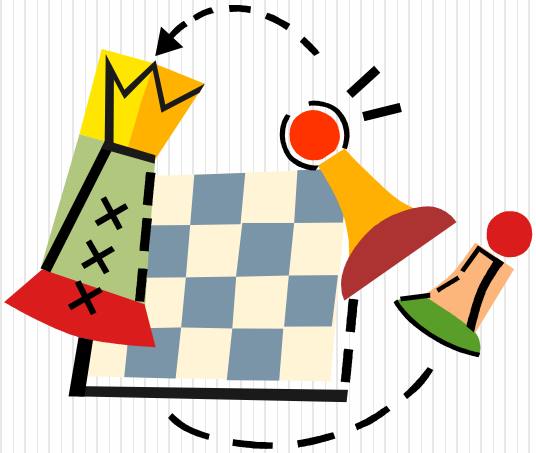
Teoria jocurilor



Catalin Stoean

catalin.stoean@inf.ucv.ro

http://inf.ucv.ro/~cstoean

Jocurile ca probleme de cautare

- Jocurile sunt fascinante, iar scrierea de programe care sa le joace este chiar si mai fascinanta!
- Se poate spune ca teoria jocurilor este pentru inteligenta artificiala cum este Grand Prix-ul pentru constructorii de motoare.
- Teoria jocurilor este unul dintre primele domenii ale inteligentei artificiale.
  - Primele programe pentru jocul de sah au fost scrise in 1950 de catre Claude Shannon si de catre Alan Turing.
  - De atunci, programele pentru jocuri s-au imbunatatit gradual, ajungand azi sa concureze direct cu campioni mondiali fara a se face de ras.

Jocurile ca probleme de cautare

- Un computer care joaca sah este dovada unei masini care face ceva ce necesita inteligenta.
- Ideea de baza este de a trata problemele care apar atunci cand planificam in avans intr-o lume care include si un **agent ostil**.
- Starea unui joc este de obicei usor de reprezentat, iar agentii sunt restrictionati la un numar limitat de actiuni bine definite.
- Faptul ca exista reguli care trebuie urmate si lumea starilor este complet **accesibila** pentru program face ca jocul sa poata fi usor reprezentat si face posibila cautarea prin spatiul diverselor stari ale jocului.

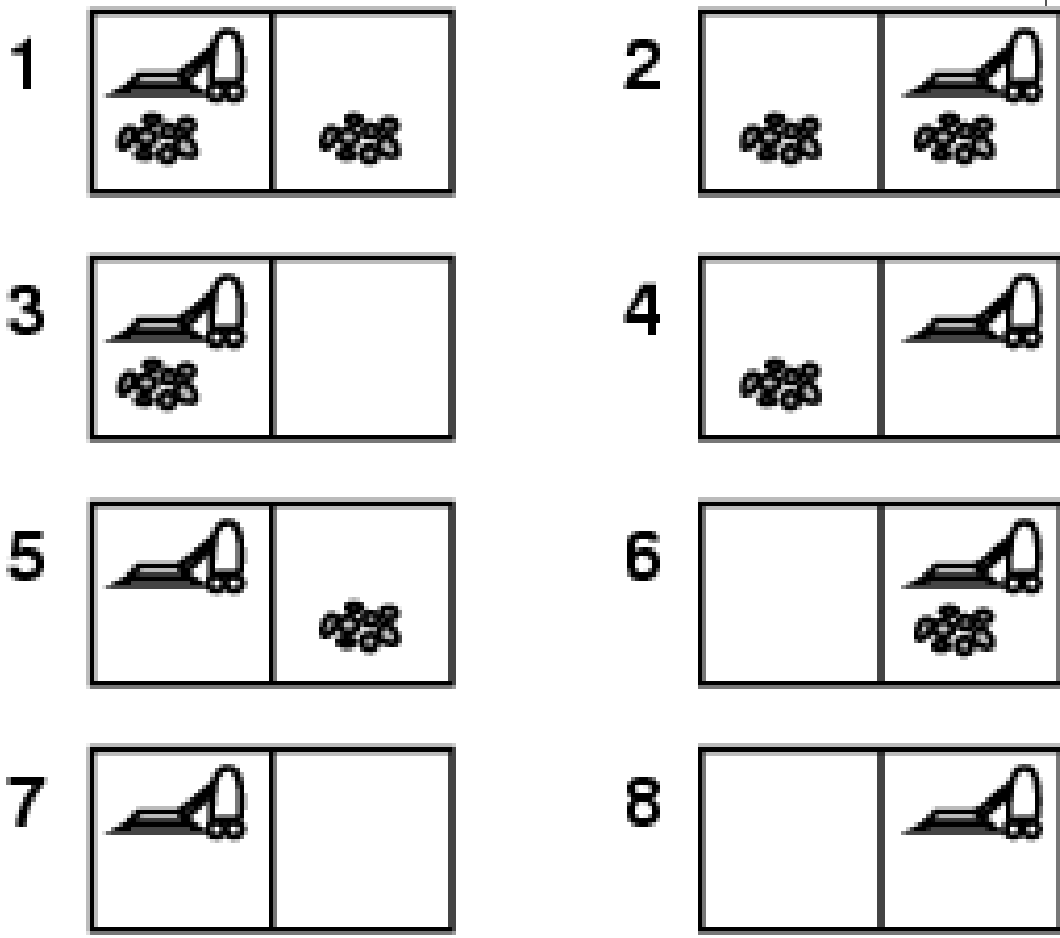
Jocurile ca probleme de cautare

- Prezenta unui adversar face ca problema de decizie sa fie mai complicata decat problemele de cautare tratate anterior.
- Adversarul este cel care aduce o **incertitudine** pentru ca nu se stie ce decizie va lua acesta.
- In esenta, toate programele facute pentru jocuri au de a face cu **probleme contingente**.
  - Problemele contingente se mai numesc si **nedeterministe**.
  - Perceptorii aduc informatie noua despre starea curenta.

Amintim!

Problema contingenta (exemplu)

- Presupunem ca mediul este nedeterminist.
- Legile lui Murphy guverneaza mediul
  - aspirarea duce la depozitarea murdariei intr-un loc… care era complet curat…
- De exemplu, in starea 4, daca aspira se poate ajunge la 2 sau 4.



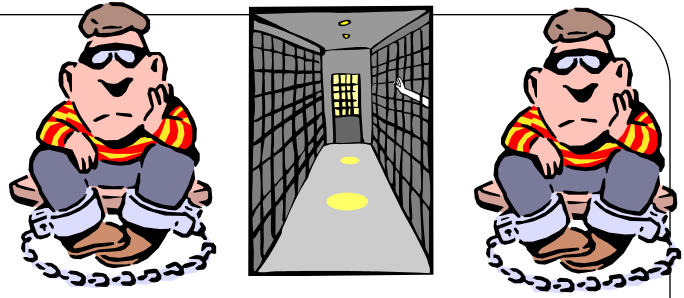
Jocurile ca probleme de cautare

- Problemele din teoria jocurilor sunt insa foarte greu de rezolvat…
- Pentru jocul de sah, factorul de ramificare este aproximativ de 35 si intr-un joc fiecare jucator are cam 50 de mutari, ceea ce face ca arborele de cautare sa aiba aproximativ 35^{100} noduri.
- X si 0 este plictisitor pentru adulti pentru ca este usor de descoperit care este mutarea corecta.
- Complexitatea jocurilor este cea care introduce un nou tip de incertitudine pe care nu l-am intalnit pana acum.
  - Incertitudinea apare nu fiindca avem informatii lipsa, ci fiindca nu este suficient timp pentru a calcula consecintele pentru toate mutarile.

Jocurile ca probleme de cautare

- **Concluzii**:
  - Jocurile sunt mult mai asemanatoare cu lumea reala decat problemele de cautare pe care le-am analizat pana acum.
  - Faptul ca mutarile adversarului sunt imprevizibile ne face sa specificam o mutare pentru fiecare posibil raspuns al adversarului.
  - Datorita limitei de timp care se impune pentru unele jocuri, nu se poate gasi tinta, trebuie sa se realizeze o aproximare a acesteia.

Dilema Prizonierului



- Doi prizonieri sunt chestionati de politie.
  - Politia stie ceva de despre ce au facut, dar nu are toate informatiile.
  - Ca sa afle, ii baga in doua celule separate si ii interogheaza.
- Prizonierii au doua optiuni:
  - Pot spune toata povestea (adica sa tradeze)
  - Pot sa nu spuna nimic (cooperare)
- Niciun prizonier nu stie ce va spune celalalt.
- Daca amandoi **coopereaza** (adica tac), ambii iau sentinta usoara (1 an).
- Daca unul **tradeaza** si celalalt **coopereaza**, tradatorul e liber, iar cel tradat primeste 10 ani de inchisoare.
- Daca ambii **tradeaza**, fiecare ia 5 ani de detentie.

Ce vor face cei doi?

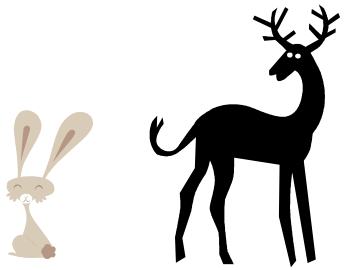
Dilema Prizonierului

		Adversarul: Cooperare	Adversarul: Tradare
EU	Cooperare	(1, 1)	(10, 0)
EU	Tradare	(0, 10)	(5, 5)

Ani de inchisoare…

Vanatoarea de cerbi/iepuri

- Doi indivizi merg la vanatoare.
- Fiecare poate alege individual sa vaneze un cerb sau un iepure si trebuie sa faca alegerea fara sa stie ce a ales celalalt.
- Daca unul alege un cerb, are nevoie de cooperarea celuilalt ca sa reuseasca.
- Fiecare poate vana un iepure de unul singur, dar un iepure valoreaza mai putin decat un cerb.



Ce vor face cei doi?

Vanatoarea de cerbi/iepuri

EU \ Adversarul	Cerb	Iepure
Cerb	(4, 4)	(1, 3)
Iepure	(3, 1)	(3, 3)

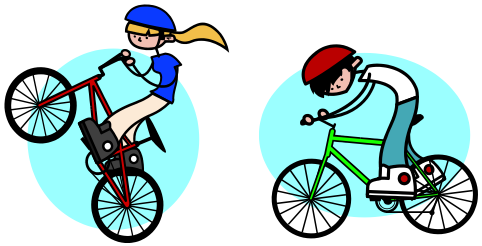
Aplicabilitate

- Ambele jocuri au o arie de aplicabilitate foarte larga:
- Economie 1
  - Daca doua companii A si B (de bere ☺) aleg sa faca reclama in o anumita prioada, ambele se anuleaza reciproc, dar costurile raman, deci cheltuielile sunt mari pentru ambele.
  - Totusi, daca B nu ar mai face reclama, A ar profita din plin prin continuarea reclamei.
  - Cantitatea de reclama a uneia depinde direct de cantitatea de reclama pe care o face cealalta.

Aplicabilitate

- Economie 2
  - Membrii unui cartel sunt implicati intr-un astfel de joc cu mai multi jucatori.
    - Cooperare in acest caz inseamna sa tina preturile la un minim prestabilit.
    - Tradarea vine de la vanzarea sub minimul prestabilit, furand astfel afacerea si profiturile celorlalti membri ai cartelului.
    - In mod ironic, autoritatile spera ca mebrii trusturilor sa se tradeze reciproc, asigurand astfel preturi reduse pentru consumatori.

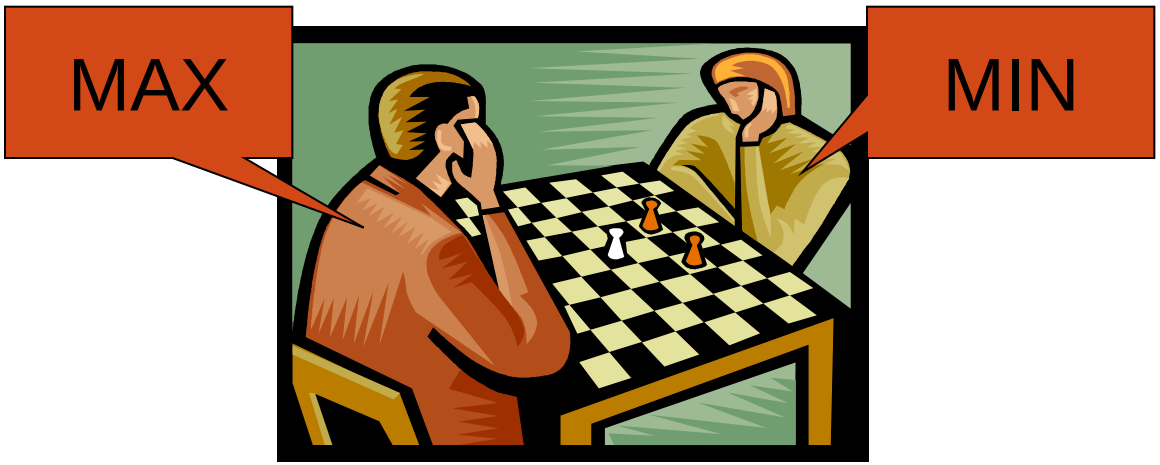
Aplicabilitate



- Sport
  - Doi ciclisti aflati in fata plutonului (grupului masiv) poarta consecutiv trena (coopereaza) pentru a nu fi ajunsi din urma.
  - De multe ori, doar unul duce trena (coopereaza), iar la linia de sosire este tradat de adversar.
- Sociologie
  - Cand cunoastem o noua persoana, tindem sa fim foarte atenti pentru a avea o pozitie de siguranta (competitie).
  - Ambii pot semnala dorinta de a se muta de la pozitiile defensive catre interactiune si recunoasterea unei intentii comune.

Decizii perfecte in jocuri de doua persoane

- Consideram cazul general al unui joc de doua persoane pe care le vom numi MAX si MIN.
- MAX este cel care muta primul, apoi muta pe rand pana la sfarsitul jocului.
- La sfarsitul jocului, puncte se atribuie jucatorului care castiga, iar pierzatorul este penalizat.

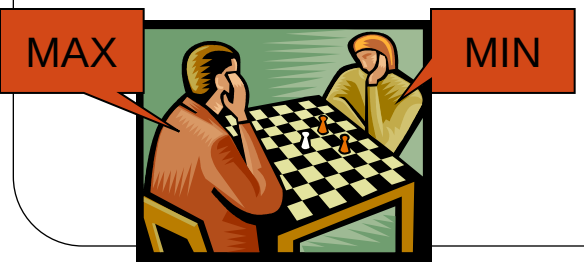


Descrierea formala a unui joc

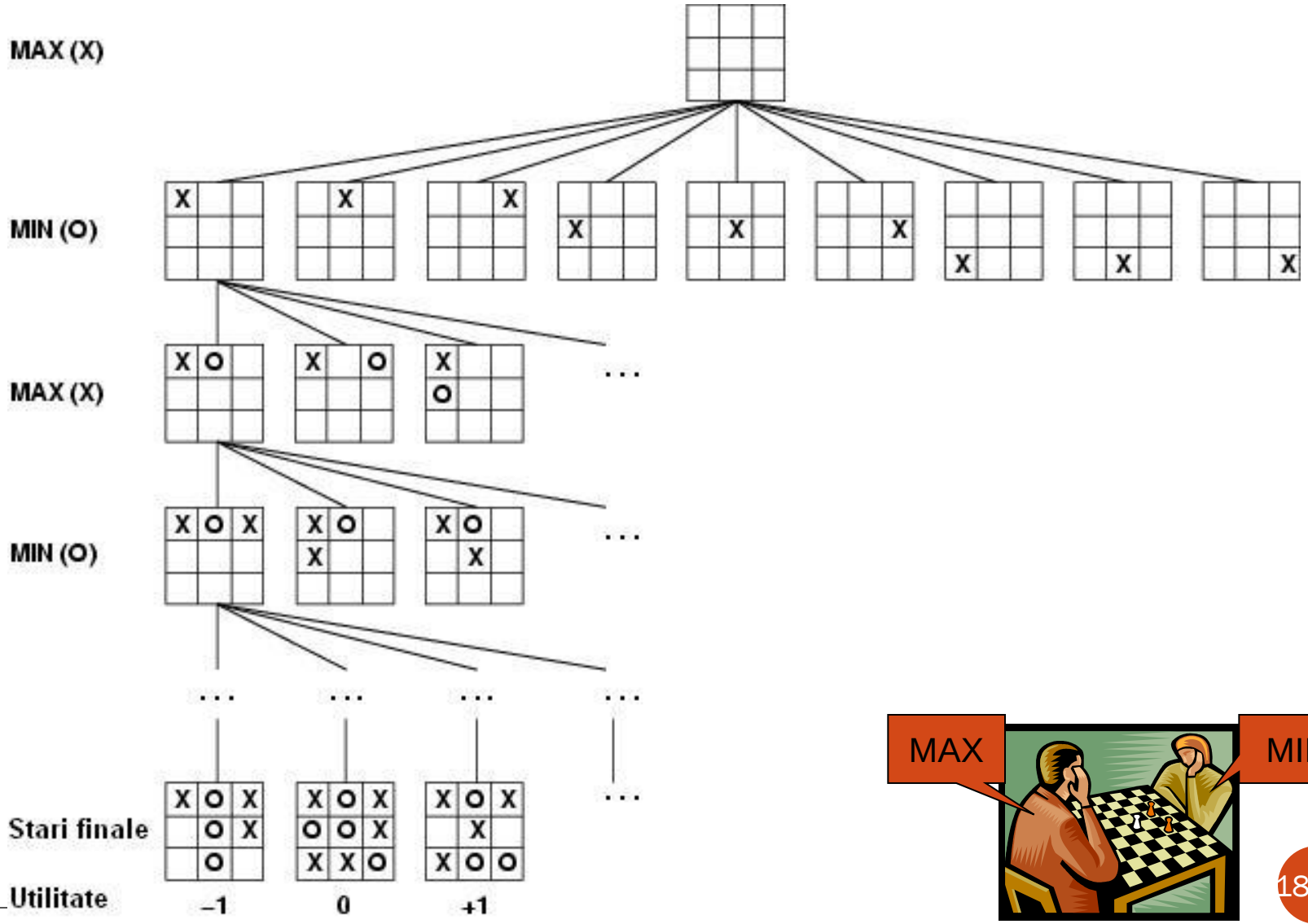
- Un joc poate fi definit formal ca o problema de cautare cu urmatoarele componente:
  - **Starea initiala** include pozitiile de pe tabla si cine este cel care este la mutare.
  - O **multime de actiuni** care definesc mutarile admise pe care le poate face un jucator.
  - O **stare terminala** care determina cand se sfarseste jocul. Starile in care jocul se incheie se numesc **stari terminale**.
  - O **functie de utilitate** care intoarce o valoare numerica pentru rezultatul jocului.
    - In general, posibilitatile sunt victorie, egal sau infrangere care pot fi reprezentate ca 1, 0 sau -1.

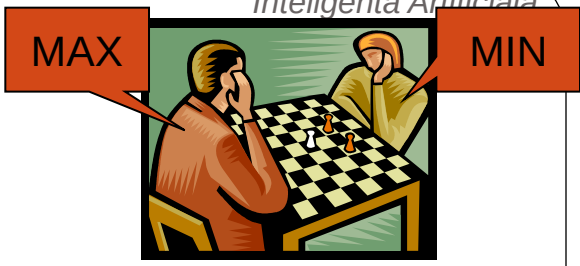
Decizii perfecte in jocuri de doua persoane

- MAX trebuie sa gaseasca o **strategie** care sa il duca la o stare terminala in care el este castigatorul, indiferent de ce mutari face MIN.
- Strategia presupune ca MAX face mutarile corecte, indiferent de mutarile lui MIN.
- Ideea este de a arata cum se gaseste o strategie optima, chiar daca in mod normal nu este timp suficient sa o gasim.



X si 0, reprezentarea jocului sub forma de arbore





X si 0, reprezentare

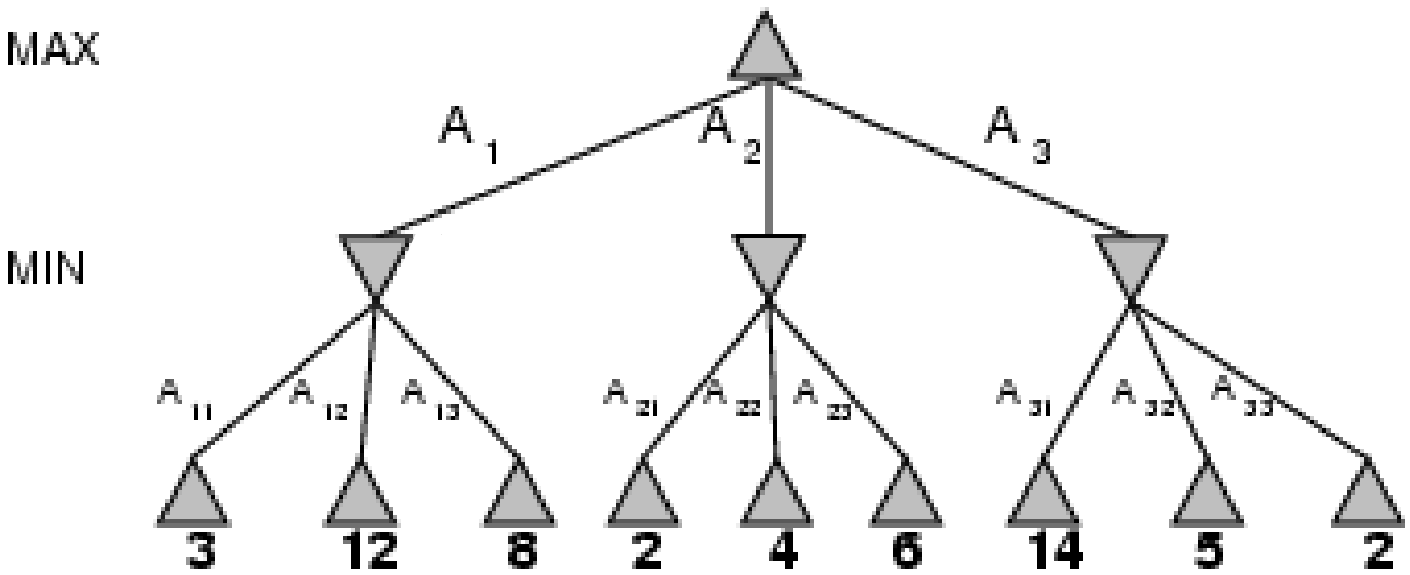
- De la starea initiala, MAX are posibilitatea de a alege din 9 stari posibile.
- Jucatorii alterneaza punand X si 0 pana cand se ajunge la o stare terminala – stare in care un jucator are trei elemente pe o linie, coloana sau diagonala ori toate casutele sunt completate.
- Numarul atasat la fiecare nod frunza se refera la utilitatea starii terminale pentru jucatorul MAX.
  - Valorile mari sunt considerate bune pentru MAX si proaste pentru MIN (si invers), de aici si numele celor doi jucatori.
- Sarcina lui MAX este sa foloseasca arborele de cautare pentru a determina cele mai bune mutari, tinand cont de utilitatile starilor terminale.

Algoritmul minimax

- Algoritmul **minimax** determina strategia optima pentru MAX.
- Consta din 5 pasi:
  1. Genereaza tot arborele pentru joc pana la starile terminale.
  2. Aplica functia de utilitate pentru fiecare stare terminala pentru a ii determina valoarea.
  3. Foloseste utilitatea starilor terminale pentru a determina utilitatea starilor de la un nivel superior din arborele de cautare.
     - Cum se face acest lucru?... (vom reveni la cei 5 pasi ai algoritmului!)

Stabilirea utilitatii - exemplu

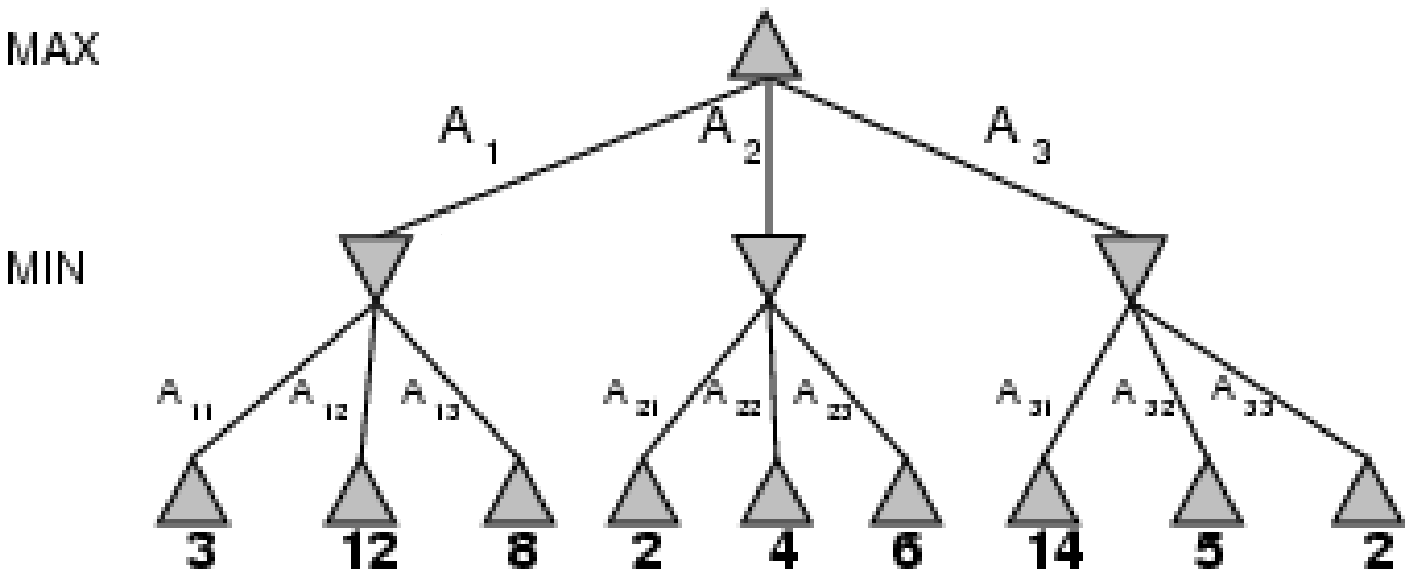
- Consideram un joc care se termina doar dupa doua mutari (una MAX si una MIN).
- Miscarile posibile ale lui MAX: A1, A2 si A3, iar ale lui MIN A11, A12 etc.
- Valorile pentru starile terminale – intre 2 si 14.



- Nodurile in care MAX ia deciziile sunt notate cu ▲
- Cele in care MIN ia deciziile: ▼

Stabilirea utilitatii - exemplu

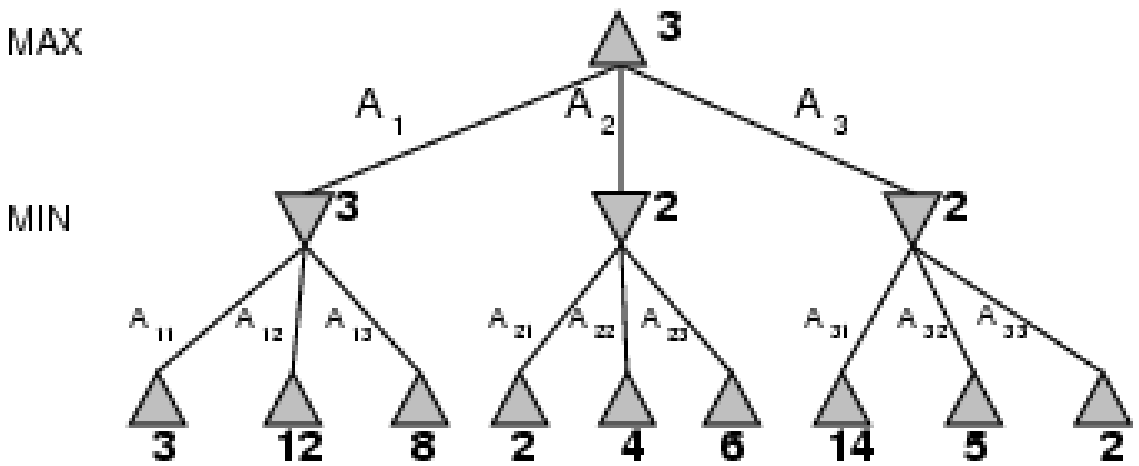
- In varful din stanga jos, utilitatea este 3.
- Alegerea pe care o face MIN in nodul de deasupra va fi cea mai mica, in cazul in care ia decizia cea mai buna.
- Analog, in celelalte noduri evaluarile vor fi 2 si 2.
- In nodul radacina, MAX va lua, evident, valoarea maxima, 3.



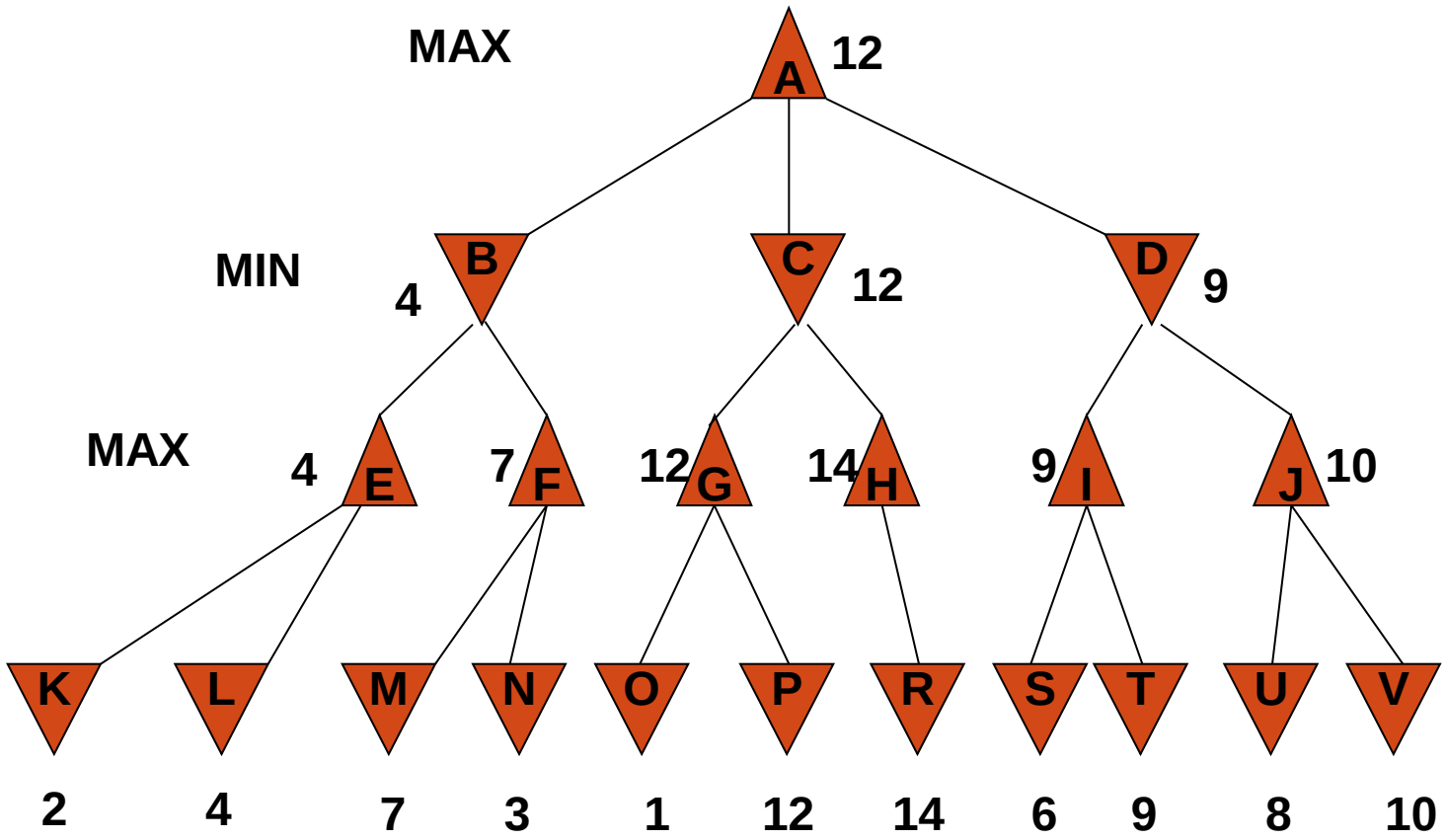
Algoritmul minimax (continuare)

4. Continua evaluarea utilitatilor nodurilor pe niveluri mergand pana la radacina.
5. Cand se ajunge la radacina, MAX alege nodul de pe nivelul inferior cu valoarea cea mai mare.

- MAX alege initial mutarea A1. Aceasta este **decizia minimax** pentru ca maximizeaza utilitatea si presupune ca adversarul va juca perfect pentru a o minimiza.



Aplicati algoritmul minimax pentru arborele de mai jos!



Algoritm de calcul al deciziilor mimimax

```
functia decizie_minimax(stare) intoarce actiune
        v = valoare_maxima(stare)
intoarce actiunea din SUCCESORI[stare] cu valoarea v
```

```
functia valoare_maxima(stare) intoarce valoarea unei utilitati
Daca TEST_TERMINAL[joc](stare) atunci intoarce utilitate(stare)
v = - ∞
Pentru orice s din SUCCESORI[stare] executa
        v = maximum(v, valoare_minima(s))
intoarce v
```

```
functia valoare_minima(stare) intoarce valoarea unei utilitati
Daca TEST_TERMINAL[joc](stare) atunci intoarce utilitate(stare)
v = ∞
Pentru orice s din SUCCESORI[stare] executa
        v = minimum(v, valoare_maxima(s))
intoarce v
```

Proprietatile algoritmului minimax

- Completitudine? Da (daca arborele este finit)
- Optimal? Da (impotriva unui adversar optimal)
- Complexitatea temporala? $O(b^m)$
- Complexitatea spatiala? O(bm) (explorare in adancime)
  - b este factorul de ramificare
  - m este adancimea arborelui

- Pentru sah, unde b ≈ 35 si m ≈100 pentru un joc, nu se poate aplica algoritmul minimax din cauza timpului.

Decizii imperfecte

- Algoritmul minimax presupune ca programul are timp sa caute pana la starile terminale, ceea ce este de obicei impractic.
- Shannon propunea ca in loc sa se mearga pana la starile terminale si sa se foloseasca functia de utilitate, cautarea ar trebui oprita mai devreme si sa se aplice o **functie de evaluare euristica** la noile frunze ale arborelui.
- Modificarea algoritmului minimax se face in 2 moduri:
  - Functia de utilitate este inlocuita de functia de evaluare.
  - Testul terminal este inlocuit de o **reducere a arborelui** si de evaluarea noilor frunze.

Functia de evaluare

- Functia de evaluare intoarce o **estimare** a utilitatii asteptate pentru joc intr-o stare data.
- O posibilitate pentru jocul de sah se poate referi la valoarea materiala pentru fiecare piesa:
  - Pion: 1
  - Nebun: 3
  - Tura: 5
  - Regina: 9
- E bine sa tinem cont si de alte considerente precum asezarea pionilor sau modul in care este protejat regele etc.

Functia de evaluare

- Performanta unui program pentru jocuri este strans legata de functia de evaluare aleasa.
  - Daca functia nu este una bine definita, aceasta va ghida programul spre stari care sunt aparent bune, dar de fapt sunt dezastruoase.
- Functia de evaluare trebuie sa corespunda cu functia de utilitate care se aplica la nodurile terminale.
- Functia de evaluare nu trebuie sa necesite foarte mult timp!
- Concluzia: trebuie facut un compromis intre acuratetea functiei de evaluare si costul de timp.
- In plus, **functia de evaluare trebuie sa reflecte** in mod corect **sansele reale de a castiga**!

Functia de evaluare

- In cazul in care evaluarea se bazeaza doar pe valoarea materiala, toate starile in care nicio piesa nu este capturata sunt egale intre ele.
- Functia de evaluare bazata pe valoarea materiala pentru sah este **liniara** si **ponderata** pentru ca poate fi data ca:

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

- Unde w_i este pondere, iar $f_i(s)$ este numarul de piese i care sunt in plus fata de adversar.
- Exemplu:
  - $w_2 = 5$ (pentru ture)
  - $f_2(s)$ = (numarul de ture ale lui MAX) – (numarul de ture ale lui MIN)

Functia de evaluare

- Pentru majoritatea programelor pentru jocuri se foloseste o functie liniara de evaluare.

- In constructia unei formule liniare, trebuie intai alese caracteristicile de care se tine cont (ex: diferenta dintre numarul de piese de acelasi tip) si apoi se ajusteaza ponderile pana cand programul da un bun randament.

Implementare

- In orice moment, trebuie sa avem toate mutarile legale disponibile.
- int genereazaListaMutari(Pozitie *p*, Mutare *lista*[MUTARIMAX])
  - Genereaza toate mutarile posibile si le stocheaza in *lista*.
- void faMutare(Mutare *m*, Pozitie *p*)
  - Face mutarea *m* in pozitia *p* (ex: aduga x in centru, la x si 0)
- void faMutareInapoi(Mutare *m*, Pozitie *p*)
  - Face inapoi mutarea (ex: scoate x din centru)
- int evaluare(Pozitie *p*)
  - Intoarce o valoare pozitiva daca pozitia este buna si negativa altfel.

```
int minimax(Pozitie p, int adancime){
Mutare lista[MUTARIMAX]; int i,n,bestval,val;
if(castig(p)) {
    if (max a pierdut) return -INFINIT;
    else return INFINIT;}
if(adancime == 0) return evaluare(p);
if(max este la mutare) bestval = -INFINIT;
else bestval = INFINIT;
n = genereazaListaMutari(p,lista);
if(n == 0) return 0; // remiza la x si 0 sau la sah
for(i=0; i<n; i++){
    faMutare(lista[i], p);
    val = minimax(p,adancime-1);
    //pentru actiunea lista[i] atasam valoarea calculata val;
     faMutareInapoi(lista[i],p);
    if(max este la mutare) bestval = max(val,bestval);
    else bestval = min(val,bestval);}//de la for
return bestval;
}
```

Reducerea cautarii

- Cea mai simpla abordare in a controla cat sa mearga de adanc cautarea este de a folosi o limita de cautare in adancime.
  - Reducerea se face in acest caz la toate nodurile care se afla la si pana la adancimea *d*.
- Adancimea se alege in asa fel incat sa nu se depaseasca timpul alocat de catre joc pentru o mutare.
- O posibilitate mai buna ar fi de a considera cautare iterativa in adancime.
  - Cand nu mai este timp, programul intoarce mutarea selectata de cautarea completa de la adancimea la care s-a ajuns.

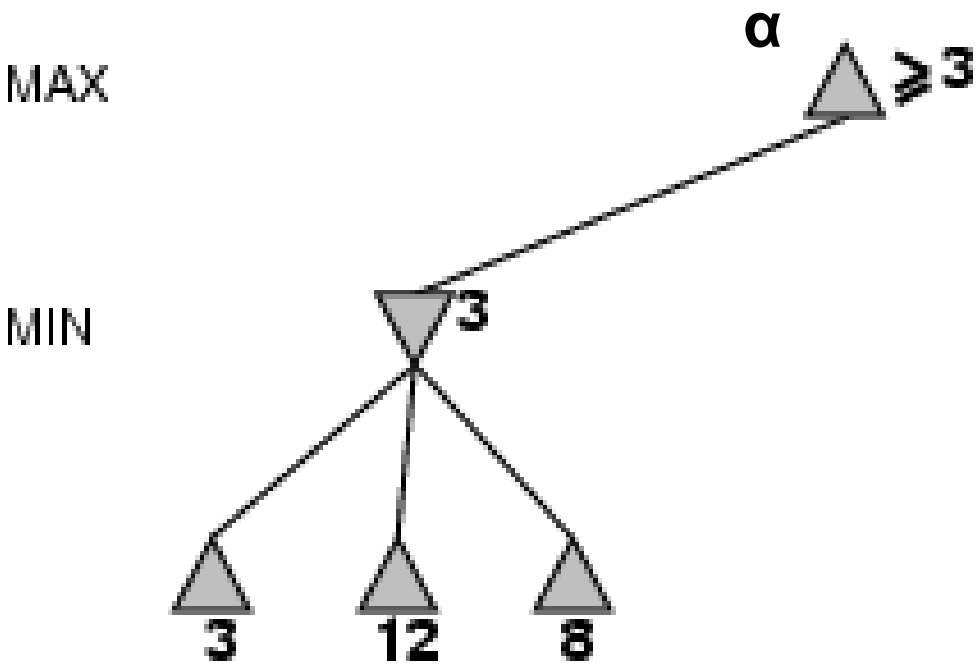
E indeajuns de bun un astfel de algoritm?

- Un program bine scris poate cauta aproximativ 100 de pozitii pe secunda.
- In turnee de sah timpul pentru mutare este de 150 de secunde, deci programul ar putea cauta 150 000 de pozitii.
- Cum factorul de ramificare este cam 35, programul va fi capabil sa meaga in avans numai cu 3-4 mutari, ceea ce l-ar face sa joace la un nivel de incepator!
  - Chiar si un jucator mediu poate vedea 6-7 mutari inainte, ceea ce il face pe program sa fie usor batut.

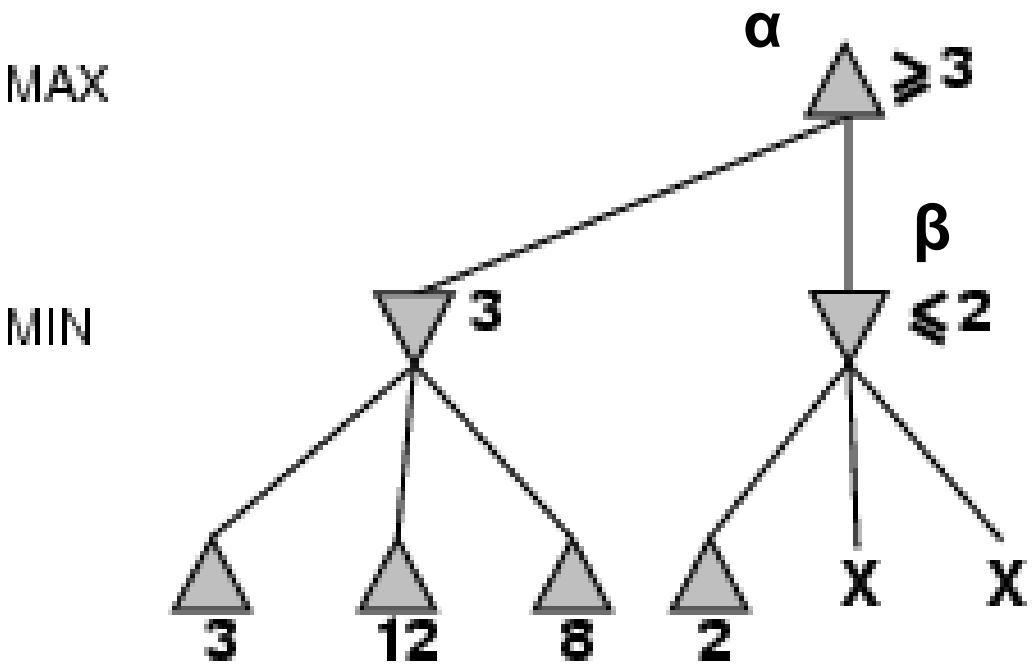
Solutia?

- Din fericire, este posibil sa se calculeze decizia minimax fara a vizita fiecare nod din arborele de cautare.
- Procesul consta intr-o **retezare** a unor ramuri ale arborelui si presupune neluarea in considerare a acelor ramuri.
- Tehnica de reducere a arborelui de care vom discuta se numeste **reducere α-β**.
- Atunci cand se aplica unui arbore, ea va intoarce aceeasi mutare ca si minimax, insa ea elimina ramuri care nu pot influenta decizia finala.

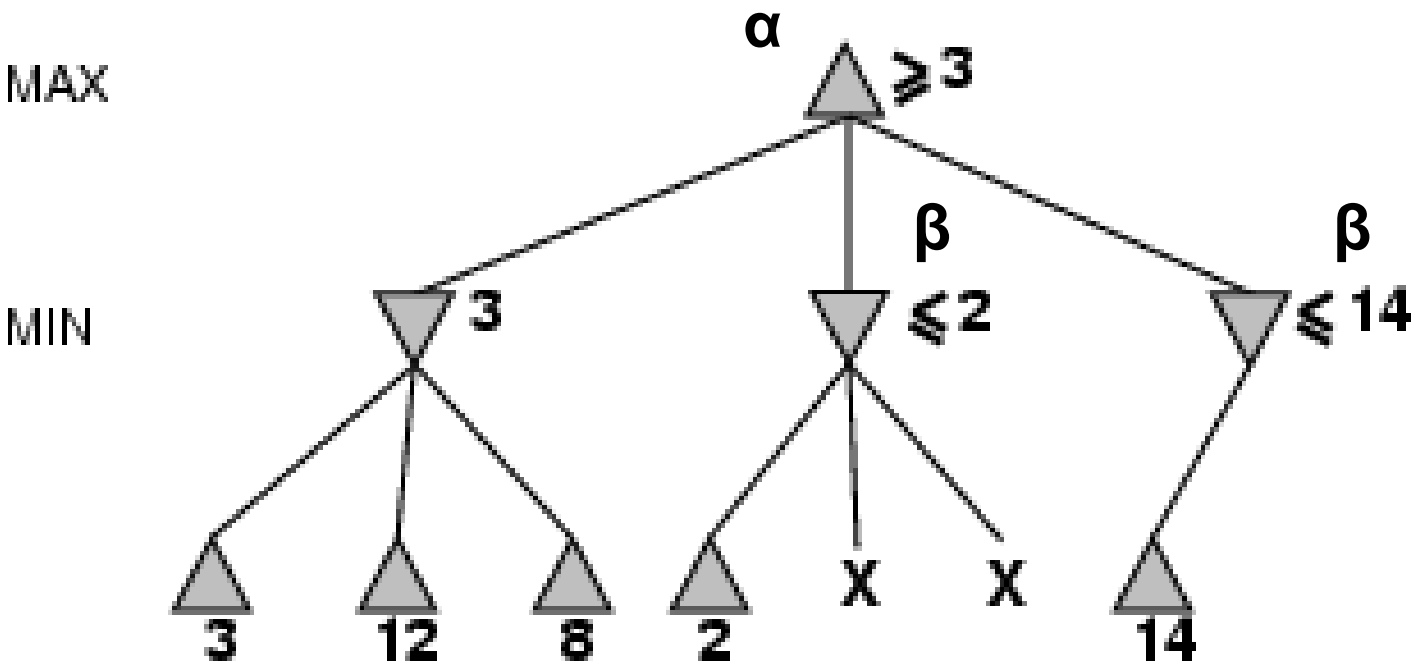
Exemplu reducere α-β



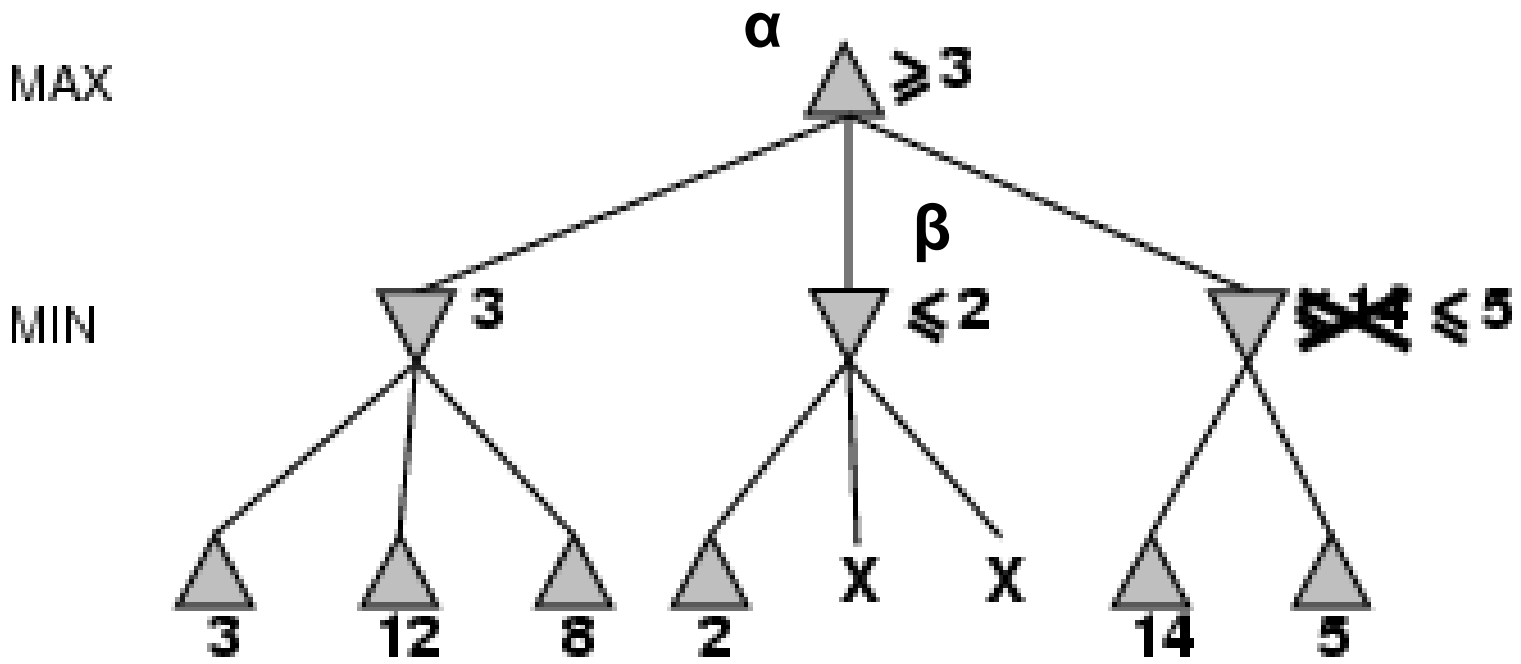
Exemplu reducere α-β



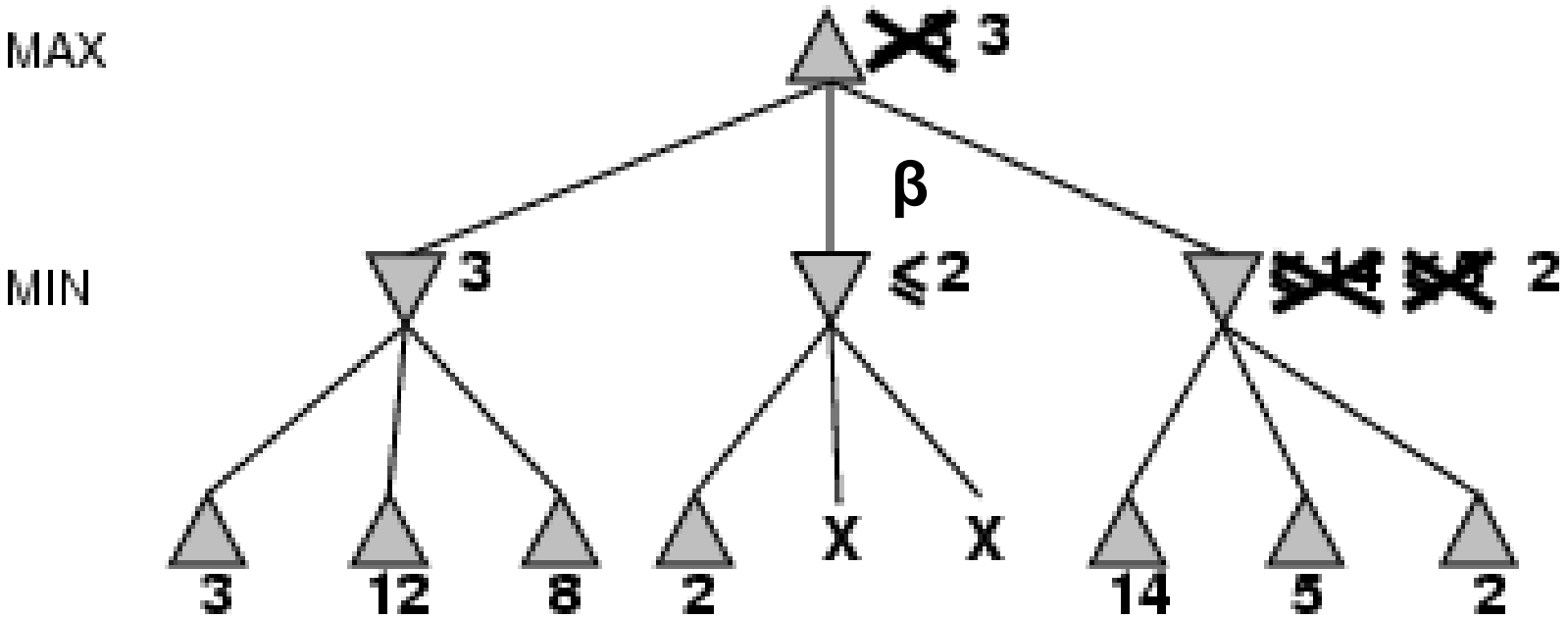
Exemplu reducere α-β



Exemplu reducere α-β

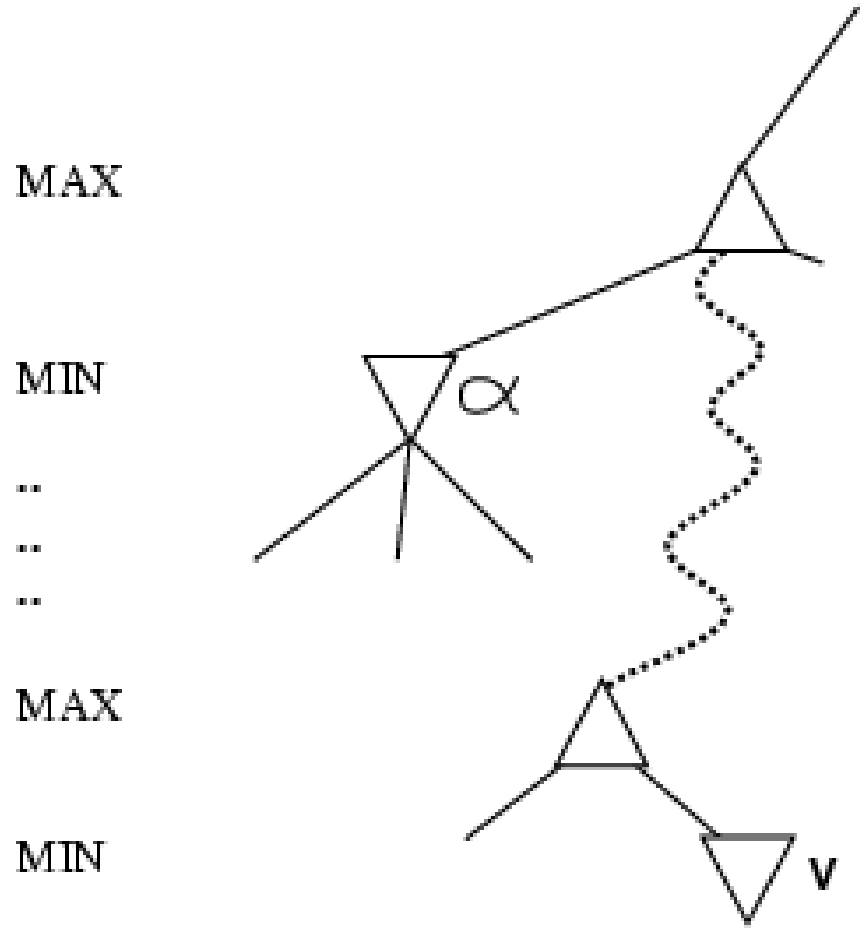


Exemplu reducere α-β

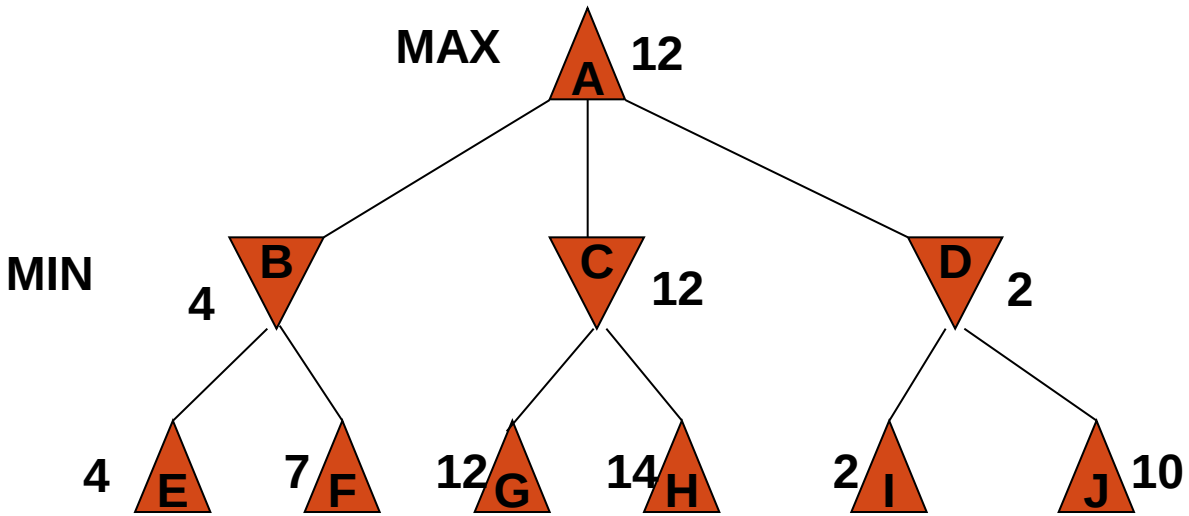


De ce se numeste α-β?

- α este valoarea celei mai bune (adica cea mai mare) alegeri gasita pana la momentul curent la orice punct de-a lungul unui drum pentru MAX.
- Daca *v* este mai prost (mai mic) decat α, MAX il va evita prin eliminarea acelei ramuri.
- β este definit in mod similar pentru MIN, adica cea mai mica valoare gasita la orice punct de-a lungul unui drum pentru MIN.



Aplicati reducerea α-β pentru arborele de mai jos!



Cand se poate reteza arborele?

- Valorile α ale lui MAX nu pot descreste
- Valorile β ale lui MIN nu pot creste

1. Retezam sub nodul MIN a carui valoare β este mai mica sau egala cu limita α care apartine nodului predecesor unde muta MAX
2. Retezam sub nodul MAX a carui valoare α este mai mare sau egala cu limita β care apartine nodului predecesor unde muta MIN

Algoritmul de reducere α-β

```
functia cautare_alfa_beta(stare) intoarce actiune
        v = valoare_maxima(stare, - ∞, ∞)
intoarce actiunea din SUCCESORI[stare] cu valoarea v
```

```
functia valoare_maxima(stare, α, β) intoarce valoarea unei utilitati
Daca TEST_TERMINAL[joc](stare) atunci intoarce utilitate(stare)
v = - ∞
Pentru orice s din SUCCESORI[stare] executa
        v = maximum(v, valoare_minima(s, α, β))
        Daca v ≥ β atunci intoarce v
        α = maximum(α, v)
Sfarsit pentru
intoarce v
```

Valoarea celei mai bune alternative pentru MAX in drumul catre stare.

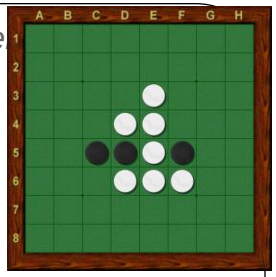
Valoarea celei mai bune alternative pentru MIN in drumul catre stare.

Algoritmul de reducere α-β (continuare)

functia valoare_minima(stare, α, β) ***intoarce*** valoarea unei utilitati
Daca TEST_TERMINAL[joc](stare) *atunci* ***intoarce*** utilitate(stare)
v = + ∞
Pentru orice s din SUCCESORI[*stare*] *executa*
    v = minimum(v, valoare_maxima(s, α, β))
    Daca v ≤ α *atunci* ***intoarce*** v
    β = minimum(β, v)
Sfarsit pentru
intoarce v

Proprietati ale algoritmului de reducere α-β

- Reducerea nu afecteaza rezultatul final!
- O buna ordonare a mutarilor imbunatateste algoritmul de reducere.
- Daca succesorii sunt pusi perfect in ordine (cei mai buni se afla primii), atunci complexitatea temporara ar fi = $O(b^{d/2})$, in loc de $O(b^d)$ cat are minimax.
  - Deci α-β poate cauta de doua ori mai mult decat minimax cu acelasi pret.
  - Intorcandu-ne la exemplul cu sahul, programul ar putea sa se uite inainte cu 8 mutari in loc de 4.
- Daca suntem atenti care sunt calculele care afecteaza decizia, putem transforma un program de la nivelul incepator la expert.



Jocurile deterministe in practica

- **Sah:** in mai, 1997, Garry Kasparov a fost invins de catre Deep Blue cu 3.5-2.5.
  - Deep Blue cauta 200 de milioane de pozitii pe secunda, foloseste evaluari foarte sofisticate si metode de a extinde unele drumuri pana la 40 de mutari.
  - http://www.research.ibm.com/deepblue/
- **Dame:** Chinook a pus punct dominatiei de 40 de ani a campionului mondial Marion Tinsley in 1994.
- **Othello:** campionii refuza sa joace impotriva calculatorului pentru ca ar fi usor invinsi.
- **Go:** campionii umani nu joaca impotriva calculatorului pentru ca acesta din urma este prea slab.
  - La go, b > 300!

Recapitulare

- Un joc poate fi definit prin:
  - **Starea initiala** (cum sunt elementele aranjate initial)
  - **Actiunile** posibile (unde sunt definite mutarile permise)
  - Un **test terminal** (care spune daca jocul s-a terminat)
  - O **functie de utilitate** (care spune cine a castigat si cu cat, cu ce scor)
- Algoritmul minimax poate determina cea mai buna mutare pentru un jucator, presupunand ca adversarul joaca perfect, prin enumerarea intregului arbore al jocului.
- Algoritmul alfa-beta face aceleasi calcule ca si minimax, dar este mai eficient pentru ca elimina ramurile arborelui de cautare care nu au relevanta pentru rezultatul final.
- De obicei nu este convenabil sa se considere intregul arbore al jocului, chiar daca se foloseste si alfa-beta, motiv pentru care se opreste cautarea la un anumit punct si se aplica o functie de performanta care estimeaza utilitatea unei stari.