w_i, i = \overline{1,n}$ reprezintă ponderile, iar $f_i, i = \overline{1,n}$ sunt caracteristicile unei anumite poziţii. În cazul jocului de şah, spre exemplu $w_i, i = \overline{1,n}$ ar putea fi valorile pieselor (1 pentru pion, 3 pentru nebun etc.), iar $f_i, i = \overline{1,n}$ ar reprezenta numărul pieselor de un anumit tip aflate pe tabla de şah.

În construirea formulei liniare trebuie mai întâi alese caracteristicile, operaţie urmată de ajustarea ponderilor până în momentul în care programul joacă suficient de bine. Această a doua operaţie poate fi automatizată punând programul să joace multe partide cu el însuşi, dar alegerea unor caracteristici adecvate nu a fost încă realizată în mod automat.

Un program Prolog care calculează valoarea minimax a unui nod intern dat va avea ca relaţie principală pe

```
miminax(Poz,SuccBun,Val)
```

unde Val este valoarea minimax a unei poziţii Poz, iar SuccBun este cea mai bună poziţie succesor a lui Poz (i.e. mutarea care trebuie făcută pentru a se obţine Val). Cu alte cuvinte, avem:

minimax(Poz, SuccBun, Val) :

Poz este o poziţie, Val este valoarea ei de tip minimax;
cea mai bună mutare de la Poz conduce la poziţia SuccBun.

La rândul ei, relaţia

```
mutari(Poz,ListaPoz)
```

corespunde regulilor jocului care indică mutările “legale” (admise): ListaPoz este lista poziţiilor succesor legale ale lui Poz. Predicatul mutari va eşua dacă Poz este o poziţie de căutare terminală (o frunză a arborelui de căutare). Relaţia

```
celmaibun(ListaPoz,PozBun,ValBuna)
```

selectează cea mai bună poziţie PozBun dintr-o listă de poziţii candidate ListaPoz. ValBuna este valoarea lui PozBun şi, prin urmare, şi a lui Poz. „Cel mai bun” are aici sensul de maxim sau de minim, în funcţie de partea care execută mutarea.

Principiul Minimax

```
minimax(Poz,SuccBun,Val):-
        % mutările legale de la Poz produc ListaPoz
        mutări(Poz,ListaPoz),!,
        celmaibun(ListaPoz,SuccBun,Val);
        %Poz nu are succesori şi este evaluat
        %în mod static
        staticval(Poz,Val).
celmaibun([Poz],Poz,Val):-
        minimax(Poz,_,Val),!.
celmaibun([Poz1|ListaPoz],PozBun,ValBuna):-
        minimax(Poz1,_,Val1),
        celmaibun(ListaPoz,Poz2,Val2),
        maibine(Poz1,Val1,Poz2,Val2,PozBun,ValBuna).

%Poz0 mai bună decât Poz1

maibine(Poz0,Val0,Poz1,Val1,Poz0,Val0):-
       %Min face o mutare la Poz0
       %Max preferă valoarea maximă
        mutare_min(Poz0),                          Val0>Val1,!
        ;
        %Max face o mutare la Poz0
        %Min preferă valoarea mai mică
        mutare_max(Poz0),
        Val0<Val1,!.

%Altfel, Poz1 este mai bună decât Poz0

maibine(Poz0,Val0,Poz1,Val1,Poz1,Val1).
```