JOCURILE CA PROBLEME DE CĂUTARE

- **Jocurile reprezintă o arie de aplicație interesantă pentru algoritmii euristici.**
- **Jocurile de două persoane sunt în general complicate datorită existenței unui oponent ostil şi imprevizibil. De aceea ele sunt interesante din punctul de vedere al dezvoltării euristicilor, dar aduc multe dificultăți în dezvoltarea şi aplicarea algoritmilor de căutare.**
- **Oponentul introduce *incertitudinea*, întrucât nu se ştie niciodată ce va face acesta la pasul următor. În esență, toate programele referitoare la jocuri trebuie să trateze aşa numita *problemă de contingență*. (Aici *contingență* are sensul de *întâmplare*).**
- **Incertitudinea care intervine în cazul jocurilor nu este de aceeaşi natură cu cea introdusă, de pildă, prin aruncarea unui zar sau cu cea determinată de starea vremii. Oponentul va încerca, pe cât posibil, să facă mutarea cea mai puțin benignă, în timp ce**

zarul sau vremea sunt presupuse a nu lua în considerație scopurile agentului.

- **Complexitatea jocurilor introduce *un tip de incertitudine complet nou*. Astfel, incertitudinea se naşte nu datorită faptului că există informație care lipseşte, ci datorită faptului că jucătorul nu are timp să calculeze consecințele exacte ale oricărei mutări. Din acest punct de vedere, jocurile se aseamănă infinit mai mult cu lumea reală decât problemele de căutare standard.**

Întrucât, în cadrul unui joc, există, de regulă, limite de timp, jocurile *penalizează ineficiența* extrem de sever. Astfel, dacă o implementare a căutării de tip A*, care este cu 10% mai puțin eficientă, este considerată satisfăcătoare, un program pentru jocul de şah care este cu 10% mai puțin eficient în folosirea timpului disponibil va duce la pierderea partidei. Din această cauză, studiul nostru se va concentra asupra *tehnicilor de alegere a unei bune mutări atunci când timpul este limitat.* *Tehnica de "retezare"* ne va permite să ignorăm porțiuni ale arborelui de căutare care nu pot avea nici un rol în stabilirea alegerii finale, iar *funcțiile de evaluare euristice* ne vor permite să aproximăm utilitatea reală a unei stări fără a executa o căutare completă.

Ne vom referi la tehnici de joc corespunzătoare unor *jocuri de două persoane cu informație completă*, cum ar fi şahul.

În cazul jocurilor interesante, arborii rezultați sunt mult prea complecşi pentru a se putea realiza o căutare exhaustivă, astfel încât sunt necesare abordări de o natură diferită. Una dintre metodele clasice se bazează pe „principiul minimax”, implementat în mod eficient sub forma Algoritmului Alpha-Beta (bazat pe aşa-numita *tehnică de alpha-beta retezare*).

O definire formală a jocurilor

- **Tipul de jocuri la care ne vom referi în continuare este acela al jocurilor de două persoane cu informație perfectă sau completă. În astfel de jocuri există doi jucători care efectuează mutări în mod alternativ, ambii jucători dispunând de informația completă asupra situației curente a jocului. (Prin aceasta, este exclus studiul majorității jocurilor de cărți). Jocul se încheie atunci când este atinsă o poziție calificată ca fiind "terminală" de către regulile jocului - spre exemplu, "mat" în jocul de şah. Aceleaşi reguli determină care este rezultatul jocului care s-a încheiat în această poziție terminală. Un asemenea joc poate fi reprezentat printr-un *arbore de joc* în care nodurile corespund situațiilor (stărilor), iar arcele corespund mutărilor. Situația inițială a jocului este reprezentată de nodul rădăcină, iar frunzele arborelui corespund pozițiilor terminale.**

Vom lua în considerație cazul general al unui joc cu doi jucători, pe care îi vom numi MAX şi respectiv MIN. MAX va face prima mutare, după care jucătorii vor efectua mutări pe rând, până când jocul ia sfârşit. La finalul jocului vor fi acordate puncte jucătorului câştigător (sau vor fi acordate anumite penalizări celui care a pierdut).

Un joc poate fi definit, în mod formal, ca fiind *un anumit tip de problemă de căutare având următoarele componente*:

- ***starea inițială*, care include poziția de pe tabla de joc şi o indicație referitoare la cine face prima mutare;**
- ***o mulțime de operatori*, care definesc mişcările permise ("legale") unui jucător;**
- ***un test terminal*, care determină momentul în care jocul ia sfârşit;**
- ***o funcție de utilitate* (numită şi *funcție de plată*), care acordă o valoare numerică rezultatului unui joc; în cazul jocului de şah, spre exemplu, rezultatul poate fi *câştig*, *pierdere* sau *remiză*, situații care pot fi reprezentate prin valorile 1, -1 sau 0.**

Dacă un joc ar reprezenta o problemă standard de căutare, atunci acțiunea jucătorului MAX ar consta din căutarea unei secvențe de mutări care conduc la o stare terminală reprezentând o stare câştigătoare (conform funcției de utilitate) şi din efectuarea primei mutări aparținând acestei secvențe. Acțiunea lui MAX interacționează însă cu cea a jucătorului MIN. Prin urmare, MAX trebuie să găsească *o strategie* care va conduce la o stare terminală câştigătoare, indiferent de acțiunea lui MIN. Această strategie include mutarea corectă a lui MAX corespunzătoare fiecărei mutări posibile a lui MIN. În cele ce urmează, vom începe prin a arăta cum poate fi găsită strategia optimă (sau rațională), deşi în realitate nu vom dispune de timpul necesar pentru a o calcula.

Algoritmul Minimax

Oponenții din cadrul jocului pe care îl vom trata prin aplicarea Algoritmului Minimax vor fi numiți, 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 informație 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 funcția de utilitate fiecărei stări terminale pentru a obține 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.**

❑

- **<u>Observaţie</u>: Decizia luată la pasul 4 al algoritmului se numeşte *<u>decizia minimax</u>*, 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:

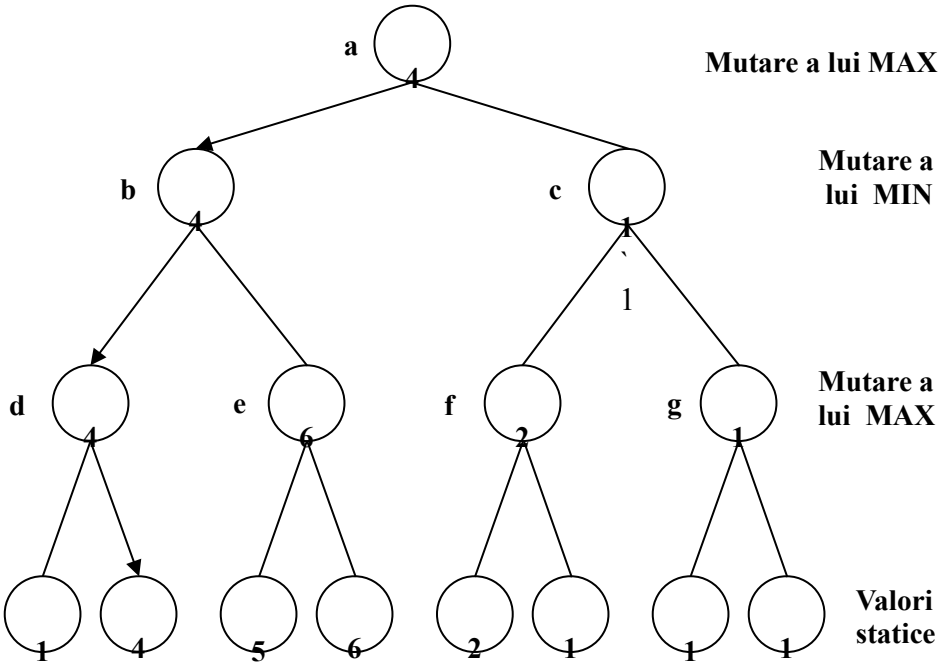

Fig. 3.1

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 părți. Se observă că valoarea pozițiilor de-a lungul variației 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 cerințele sale de spațiu sunt numai liniare în *m* şi *b*.

În cazul jocurilor reale, cerințele 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 porțiune 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 poziții 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 poziția inițială, nodul-rădăcină, la cel mai promițător succesor al său (conform acestor evaluări) este apoi efectuată în cadrul jocului.

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

Cea mai directă abordare a problemei deținerii controlului asupra cantității 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.

Funcții de evaluare

O *funcție de evaluare* întoarce o *estimație*, realizată dintr-o poziție dată, a utilității aşteptate a jocului. Ea are la bază evaluarea şanselor de câştigare a jocului de către fiecare dintre părți, pe baza calculării caracteristicilor unei poziții. Performanța unui program referitor la jocuri este extrem de dependentă de calitatea funcției de evaluare utilizate.

Funcția de evaluare trebuie să îndeplinească anumite condiții evidente: ea trebuie să concorde cu funcția 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 funcției de evaluare acoperă mai multe poziții diferite, grupate laolaltă într-o *categorie* de poziții 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 poziția de deschidere evaluarea este 0 şi toate pozițiile până la prima captură vor avea aceeaşi evaluare. Dacă MAX reuşeşte să captureze un nebun fără a pierde o piesă, atunci poziția rezultată va fi evaluată la valoarea 3. Toate pozițiile de acest fel ale lui MAX vor fi grupate într-o *categorie* etichetată cu "3". Funcția de evaluare trebuie să reflecte şansa ca o poziție aleasă la întâmplare dintr-o asemenea categorie să conducă la câştig (sau la pierdere sau la remiză) pe baza experienței anterioare.

Funcția 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 funcție de evaluare se numeşte *funcție liniară ponderată*, întrucât are o expresie de forma

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

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

În construirea formulei liniare trebuie mai întâi alese caracteristicile, operație urmată de ajustarea ponderilor până în momentul în care programul joacă suficient de bine. Această a doua operație 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 relație principală pe

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

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

minimax(Poz, SuccBun, Val) :

Poz este o poziție, Val este valoarea ei de tip minimax;

cea mai bună mutare de la Poz conduce la poziția SuccBun.

La rândul ei, relația

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

corespunde regulilor jocului care indică mutările "legale" (admise): ListaPoz este lista pozițiilor succesor legale ale lui Poz. Predicatul mutari va eşua dacă Poz

este o poziție de căutare terminală (o frunză a arborelui de căutare). Relația

`celmaibun(ListaPoz,PozBun,ValBuna)`

selectează cea mai bună poziție PozBun dintr-o listă de poziții 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 funcție 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).
```

O implementare eficientă a principiului Minimax: Algoritmul Alpha-Beta

Tehnica pe care o vom examina, în cele ce urmează, este numită în literatura de specialitate alpha-beta prunning ("alpha-beta retezare"). Atunci când este aplicată unui arbore de tip minimax standard, ea va întoarce aceeaşi mutare pe care ar furniza-o şi Algoritmul Minimax, dar într-un timp mai scurt, întrucât realizează *o retezare a unor ramuri ale arborelui care nu pot influența decizia finală*.

Principiul general al acestei tehnici constă în a considera un nod oarecare *n* al arborelui, astfel încât jucătorul poate alege să facă o mutare la acel nod. Dacă acelaşi jucător dispune de o alegere mai avantajoasă, *m*, fie la nivelul nodului părinte al lui *n*, fie în orice punct de decizie aflat mai sus în arbore, atunci *n* nu va fi niciodată atins în timpul jocului. Prin urmare, de îndată ce, în urma examinării unora dintre descendenții nodului *n*, ajungem să deținem suficientă informație relativ la acesta, îl putem înlătura.

Ideea tehnicii de alpha-beta retezare este aceea de a găsi o mutare "suficient de bună", nu neapărat cea mai bună, dar suficient de bună pentru a se lua decizia corectă. Această idee poate fi formalizată prin introducerea a *două limite*, *alpha* şi *beta*, reprezentând limitări ale valorii de tip minimax corespunzătoare unui nod intern.

Semnificația acestor limite este următoarea: *alpha* este *valoarea minimă* pe care este deja garantat că o va obține MAX, iar *beta* este *valoarea maximă* pe care MAX poate spera să o atingă. Din punctul de vedere al jucătorului MIN, beta este valoarea cea mai nefavorabilă pentru MIN pe care acesta o va atinge. Prin urmare, *valoarea efectivă* care va fi găsită se află *între alpha şi beta*.

Valoarea alpha, asociată nodurilor de tip MAX, nu poate niciodată să descrească, iar valoarea beta, asociată nodurilor de tip MIN, nu poate niciodată să crească.

Dacă, spre exemplu, valoarea alpha a unui nod intern de tip MAX este 6, atunci MAX nu mai trebuie să ia în cosideraţie nici o valoare internă mai mică sau egală cu 6 care este asociată oricărui nod de tip MIN situat sub el. Alpha este scorul cel mai prost pe care îl poate obţine MAX, presupunând că MIN joacă perfect. În mod similar, dacă MIN are valoarea beta 6, el nu mai trebuie să ia în consideraţie nici un nod de tip MAX situat sub el care are valoarea 6 sau o valoare mai mare decât acest număr.

Cele două reguli pentru încheierea căutării, bazată pe valori alpha şi beta, pot fi formulate după cum urmează:

1. **Căutarea poate fi oprită dedesubtul oricărui nod de tip MIN care are o valoare beta mai mică sau**

egală cu valoarea alpha a oricăruia dintre strămoşii săi de tip MAX.

2. **Căutarea poate fi oprită dedesubtul oricărui nod de tip MAX care are o valoare alpha mai mare sau egală cu valoarea beta a oricăruia dintre strămoşii săi de tip MIN.**

Dacă, referitor la o poziție, se arată că valoarea corespunzătoare ei se află în afara intervalului alpha-beta, atunci această informație este suficientă pentru a şti că poziția respectivă nu se află de-a lungul *variației principale*, chiar dacă nu este cunoscută valoarea exactă corespunzătoare ei. Cunoaşterea valorii exacte a unei poziții este necesară numai atunci când această valoare se află între alpha şi beta.

Din punct de vedere formal, putem defini o *valoare de tip minimax* a unui nod intern, P, V(P, alpha, beta), ca fiind "*suficient de bună*" dacă satisface următoarele cerințe:

V(P, alpha, beta) < alpha, dacă V(P) < alpha (1)

V(P, alpha, beta) = V(P), dacă alpha ≤ V(P) ≤ beta

V(P, alpha, beta) > beta, dacă V (P) > beta,

unde prin V(P) am notat valoarea de tip minimax corespunzătoare unui nod intern.

Valoarea exactă a unui nod-rădăcină P poate fi întotdeauna calculată prin setarea limitelor după cum urmează:

V(P, -∞, +∞) = V(P).

Fig. 3.2 ilustrează acțiunea Algoritmului Alpha-Beta în cazul arborelui din Fig. 3.1. Aşa cum se vede în figură, unele dintre valorile de tip minimax ale nodurilor interne sunt aproximative. Totuşi, aceste aproximări sunt suficiente pentru a se determina în mod exact valoarea rădăcinii. Se observă că Algoritmul Alpha-Beta reduce complexitatea căutării de la 8 evaluări statice la numai 5 evaluări de acest tip:

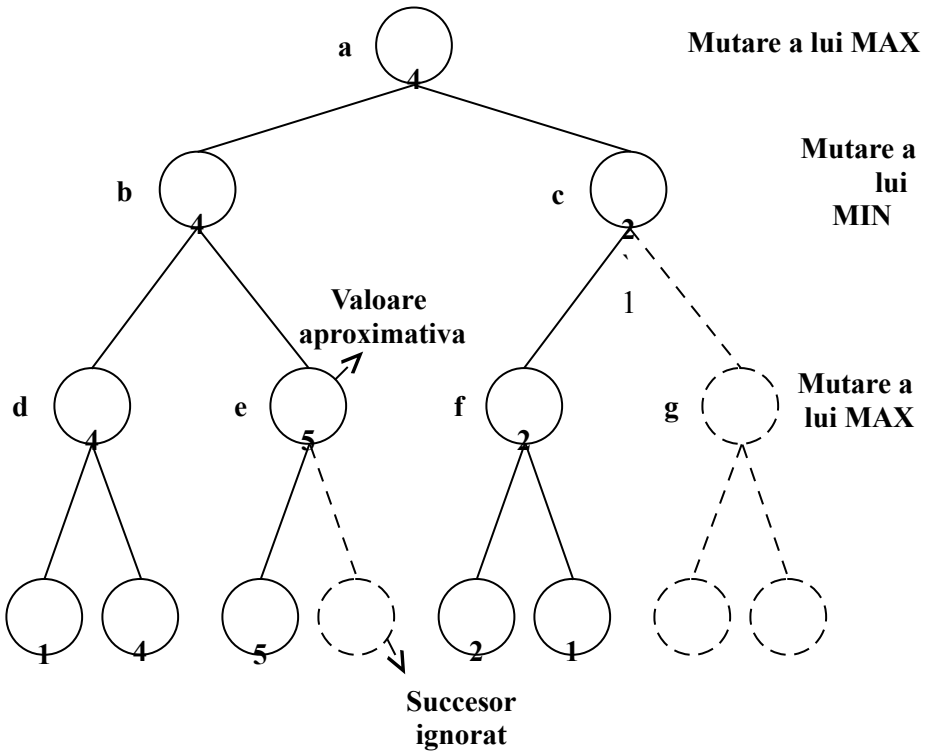
a
Mutare a lui MAX
Mutare a lui MIN
b
c
Valoare aproximativa
1
d
e
f
g
Mutare a lui MAX
Succesor ignorat

Fig. 3.2

Corespunzător arborelui din Fig. 3.2 procesul de căutare decurge după cum urmează:

1. **Începe din poziția *a*.**
2. **Mutare la *b*.**
3. **Mutare la *d*.**
4. **Alege valoarea maximă a succesorilor lui *d*, ceea ce conduce la V(*d*) = 4.**
5. **Întoarce-te în nodul *b* şi execută o mutare de aici la *e*.**
6. **Ia în considerație primul succesor al lui *e* a cărui valoare este 5. În acest moment, MAX, a cărui mutare urmează, are garantată, aflându-se în poziția *e*, cel puțin valoarea 5, indiferent care ar fi celelalte alternative plecând din *e*. Această informație este suficientă pentru ca MIN să realizeze că, la nodul *b*, alternativa *e* este inferioară alternativei *d*. Această concluzie poate fi trasă fără a cunoaşte valoarea *exactă* a lui *e*. Pe această bază, cel de-al doilea succesor al lui *e* poate fi neglijat, iar nodului *e* i se poate atribui valoarea *aproximativă* 5.**

Căutarea de tip alpha-beta retează nodurile figurate în mod discontinuu. Ca rezultat, câteva dintre valorile intermediare nu sunt exacte (nodurile *c*, *e*), dar aproximările făcute sunt suficiente pentru a determina atât valoarea corespunzătoare rădăcinii, cât şi variația principală, în mod exact.

Un alt exemplu de aplicare a Algoritmului Alpha-Beta este cel din Fig. 3.4. Astfel, corespunzător spațiului de stări din Fig. 3.3, alpha-beta retezarea produce arborele de căutare din Fig. 3.4, în care stările fără numere nu sunt evaluate.

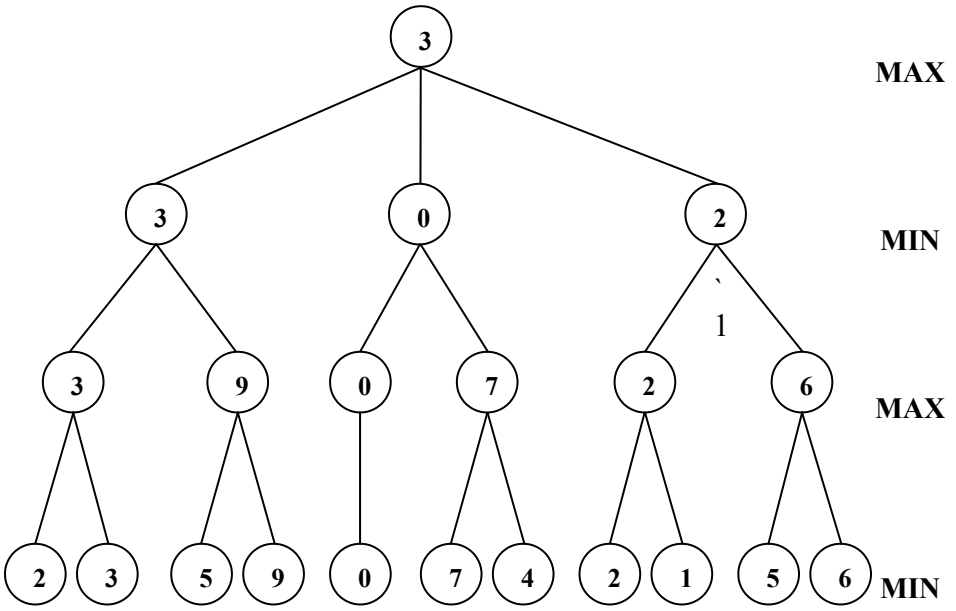

Fig. 3.3

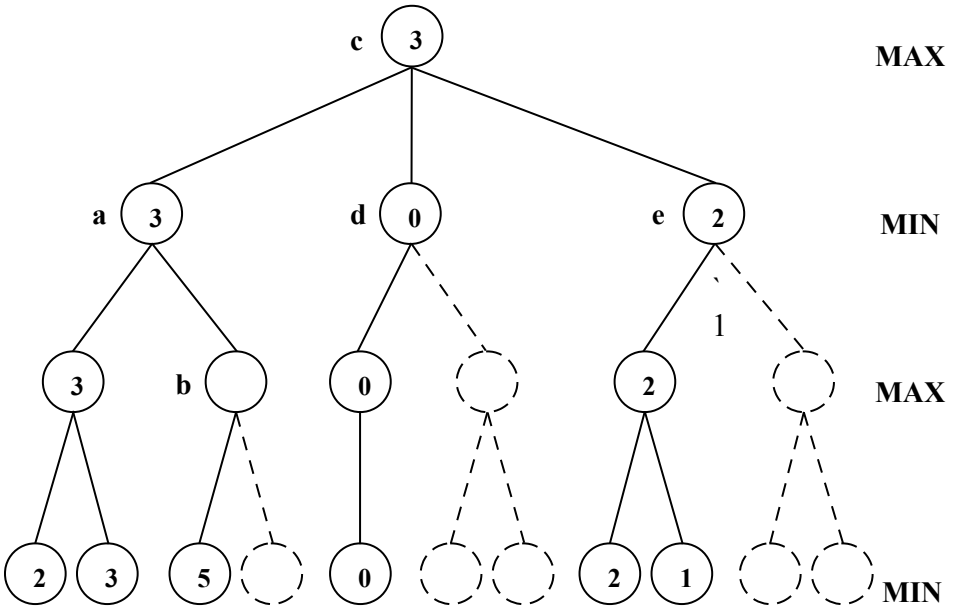

Fig. 3.4

<u>În Fig. 3.4</u>: ***a*** **are** $\beta = 3$ **(valoarea lui** ***a*** **nu va depăşi 3);**

***b* este *β* - retezat, deoarece 5 > 3;**

***c* are *α* = 3 (valoarea lui *c* nu va fi mai mică decât 3);**

***d* este *α* – retezat, deoarece 0 < 3;**

***e* este *α* – retezat, deoarece 2 < 3;**

***c* este 3.**

Implementare în Prolog

În cadrul unei implementări în Prolog a Algoritmului Alpha-Beta, relația principală este

```
alphabeta(Poz,Alpha,Beta,PozBuna,Val)
```

unde PozBuna reprezintă un succesor "suficient de bun" al lui Poz, astfel încât valoarea sa, Val, satisface cerințele (1):

$$Val = V(Poz, Alpha, Beta)$$

Procedura

```
limitarebuna(ListaPoz,Alpha,Beta,PozBuna,
Val)
```

găseşte, în lista ListaPoz, o poziție suficient de bună, PozBuna, astfel încât valoarea de tip minimax, Val, a lui PozBuna, reprezintă o aproximație suficient de bună relativ la Alpha şi Beta.

Intervalul alpha-beta se poate îngusta (dar niciodată lărgi) în timpul apelărilor recursive, de la o mai mare adâncime, ale procedurii.

Relația

```
limitenoi(Alpha,Beta,Poz,Val,AlphaNou,
BetaNou)
```

defineşte noul interval [AlphaNou, BetaNou]. Acesta este întotdeauna mai îngust sau cel mult egal cu vechiul interval

[Alpha, Beta]. Îngustarea intervalului conduce la operații suplimentare de retezare a arborelui.

Algoritmul Alpha-Beta

```
alphabeta(Poz,Alpha,Beta,PozBuna,Val ):-
      mutari(Poz,ListaPoz),!,
      limitarebuna(ListaPoz,Alpha,Beta,PozBuna,Val);
     %valoare statică a lui Poz
      staticval(Poz,Val).

limitarebuna([Poz|ListaPoz],Alpha,Beta,PozBuna,
ValBuna):-
      alphabeta(Poz,Alpha,Beta,_,Val),
      destuldebun(ListaPoz,Alpha,Beta,Poz,Val,PozBuna,
      ValBuna).

%nu există alt candidat

destuldebun([ ],_,_,Poz,Val,Poz,Val):-!.
destuldebun(_,Alpha,Beta,Poz,Val,Poz,Val):-
      %atingere limită superioară
      mutare_min(Poz),Val > Beta,!;
      %atingere limită inferioară
      mutare_max(Poz),Val < Alpha,!.
destuldebun(ListaPoz,Alpha,Beta,Poz,Val,PozBuna,
ValBuna):-
     %rafinare limite
      limitenoi(Alpha,Beta,Poz,Val,AlphaNou,BetaNou),
      limitarebuna(ListaPoz,AlphaNou,BetaNou,Poz1,
                   Val1),
      maibine(Poz,Val,Poz1,Val1,PozBuna,ValBuna).

limitenoi(Alpha,Beta,Poz,Val,Val,Beta):-
      %creştere limită inferioară
      mutare_min(Poz),Val > Alpha,!.
limitenoi(Alpha,Beta,Poz,Val,Alpha,Val):-
      %descreştere limită superioară
      mutare_max(Poz),Val < Beta,!.
```

```
%altfel,limitele nu se schimbă
limitenoi(Alpha,Beta,_,_,Alpha,Beta).

%Poz mai bună ca Poz1
maibine(Poz,Val,Poz1,Val1,Poz,Val):-
      mutare_min(Poz),Val > Val1,!;
      mutare_max(Poz),Val < Val1,!.
%altfel, Poz1 mai bună
maibine(_,_,Poz1,Val1,Poz1,Val1).
```

Considerații privitoare la eficiență

Eficiența Algoritmului Alpha-Beta depinde de *ordinea în care sunt examinați succesorii.* Este preferabil să fie examinați mai întâi succesorii despre care se crede că ar putea fi cei mai buni. În mod evident, acest lucru nu poate fi realizat în întregime. Dacă el ar fi posibil, funcția care ordonează succesorii ar putea fi utilizată pentru a se juca un joc perfect. În ipoteza în care această ordonare ar putea fi realizată, s-a arătat că Algoritmul Alpha-Beta nu trebuie să examineze, pentru a alege cea mai bună mutare, decât $O(b^{d/2})$ noduri, în loc de $O(b^d)$, ca în cazul Algoritmului Minimax. Aceasta arată că factorul de ramificare efectiv este $\sqrt{b}$ în loc de b – în cazul jocului de şah 6, în loc de 35. Cu alte cuvinte, Algoritmul Alpha-Beta poate "privi înainte" la o adâncime dublă față de Algoritmul Minimax, pentru a găsi acelaşi cost.

În cazul unei ordonări neprevăzute a stărilor în spațiul de căutare, Algoritmul Alpha-Beta poate dubla adâncimea spațiului de căutare (Nilsson 1980). Dacă există o anumită ordonare nefavorabilă a nodurilor, acest algoritm nu va căuta mai mult decât Algoritmul Minimax. Prin urmare, în cazul cel mai nefavorabil, Algoritmul Alpha-Beta nu va oferi nici un avantaj în comparație cu căutarea exhaustivă de tip minimax. În cazul unei ordonări favorabile însă, dacă notăm prin N numărul pozițiilor de căutare terminale evaluate în mod static de către Algoritmul Minimax, s-a arătat că, în cazul cel mai bun, adică atunci când mutarea cea mai puternică este prima luată în considerație, Algoritmul Alpha-Beta nu va evalua în mod static decât $\sqrt{N}$ poziții. În practică, o funcție de ordonare relativ simplă (cum ar fi încercarea mai întâi a capturilor, apoi a amenințărilor, apoi a mutărilor înainte, apoi a celor înapoi) ne poate apropia suficient de mult de rezultatul obținut în cazul cel mai favorabil.