Inteligenta Artificiala

Universitatea Politehnica Bucuresti

Anul universitar 2013-2014

Adina Magda Florea

Curs nr. 3

Strategii de rezolvare a problemelor

- Strategii de cautare locala
- Cautare cu actiuni nedeterministe
- Cautare on-line
- Strategii de cautare in jocuri

1 Cautari locale

- **Cautari locale** – opereaza asupra starii curente generand succesorii
- Calea catre solutie nu are importanta
- Gasire solutie – CSP - nu conteaza calitatea solutiei
- Gasire solutie optima – functie de evaluare sau de cost
- Probleme in gasirea solutiei optime pe baza de cautari locale in functie de forma spatiului de cautare
- Maxim global, maxim local, platou, umar

Cautari locale

Caracteristici

- Folosesc putina memorie
- Gasesc solutii destul de bune chiar si in spatii infinite
- Folosite si pt probleme de optimizare
- O cautare locala completa va gasi intotdeauna solutia daca aceasta exista
- Hill climbing
- Simulated annealing
- Local beam search

Algoritm: **Hill climbing**

/* intoarce o stare cu functia de evaluare un maxim local */

1. S ← stare initiala
2. Genereaza S_j=Succ(S), toti succesorii starii S
3. S' ← S_j cu Eval(S_j) maxim dintre toti succesorii S_j
4. **daca** Eval(S') ≤ Eval(S) **atunci intoarce** S
5. S ← S'
6. **repeta** de la 2

sfarsit

- Cautarea se orienteaza permanent in directia cresterii valorii;
- Se termina cand ajunge la un maxim (nici un succesor nu are valoarea mai mare)

Hill Climbing

- Problema 8 regine
- S – tabla completa
- Succesori – toate starile posibil de generat prin mutarea a 1 regina intr-un alt patrat in coloana
- o stare are 8 x 7 = 56 succesori
- Fct de evaluare = nr de perechi de regine care se ataca
- Minim global = 0
- 86% instante nu gaseste solutia (3 pasi)
- 14% gaseste soluta (4 pasi)
- Spatiul starilor $8^8 \approx 17$ milioane de stari
- Limitari

Hill Climbing - imbunatatiri

- Miscari laterale – se spera sa fie “umar” si nu “platou”
- Daca “platou” – risc de bucla infinita – necesita limita in cautare
- Cu aprox. 100 miscari laterale – problema 8 regine – 94% instante pt care se gaseste solutia

Hill Climbing - imbunatatiri

Hill climbing stohastic

- Dintre starile succesoare cu $Eval(S_j) \geq Eval(S)$, se alege aleator un S_j

First choice hill climbing

- Genereaza aleator succesori pana gaseste $Eval(S_j) \geq Eval(S)$, continua cautarea cu S_j

Random restart Hill Climbing

- Repeta diferite HC cu stari initiale generate aleator
- Daca fiecare HC are o probabilitate de succes de p $\rightarrow$ 1/p repetitii

Simulated annealing

- Simuleaza un proces fizic (calirea)
- T – temperatura
- Scaderea gradientului – solutii de cost minim
- Alege o miscare la intamplare
- Daca starea este mai buna, cauta in continuare de la aceasta
- Altfel alege starea mai "proasta" cu o probabilitate
- Scade probabilitatea de selectie a starilor mai "proaste" pe masura ce temperatura scade

Algoritm: **Cautare Simulated Annealing**

1. T ← temperatura initiala
2. S ← stare initiala
3. v ← Eval(S)
2. **cat timp** T > temp finala **executa**
   2.1 **pentru** i=1,n **executa**
   S' ← Succ(S)
   v' ← Eval(S')
   daca v' < v **atunci S** ← S'
   altfel S ← S' cu prob. exp(-(v'-v)/kT)
   (in rest S nemodificat)
   2.2 T ← 0.95 * T

sfarsit

Local beam search

- Tine minte k stari la un moment dat
- Incepe cu k stari generate aleator
- La fiecare pas genereaza succesorii tuturor starilor curente si alege dintre acestia pe cei mai buni k succesori cu care se continua cautarea
- Diferit de k algoritmi HC care ar rula in paralel

Cautari locale in spatii continue

- Factor de ramificare infinit
- First choice HC, Simulated annealing
- ***Problema:*** dorim sa plasam 3 supermarket-uri pe o harta a.i. suma patratelor distantelor de la fiecare comuna de pe harta la cel mai apropiat supermarket sa fie minima.
- Spatiul starilor este definit prin coordonatele supermarket-urilor

$$(x_1,y_1)\ (x_2,y_2)\ (x_3,y_3)$$

- (spatiu *n*-dimensional cu *n* variabile)
- O miscare in acest spatiu – miscarea unui sm pe harta
- C_i- multimea de orase care sunt cel mai aproape de sm ***i*** in starea curenta

$$f(x_1,y_1,x_2,y_2,x_3,y_3) = \Sigma_{i=1,3}\ \Sigma_{c\in Ci}\ (x_i-x_c)^2+(y_i-y_c)^2$$

Cautari locale in spatii continue

Cum putem gasi solutia?

VARIANTE

- Discretizare spatiu – discretizarea vecinatatii fiecarei stari

(de ex mutam cate un sm o data fie in directai x fie in directia y cu cate o cantitate fixa M => 12 stari succesoare)

- Utilizarea gradientului pt a gasi maximul
- Determinam gradientul lui f, global sau local $df/dx_1 = 2 \Sigma_{c \in Ci}(x_i - x_c)$

Si aplicam HC prin actualizarea starii curente cu x + constanta * gradientul

2. Cautare cu actiuni nedeterministe

Problema aspiratorului determinist

- Locatii A,B care pot fi curate (C) sau murdare (M)
- Actiuni agent: St, Dr, Aspira, (nimic)
- 2×2^2 stari posibile (2×2^n)
- M,M, $Agent^A \rightarrow_{\mathbf{Dr}}$ M,M, $Agent^B$
- M,M, $Agent^A \rightarrow_{\mathbf{St}}$ M,M, $Agent^A$
- M,M, $Agent^A \rightarrow_{\mathbf{Aspira}}$ C,M,$Agent^A$
- Stare initiala (M,M, $Agent^A$)
- Plan = [Aspira, Dr, Aspira]

Problema aspiratorului nedeterminist

Aspira nedeterminist

- daca Aspira in M atunci (C) sau (C si C patrat alaturat)
- daca Aspira in C atunci (C) sau (M)

- Stare initiala (M,M, $Agent^A$)
- Plan contingent =

  [Aspira,

  daca Stare = (C,M, $Agent^A$) **atunci** Dr, Aspira

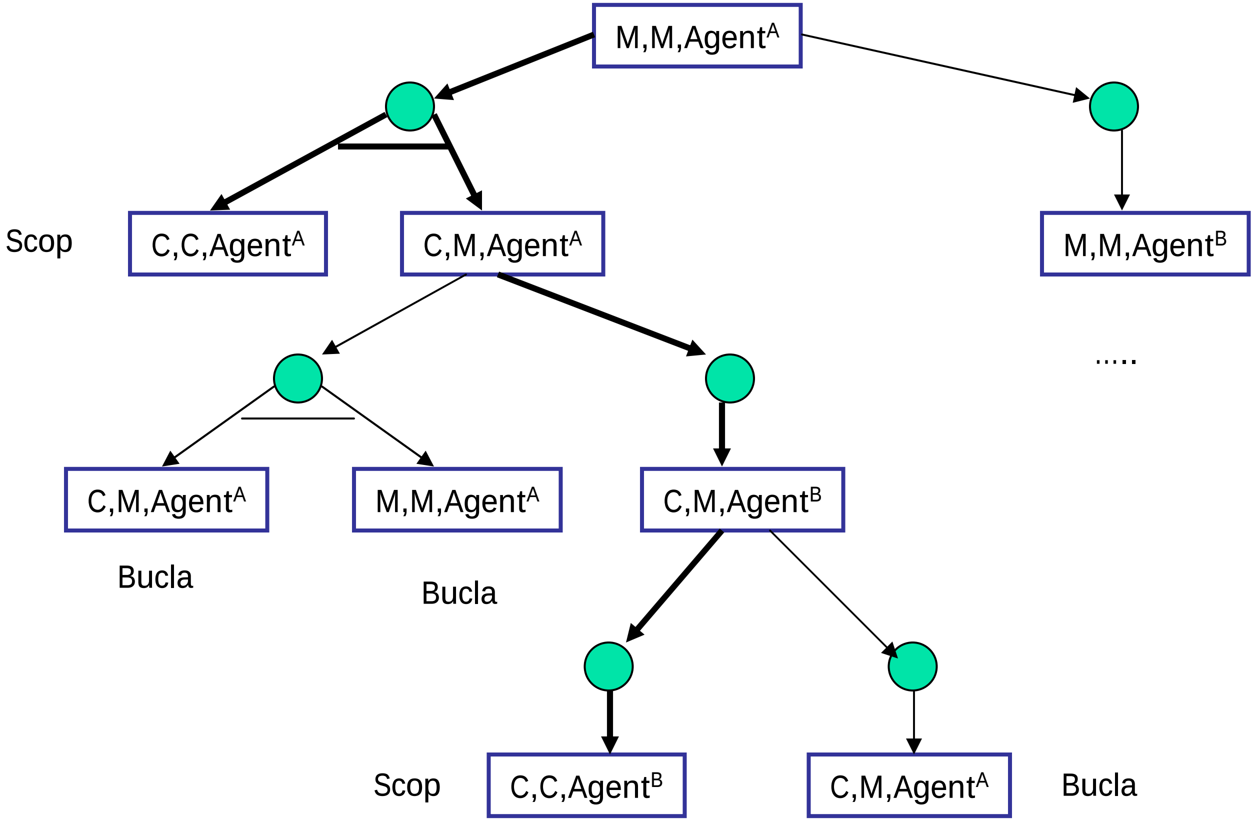
  altfel nimic]

- Planul – arbore SI/SAU

Problema aspiratorului nedeterminist

Solutie – un arbore SI/SAU:

- stare scop in fiecare frunza
- o actiune dintr-o ramura a unui nod SAU
- toate actiunile din ramurile unui nod SI



Plan contingent

Algoritm Plan: **Determina graf SI/SAU de actiuni**

1. Inspec-SAU(S_i,[])

/* intoarce plan contingent sau INSUCCES */

Inspec-SAU(S, Cale)

1. **daca** S este stare finala
   atunci intoarce Planul vid
2. **daca** $S \in$ Cale **atunci intoarce** INSUCCES
3. **pentru** fiecare actiune A_i posibil de executat din S **executa**
   3.1 Plan $\leftarrow$ Inspec-SI(Stari(S,A_i), [S|Cale])
   3.2 **daca** Plan $\neq$ INSUCCES **atunci intoarce** [A_i|Plan]
4. **intoarce** INSUCCES

sfarsit

Plan contingent

Inspec-SI(Stari, Cale)

1. **pentru** fiecare $S_i \in$ Stari **executa**

   1.1 $Plan_i \leftarrow$ Inspec-SAU(S_i, Cale)

   1.2 **daca** $Plan_i$ = INSUCCES

   atunci intoarce INSUCCES

2. **intoarce**

[**if** S_1 **then** $Plan_1$ **else** …**if** S_{n-1} **then** $Plan_{n-1}$ **else** $Plan_n$]

sfarsit

3. Cautare on-line

- Cautare “offline”
- **Cautare “on-line” – cautare + actiune**
- Exemplu: robot care investigheaza mediul
- **Cautare online:** pentru fiecare actiune, agentul primeste perceptia care ii spune in ce stare a ajuns.

  Construieste Rezultat[s,a]
- **Cautari locale online:** HC, nu pot random restart

  Random walk – selectez aleator o actiune din actunile disponibile in starea curenta
- Imbunatatim HC cu H(s) costul estimat de a ajunge la S_f din starea vizitata

3. Cautare on-line

- Programare dinamica asincrona
- Learning Real-Time A*
- Cautare cu tinta mobila

3.1 Programare dinamica asincrona (ADP)

Principiul optimalitatii – o cale este optima $\leftrightarrow$ orice segment (subcale) a acesteia este optima

- $S_i \rightarrow S_f$ si S pe aceasta cale
- $S_i \rightarrow S$, $S \rightarrow S_f$

Daca cunoaste **h*** pt fiecare nod, calea de cost minim poate fi obtinuta repetand procedura:

- Pentru fiecare nod succesor **j** a nodului curent **i** **calculeaza f*(j)=c(i,j) + h*(j)**
- Mergi la starea j pt care **f*(j) este minim**

Programare dinamica asincrona

Presupunem urmatoarea situatie

- Pentru fiecare nod **i** exista un proces care corespunde lui **i**
- Fiecare proces inregistreaza **h(i)** – estimarea lui **h*(i)**
- Valoarea initiala a lui **h(i)** este arbitrara cu exceptia nodurilor stare finala
- Fiecare proces **i** actualizeaza **h(i)** pentru fiecare vecin **j - calculeaza** $f(j)=c(i,j) + h(j)$
- $h(i) \leftarrow \min_j f(j)$

Programare dinamica asincrona

- Ordinea de actualizare este arbitrara
- Daca costul arcelor este pozitiv, converge la valorile reale
- Spatiu de stari mare – nepractic
- Foloseste ca fundament

3.2 Learning Real-Time A* (LRTA*)

Agentul considera numai nodul curent

Agent in nodul **i**

Agentul inregistreaza distanta estimata pentru fiecare nod

1. Lookahead

- Pentru fiecare vecin **j** a lui **i calculeaza f(j)=c(i,j) + h(j)**

h(j) – estimarea caii $j \rightarrow S_f$

c(i,j) – cost arc $i \rightarrow j$

Learning Real-Time A* (LRTA*)

2. Actualizeaza h

- Actualizeaza h estimat a nodului **i**

$$\mathbf{h(i) \leftarrow min_j\, f(j)}$$

3. Selecteaza actiunea si memoreaza

- Mergi la starea **j** cu valoarea **minima f(j)** – actiunea ***a***
- Memoreaza $i \rightarrow_{\boldsymbol{a}} j$

Valorile initiale ale lui **h** trebuie sa fie admisibile

Learning Real-Time A* (LRTA*)

Algoritm LRTA*(s') intoarce o actiune

intrari: s', o perceptie care identifica starea curenta

variabile globale: Rezultat: o tabela indexata dupa stari si actiuni, initial vida

H – o tabela cu estimarea starilor, initial vida

s, a – starea anterioara si actiunea anterioara, initial nule

daca s' este stare scop **atunci intoarce** stop

daca s' este stare noua (s' $\notin$ H) **atunci** H[s'] $\leftarrow$ h(s')

daca s $\neq$ null **atunci**

Rezultat[s,a] $\leftarrow$ s'

H[s] $\leftarrow$ min $_{b \in \text{Actiuni(s)}}$ LRTA*-Cost(s, b, Rezultat[s,b], H)

a $\leftarrow$ o actiune b din Actiuni(s') care minimizeaza LRTA*-Cost(s', b, Rezultat[s',b], H)

s $\leftarrow$ s'

intoarce a

Algoritm LRTA*-Cost(s,a,s',H) intoarce o estimare de cost

daca s' nu este definita **atunci intoarce** h(s)

altfel intoarce c(s,a,s') + H[s']

Learning Real-Time A* (LRTA*)

- Intr-un spatiu de cautare finit cu costuri pozitive, in care exsita o cale de la orice stare S la S_f si se utilizeaza estimari admisibile nenegative, LRTA* este complet
- In plus « invata » solutia optima in timp

3.3 Cautare cu tinta mobila (MTS)

- Starea scop se schimba pe parcursul cautarii
- MTS – generalizare a LRTA*
- MTS trebuie sa obtina informatie despre locatia starii scop
- Matrice de valori **h(x,y)**
- 2 evenimente, fiecare face o actualizare a valorii euristicilor:
  - Miscare a PS
  - Miscare a T

Cautare cu tinta mobila (MTS)

- Presupunem ca PS si T se misca alternativ
- Stare finala (scop) = PS si T au aceeasi pozitie
- x_i,x_j – pozitia curenta si pozitia vecinilor pt PS
- y_i, y_j – pozitia curenta si pozitia vecinilor pt T
- Presupun ca toate arcele au ***cost 1***

Cautare cu tinta mobila (MTS)

Miscare PS

- Calculeaza $h(x_j,y_i)$ pentru fiecare vecin x_j a lui x_i
- Actualizeaza valoarea $h(x_i,y_i)$ astfel

$$h(x_i,y_i) = \max\{h(x_i,y_i), \min_j(h(x_j,y_i)+1)\}$$

- Miscare la x_j cu $h(x_j,y_i)$ minim

$$x_i \leftarrow x_j$$

Miscare T

- Calculeaza $h(x_i,y_j)$ pentru noua pozitie y_j a lui T
- Actualizeaza valoarea $h(x_i,y_i)$ astfel

$$h(x_i,y_i) = \max\{h(x_i,y_i), h(x_i,y_j)-1\}$$

- Actualizeaza starea scop cu noua pozitie a lui T

$$y_i \leftarrow y_j$$

Cautare cu tinta mobila (MTS)

- Intr-un spatiu de cautare finit cu costuri pozitive in care exista o cale de la fiecare stare S la starea scop S_f, daca se porneste cu valori ale functiei euristice admisibile si se permit miscari ale PS si T in orice directie cu cost unitar, PS care executa MTS va ajunge la T daca T sare periodic peste miscari.

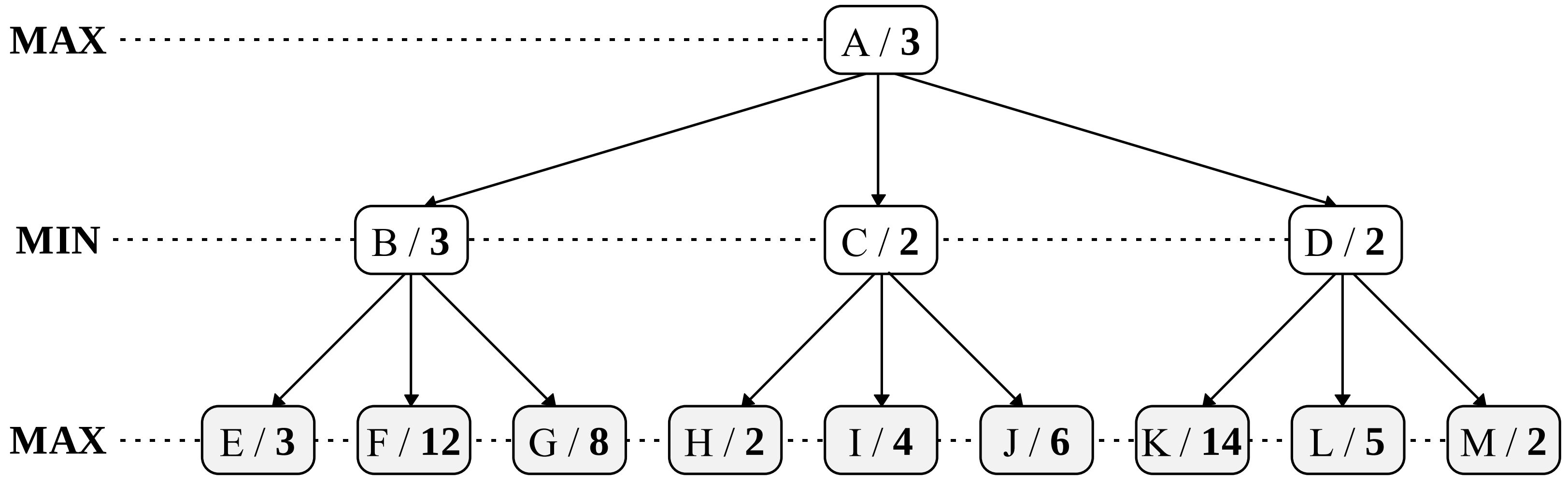
4. Strategii de cautare in jocuri

- Jocuri ce implică doi adversari
  - jucator
  - adversar
- Jocuri in care spatiul de cautare poate fi investigat exhaustiv
- Jocuri in care spatiul de cautare nu poate fi investigat complet deoarece este prea mare.
- Algoritmul Minimax (von Neumann 1928)

4.1 Minimax pentru spatii de cautare investigate exhaustiv

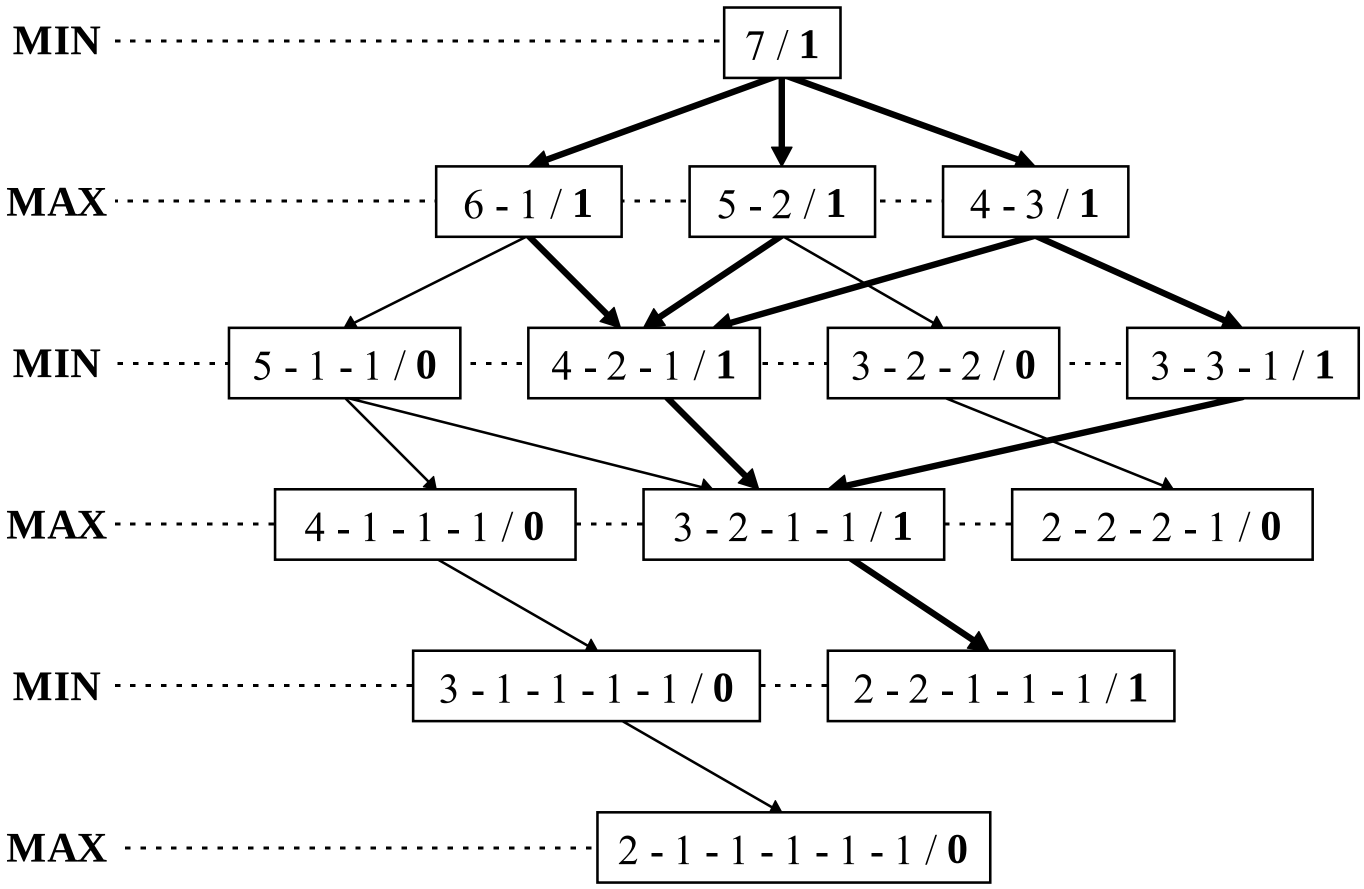
- Jucator – MAX
- Adversar – MIN
- Principiu Minimax
- Etichetez fiecare nivel din AJ cu **MAX** (jucator) si **MIN** (adversar)
- Etichetez frunzele cu scorul jucatorului
- Parcurg AJ
  - daca nodul parinte este **MAX** atunci i se atribuie valoarea maxima a succesorilor sai;
  - daca nodul parinte este **MIN** atunci i se atribuie valoarea minima a succesorilor sai.

Spatiu de cautare Minimax (AJ)



Spatiu de cautare Minimax (AJ)

Nim cu 7 bete



Algoritm: **Minimax cu investigare exhaustiva**

AMinimax(S)

1. pentru fiecare succesor S_j al lui S (obtinut printr-o mutare op_j) **executa**

val(S_j) $\leftarrow$ Minimax(S_j)

2. aplica op_j pentru care val(S_j) este maxima

sfarsit

Minimax(S)

1. daca S este nod final **atunci intoarce** scor(S)

2. altfel

2.1 **daca** MAX muta in S **atunci**

2.1.1 **pentru** fiecare succesor S_j al lui S **executa**

val(S_j) $\leftarrow$ Minimax(S_j)

2.1.2 **intoarce max**(val(S_j), $\forall$j)

2.2 **altfel** { MIN muta in S }

2.2.1 **pentru** fiecare succesor S_j al lui S **executa**

val(S_j) $\leftarrow$ Minimax(S_j)

2.2.2 **intoarce min**(val(S_j), $\forall$j)

sfarsit

4.2 Minimax pentru spatii de cautare investigate pana la o adancime *n*

- Principiu Minimax
- Algoritmul Minimax pana la o adancime *n*
- *nivel(S)*
- O functie euristica de evaluare a unui nod *eval(S)*

Algoritm: **Minimax cu adancime finita *n***

AMinimax(S)

1. pentru fiecare succesor S_j al lui S (obtinut printr-o mutare op_j) **executa**

val(S_j) $\leftarrow$ Minimax(S_j)

2. aplica op_j pentru care val(S_j) este maxima

sfarsit

Minimax(S) { intoarce o estimare a starii S }

0. daca S este nod final **atunci intoarce** scor(S)

1. daca nivel(S) = n **atunci intoarce** **eval(S)**

2. altfel

2.1 **daca** MAX muta in S **atunci**

2.1.1 **pentru** fiecare succesor S_j al lui S **executa**

val(S_j) $\leftarrow$ Minimax(S_j)

2.1.2 **intoarce** **max**(val(S_j), $\forall j$)

2.2 **altfel** { MIN muta in S }

2.2.1 **pentru** fiecare succesor S_j al lui S **executa**

val(S_j) $\leftarrow$ Minimax(S_j)

2.2.2 **intoarce** **min**(val(S_j), $\forall j$)

sfarsit

Implementare Prolog

```
play:-
	initialize(Position,Player),
	display_game(Position,Player),
	play(Position,Player,Result).

% play(+Position,+Player,-Result)
play(Position, Player, Result) :-
	game_over(Position,Player,Result), !, write(Result),nl.

play(Position, Player, Result) :-
	choose_move(Position,Player,Move),
	move(Move,Position,Position1),
	next_player(Player,Player1),
	display_game(Position1,Player1),
	!, play(Position1,Player1,Result).

% apel   ?-play.
```

```
move(a1,a,b).
move(a2,a,c).
move(a3,a,d).
move(b1,b,e).
move(b2,b,f).
move(b3,b,g).
move(c1,c,h).
move(c2,c,i).
move(c3,c,j).
move(d1,d,k).
move(d2,d,l).
move(d3,d,m).

next_player(max,min).
next_player(min,max).

initialize(a,max).
display_game(Position,Player):-
        write(Position),nl,write(Player),nl.

game_over(e,max,3).
game_over(f,max,12).
game_over(g,max,8).
game_over(h,max,2).
game_over(i,max,4).
game_over(j,max,6).
game_over(k,max,14).
game_over(l,max,5).
game_over(m,max,2).

% move(+Move,+Position,-Position1)

% game_over(+Position,+Player,
                        -Result).

% next_player(+Player, - Player1)
```

```
% choose_move(+Position, +Player, -BestMove)
choose_move(Position, Player, BestMove):-
    get_moves(Position,Player,Moves),
    evaluate_and_choose(Moves,Position,10,Player,Record,[BestMove,_]).

% get_moves(+Position, +Player, -Moves)
get_moves(Position, Player, Moves):-
    findall(M,move(M,Position,_),Moves).

% evaluate_and_choose(+Moves, +Position,+D,+MaxMin,
                                          +Record,-BestRecord).
evaluate_and_choose([Move|Moves],Position,D,MaxMin,Record,BestRecord)
    :-
    move(Move,Position,Position1),
    next_player(MaxMin, MinMax),
    minimax(D,Position1,MinMax,Value),
    update(MaxMin,Move,Value,Record,Record1),
    evaluate_and_choose(Moves,Position,D,MaxMin,Record1,BestRecord).

evaluate_and_choose([], Position, D, MaxMin, BestRecord, BestRecord).
```

```
% minimax(+Depth,+Position,+MaxMin,-Value)

minimax(_, Position, MaxMin, Value) :-
        game_over(Position,MaxMin,Value),!.

minimax(0, Position, MaxMin, Value) :-
        eval(Position,Value),!.

minimax(D, Position, MaxMin, Value) :-
        D > 0, D1 is D-1,
        get_moves(Position,MaxMin,Moves),
       evaluate_and_choose(Moves, Position, D1, MaxMin, Record,
                                                 [BestMove,Value]).
```

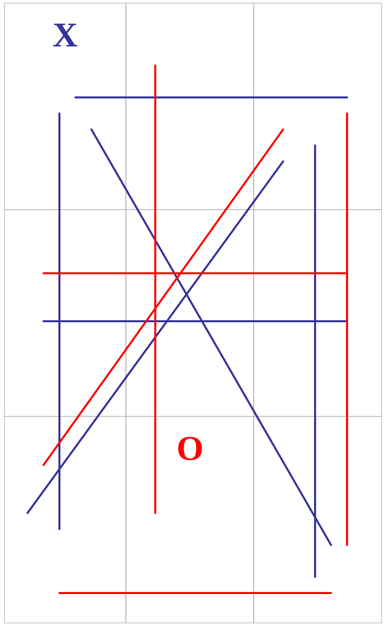
```
% update(+MaxMin, +Move, +Value, +Record, -Record1)

update(_, Move, Value, Record, [Move,Value]) :-
        var(Record),!.

update(max, Move, Value, [Move1,Value1], [Move1,Value1]) :-
         Value =< Value1.
update(max, Move, Value, [Move1,Value1],  [Move,Value]) :-
        Value > Value1.

update(min, Move, Value, [Move1,Value1], [Move1,Value1]) :-
         Value > Value1.
update(min, Move, Value, [Move1,Value1], [Move,Value]) :-
        Value =< Value1.
```

Exemplu de functie de evaluare

Jocul de Tic-Tac-Toe (X si O)

- Functie de estimare euristica **eval(S)** - conflictul existent in starea **S**.
- eval(**S**) = numarul total posibil de linii castigatoare ale lui **MAX** in starea **S** - numarul total posibil de linii castigatoare ale lui **MIN** in starea **S**.
- Daca **S** este o stare din care **MAX** poate face o miscare cu care castiga, atunci eval(**S**) = ∞ (o valoare foarte mare)
- Daca **S** este o stare din care **MIN** poate castiga cu o singura mutare, atunci eval(**S**) = -∞ (o valoare foarte mica).

eval(S) in Tic-Tac-Toe



X are 6 linii castigatoare posibile

O are 5 linii castigatoare posibile

eval(**S**) = 6 - 5 = 1

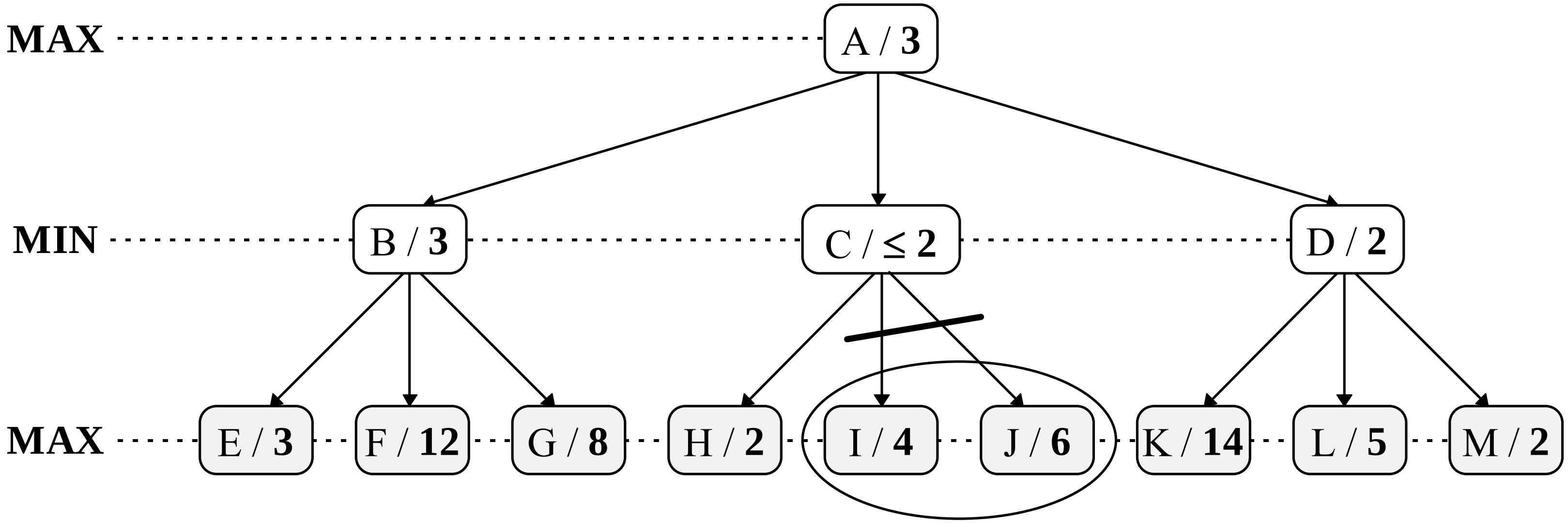
4.3 Algoritmul taierii alfa-beta

- Este posibil sa se obtină decizia corecta a algoritmului **Minimax** fara a mai inspecta toate nodurile din spatiului de cautare pana la un anumit nivel.
- Procesul de eliminare a unei ramuri din arborele de cautare se numeste *taierea arborelui de cautare* (*pruning*).
- Alpha-beta pruning (Knuth and Moore, 1975)

Algoritmul taierii alfa-beta

- Fie α cea mai buna valoare (cea mai mare) gasita pentru **MAX** si β cea mai buna valoare (cea mai mica) gasita pentru **MIN**.
- Algoritmul **alfa-beta** actualizeaza α si β pe parcursul parcurgerii arborelui si elimina investigarile subarborilor pentru care α sau β sunt mai proaste.
- Terminarea cautarii (taierea unei ramuri) se face dupa doua reguli:
  - Cautarea se opreste sub orice nod **MIN** cu o valoare β mai mica sau egala cu valoarea α a oricaruia dintre nodurile **MAX** predecesoare nodului **MIN** in cauza.
  - Cautarea se opreste sub orice nod **MAX** cu o valoare α mai mare sau egala cu valoarea β a oricaruia dintre nodurile **MIN** predecesoare nodului **MAX** in cauza.

Tăierea alfa-beta a spațiului de căutare



Algoritm: **Alfa-beta**

MAX(S, α, β) { intoarce valoarea maxima a unei stari. }

0. daca S este nod final **atunci intoarce** scor(S)

1. daca nivel(S) = n **atunci** intoarce eval(S)

2. altfel

2.1 **pentru** fiecare succesor S_j al lui S **executa**

2.1.1 $\alpha \leftarrow \max(\alpha, \text{MIN}(S_j, \alpha, \beta))$

2.1.2 **daca** $\alpha \geq \beta$ **atunci** intoarce β

2.2 **intoarce** α

sfarsit

MIN(S, α, β) { intoarce valoarea minima a unei stari. }

0. daca S este nod final **atunci intoarce** scor(S)

1. daca nivel(S) = n **atunci intoarce** eval(S)

2. altfel

2.1 **pentru** fiecare succesor S_j al lui S **executa**

2.1.1 $\beta \leftarrow \min(\beta, \text{MAX}(S_j, \alpha, \beta))$

2.1.2 **daca** $\beta \leq \alpha$ **atunci intoarce** α

2.2 **intoarce** β

sfarsit

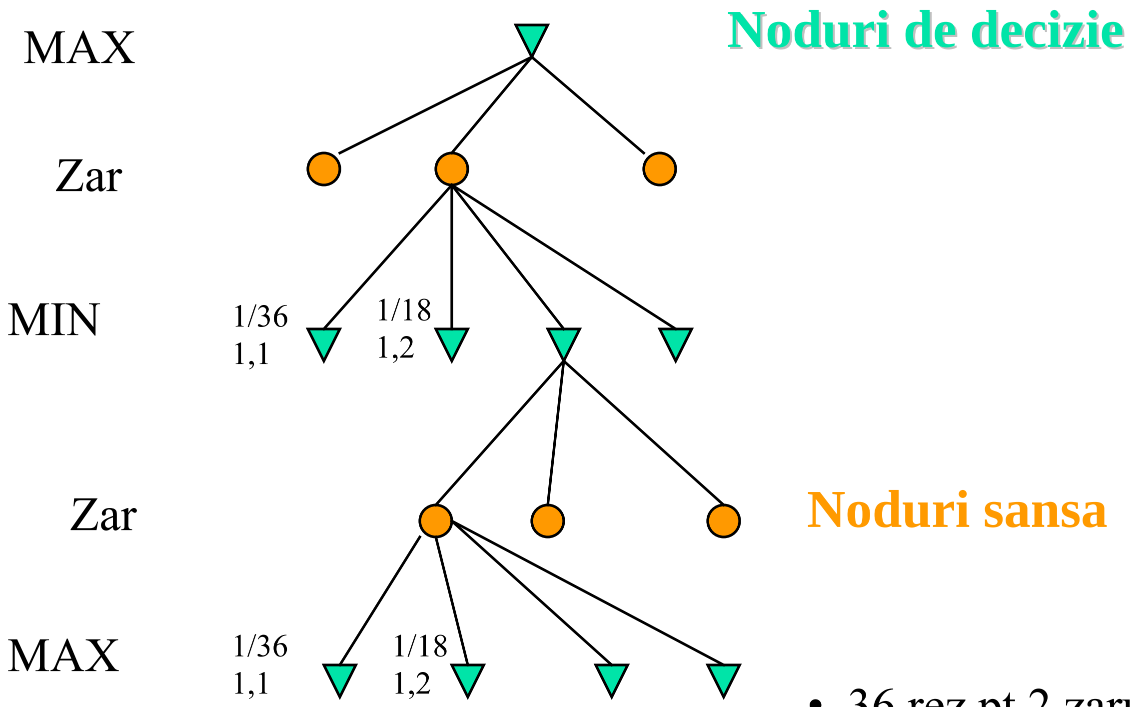
Algoritmul taierii alfa-beta

- Eficienta algoritmului depinde semnificativ de ordinea de examinare a starilor
- Se recomanda o ordonare euristica a succesorilor, evetual de generat numai primii cei mai buni succesori
- Poate reduce semnificativ timpul de cautare
- Table look-up – imbunatatire semnificativa prin memorarea pozitiilor curente

4.4 Jocuri cu elemente de sansa

- Jucatorul nu cunoaste miscarile legale ale oponentului
- 3 tipuri de noduri:
  - MAX
  - MIN
  - Sansa (chance nodes)

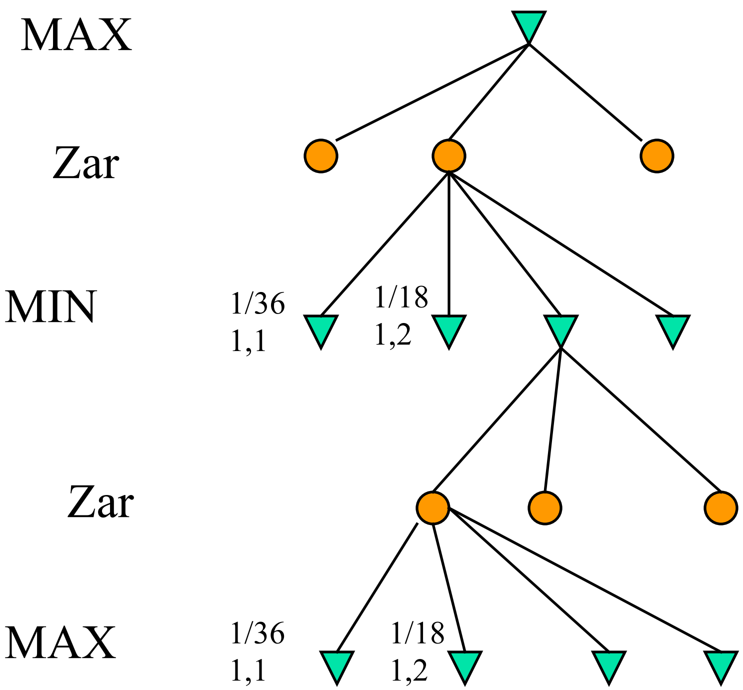
Noduri sansa – ramurile care pleaca dintr-un nod sansa indica posibile rezultate ale sansei (de exemplu zar)



- 36 rez pt 2 zaruri, toate la fel de probabile
- 21 noduri distincte (3-4 la fel cu 4-3)
- Zaruri egale (6 dist) - > 1/36 pt 1-1
- Zaruri diferite (15 dist) -> 1/18 pt zaruri diferite

Functia de evaluare

- scor – nod terminal
- max din Minimax succesori - MAX
- min din Minimax succesori - MIN
- Σ [P(S_j)*Minimax(S_j)] succesori - SANSA



Valoarea estimata pt noduri sansa

- $\Sigma_{S_j \text{ suc } S}$[P($S_j$)*Minimax($S_j$)]

Noduri sansa

Noduri de decizie

4.5 Monte Carlo Tree Search

- MCTS - Este un fel de best-first search ghidat de rezultatele unei simulari Monte-Carlo
- Metoda se bazeaza pe 2 ipoteze:
  - Adevarata **valoare a unei actiuni** (mutare in joc) poate fi aproximata utilizand simulari aleatoare
  - Valorile astfel obtinute pot fi utilizate pentru a **ajusta politica de selectie** spre o cea mai buna strategie
- MCTS (Coulom, 2006)

Monte Carlo Tree Search

- Baza metodei este o **unda de joc** (“**playout**”)
- **Playout** = un joc rapid jucat cu mutari aleatoare dintr-o anumita stare pana la sfarsitul jocului, obtinandu-se castig/pierdere sau un scor
- O unda poate fi “usoara” sau “grea”
- Fiecarui nod i se asociaza un procent de castig = de cate ori s-a castigat daca s-a pornit unda din acel nod

Monte Carlo Tree Evaluation

- Precursor – **Monte-Carlo Evaluation** (MCE) – utilizat in table (1997), poker, bridge, Go (2003)
- Evalueaza o pozitie de joc P prin simulare Monte Carlo si utilizeaza evaluarile intr-un algoritm Alfa-Beta
- Daca R_i este rezultatul unei simulari, evaluarea pozitiei P este

$$E_n(P)=(1/n)*\sum R_i$$

- Se poate demonstra ca daca valorile R_i sunt limitate, $E_n(P)$ converge la o valoare fixa daca *n* tinde la infinit

Monte Carlo Tree Evaluation

- Cam dificil de aplicat in practica
- Pt a evalua 1,000,000 de noduri la sah, presupunand 100,000 simulari/secunda si 10,000 simulari pe evaluare => aprox. 28h
- Evaluare lenesa – daca este cu o probabilitate de 95% mai mica decat α sau mai mare decat β, se opreste

Monte Carlo Tree Search

- Algoritmul construieste progresiv un **arbore de joc partial**, ghidat de rezultatele explorarilor anterioare ale acestui arbore
- Arborele este utilizat pentru a **estima valoarea miscarilor**, estimarile devenind din ce in ce mai bune pe masura ce arborele este construit
- Algoritmul implica construirea iterativa a arborelui de cautare pana cand o anumita cantitate de efort s-a atins, si intoarce cea mai buna actiune gasita

Monte Carlo Tree Search

Pentru fiecare iteratie se aplica 4 pasi:

- **Selectie** – Pornind de la radacina o politica de selectie a copiilor este aplicata recursiv pana cand gasesc cel mai interesant nod neexpandat (un nod E care are un copil ce nu este inca parte a arborelui)
- **Expandare** – Unul sau mai multe noduri copii a lui E sunt adaugate in arbore cf. actiunilor disponibile
- **Simulare** – Se executa o simulare de la nodul/nodurile noi cf. politicii implicite pentru a obtine un rezultat R
- **Backpropagation** – rezultatul simulatii este propagat inapoi catre nodurile care au fost parcurse si se actualizeaza valorile acestora

Monte Carlo Tree Search

Algoritm MCTS(radacina) intoarce cea mai buna miscare

T – arborele

Nc(nod) – multimea de copii a nodului N

R – rezultatul, poate fi 1, -1 sau 0

1. cat timp nu s-au epuizat resursele **repeta**

1.1. nod_curent ← radacina

1.2. **cat timp** nod curent∈T **repeta** /* traverseaza arborele */

ultimul_nod ← nod_curent

nod_curent ← Selectie(nod_curent)

1.3. ultimul_nod ← Expandeaza(ultimul_nod) /* adauga un nod */

1.4. R ← Joaca_joc_simulat(ultimul_nod) /* joaca un joc simulat */

1.5. nod_curent ← ultimul_nod /* se propaga rezultatul */

1. 6. **cat timp** nod_curent ∈ T **repeta**

Backprop(nod_curent, R)

nod_curent ← Parinte(nod_curent)

2. intoarce cea_mai_buna_mutare ← $\text{argmax}_{N \in Nc}$(radacina)

Monte Carlo Tree Search

Ce strategii se folosesc pt fiecare pas?

- **Selectia** – multe strategii propuse

Fie I multimea de noduri la care se poate ajunge din nodul curent p. Se selecteaza copilul K a nodului p care satisface formula

$$K \in \arg\max_{i \in I} \left(v_i + C * \sqrt{\frac{\ln n_p}{n_i}}\right)$$

v_i - este valoarea nodului I

N_i – numarul de vizitari a nodului i

N_p – numarul de vizitari a nodului p

C – coeficient experimental

Monte Carlo Tree Search

Expandarea

Prima pozitie care nu a fost deja memorata

Simularea

- Total aleator sau pseudoaleator sau combinat cu euristici

Backpropagation

- Diferite metode

$$V_p = \frac{V_{med} * W_{med} + V_r * N_r}{W_{med} + N_r}$$

V_r – mutarea cu cel mai mare numar de simulari

N_r – numarul de ori de care s-a jucat V_r

V_{med} – media valorilor nodurilor copii

W_{med} – ponderea acestei medii

Monte Carlo Tree Search

MoGo

A participat in 30 de turnee intre 2006 si 2010

A castigat contra profesionistilor

- MoGo invinge pe campionul Myungwan Kim, august 2008, utilizand MCTS