Inteligență Artificială

Universitatea Politehnica Bucuresti
Anul universitar 2021-2022

Adina Magda Florea

Curs 4

Strategii de căutare

- Căutare adversarială în jocuri

1. Strategii de căutare în jocuri

- Teoria jocurilor – teoria deciziei pt agenți care interacționează
- Căutări specifice datorita acestei particularități
- S: set de stări cu S_0
- N – număr de jucători
- A – mulțime de acțiuni
- f: S x A $\rightarrow$ S (f sau next)
- Q: S $\rightarrow$ R^N – funcția de utilitate / recompensa
- J: S $\rightarrow$ (1, 2…., N) – jucătorul care joaca

1.1 Jocuri cu 2 adversari

- Jocuri ce implică doi adversari (N=2)
  - Jucător
  - Adversar
- Algoritmul Minimax
- Algoritmul Alfa-Beta
- Algoritmul Monte Carlo Tree Search

Minimax

- 1944 - von Neumann si Morgenstern descriu cum un proces, numit Minimax, este capabil sa identifice rezultatul final al unui joc si sa aleaga mutarea cea mai buna pentru orice stare de joc, în cazul jocurilor cu informatie perfecta.
- Bazele teoretice ale algoritmului – 1928 - von Neumann
- **Non-cooperative Game Theory** / 2 players zero-sum games
- Valoarea **maxmin** a J1 este cea mai mare valoare pe care J1 poate spera sa o obtina fara sa stie actiunea lui J2

$$\mathbf{max_{J1}\ min_{J2} u_{J2} (a_{J1}, a_{J2})}$$

- **Combinatorial game theory /** sequential games with perfect information
- **Algoritmul Minimax**

Minimax

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

Minimax pentru spații de căutare investigate până la o adancime *n*

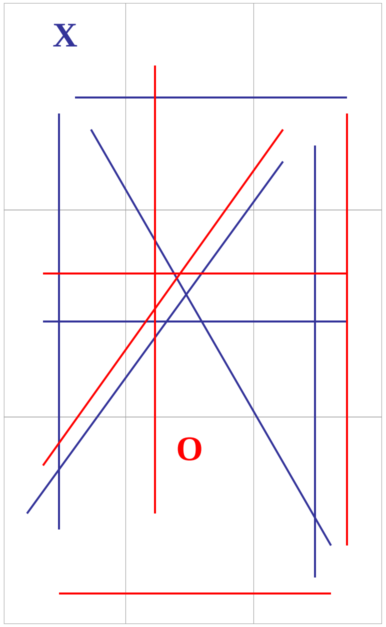
- Spatiul de cautare este f mare sau infinit, nu poate fi investigat exhaustiv
- Algoritmul Minimax pana la o adancime *n*
- *nivel(S)*
- O functie euristica de evaluare a unui nod *eval(S)*

Exemplu de funcție 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) în Tic-Tac-Toe



X are 6 linii castigatoare posibile

O are 5 linii castigatoare posibile

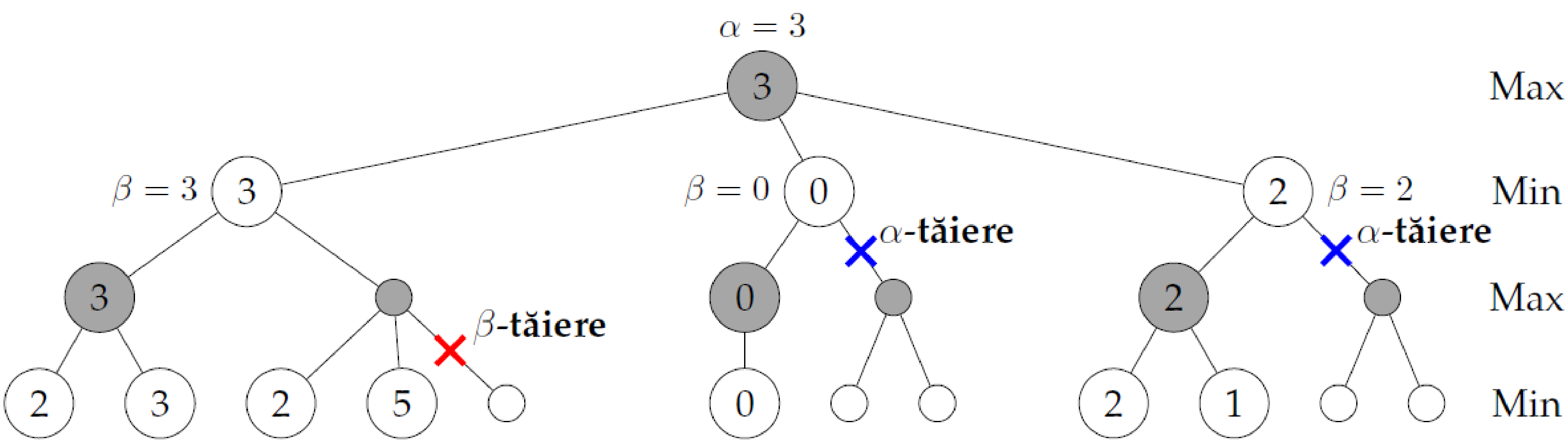
eval(**S**) = 6 **-** 5 = 1

Algoritmul tăierii 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 tăierii 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:
  - **α -taieri** - În cazul în care exista, pentru un nod **Min**, o actiune ce are asociata o valoare v $\leq \alpha$, atunci putem renunta la expandarea subarborelui sau, deoarece **Max** poate atinge deja un câstig mai mare, dintr-un subarbore precedent.
  - **β -taieri** - În cazul în care exista, pentru un nod de tip **Max**, o actiune ce are asociata o valoare v $\geq \beta$, atunci putem renunta la expandarea subarborelui sau, deoarece **Min** a limitat deja castigul lui **Max** la β


$\alpha = 3$
3
Max
$\beta = 3$
3
$\beta = 0$
0
2
$\beta = 2$
Min
α-tăiere
α-tăiere
3
0
2
Max
β-tăiere
2
3
2
5
0
2
1
Min

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 tăierii alfa-beta

- Eficienta algoritmului depinde semnificativ de ordinea de examinare a starilor
- Se recomanda o ordonare euristica a succesorilor, eventual de generat numai primii cei mai buni succesori
- Poate reduce semnificativ timpul de cautare
- *De exemplu – incepe cu cea mai "simpla" miscare sau favorizeaza nodurile cu taieri B*

Imbunatatiri

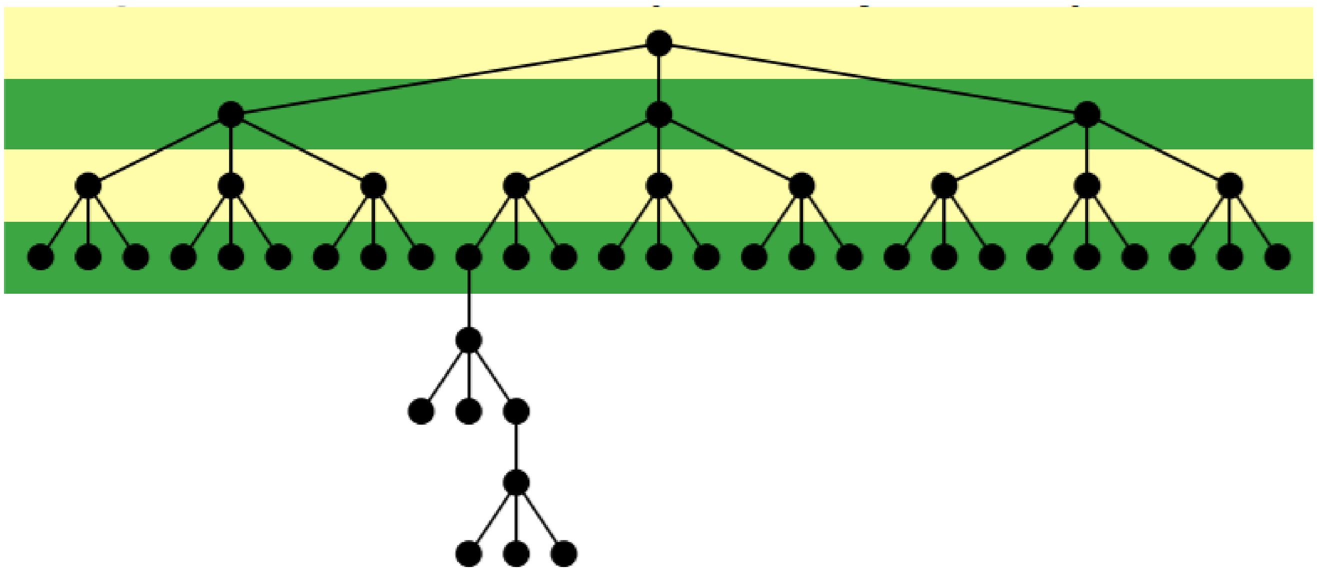
- Timp limitat pentru executarea unei miscari – anytime algorithm
- Foloseste **Iterative Deepening**
- incepe cu ply 1 si obtine cea mai buna miscare
- apoi ply 2 folosind cele mai bune stari cf evaluarii anterioare
- continua cresterea ply pana la expirarea timpului

Imbunatatiri

- Diferite secvente de mutari pot duce la aceleasi pozitii
- Mai multe pozitii de joc pot fi functional echivalente (de ex pozitiile simetrice)
- **Memoize** – tabela hash cu pozitiile de joc pentru a obtine:
  - Estimari ale nodurilor
  - Cea mai buna miscare dintr-un nod

Imbunatatiri

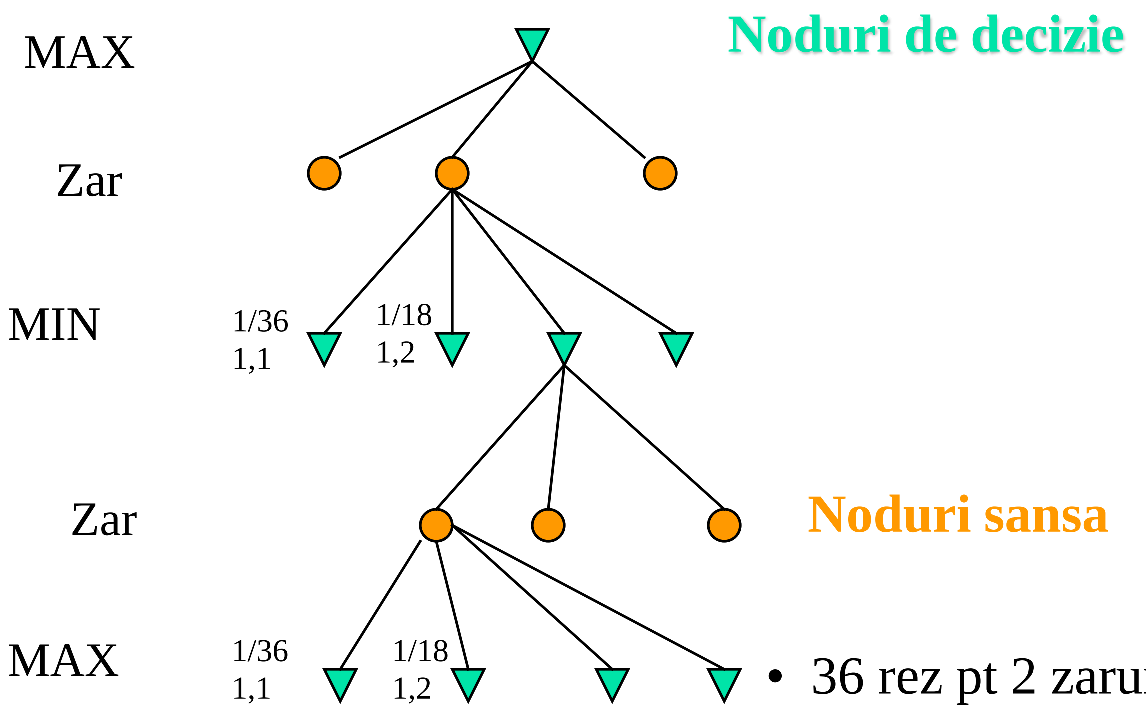
- **Efectul de orizont** – cauta mai mult decat limita de cautare pentru anumite pozitii



Jocuri cu elemente de șansă

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

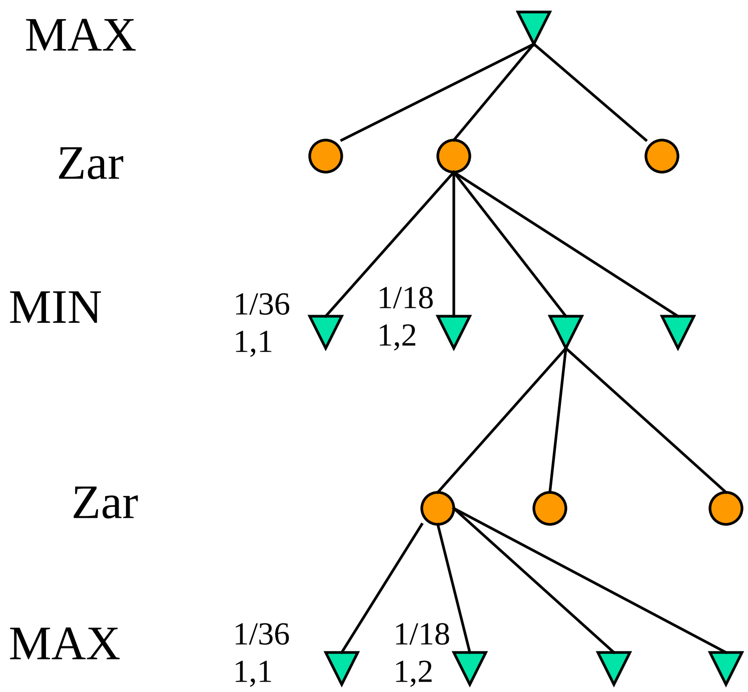
Noduri șansă – ramurile care pleaca dintr-un nod șansa indica posibile rezultate ale sansei (de exemplu zar)



- 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

1.2 Jocuri cu mai mulți jucători

Exemplu: Chinese checkers (table chinezesti)

- Chinese checkers nu este un joc de origine chineza, ci este o variatie moderna si simplificata, aparuta în Germania prin 1892, a celebrului joc american Halma, inventat de George H. Monks în 1880.
- Abia în momentul în care a devenit cunoscut în Statele Unite ale Americii, a primit denumirea Chinese checkers

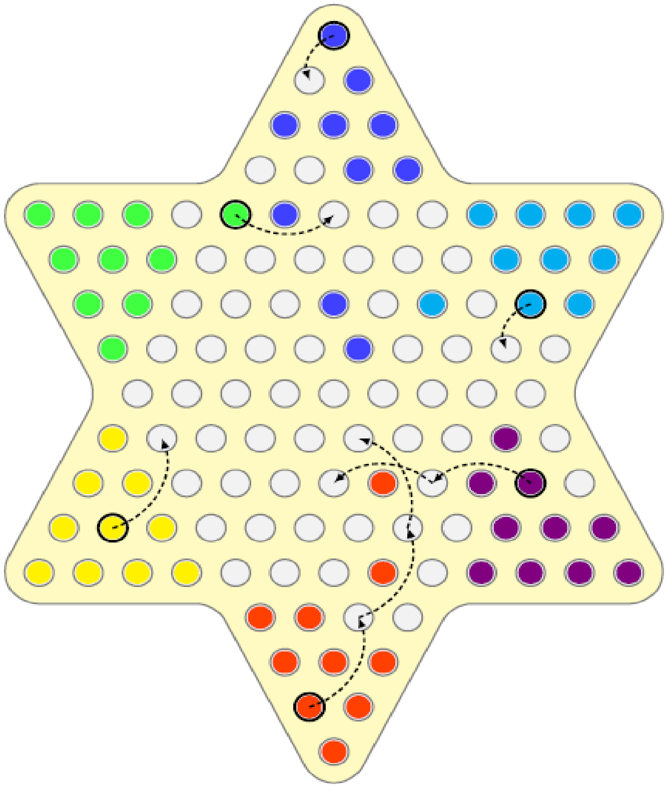
George Howard Monks

Jocuri cu mai mulți jucători

Chinese checkers

- Este un joc cu informatie perfecta pentru 2-6 jucatori
- Scopul este sa deplasam 10 piese dintr-o pozitie de start intr-o pozitie finala cat mai repede.

- **Chinese checkers**
- Piesele se muta prin deplasare intr-o pozitie alaturata sau sarind peste piese alaturate daca exista un loc liber. Se poate sari peste orice numar de piese si se pot inlantui mai multe sarituri. Piesele nu sunt eliminate dupa sarituri.



Jocuri cu mai mulți jucători

Nu exista algoritmi buni consacrati

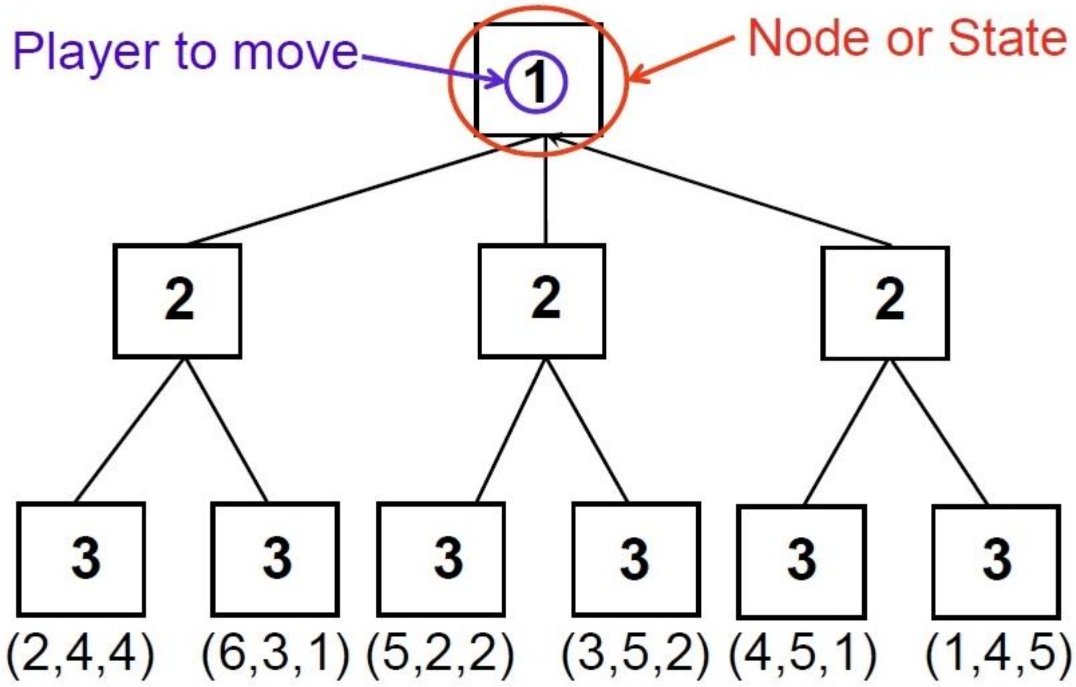
In general, 2 strategii

- **Max^n** **-** generalizare a Minimax pt **n** jucatori
- **Paranoic** – reduce la joc cu 2 jucatori in care se presupune ca toti ceilalti colaboreaza impotriva jucatorului simulat

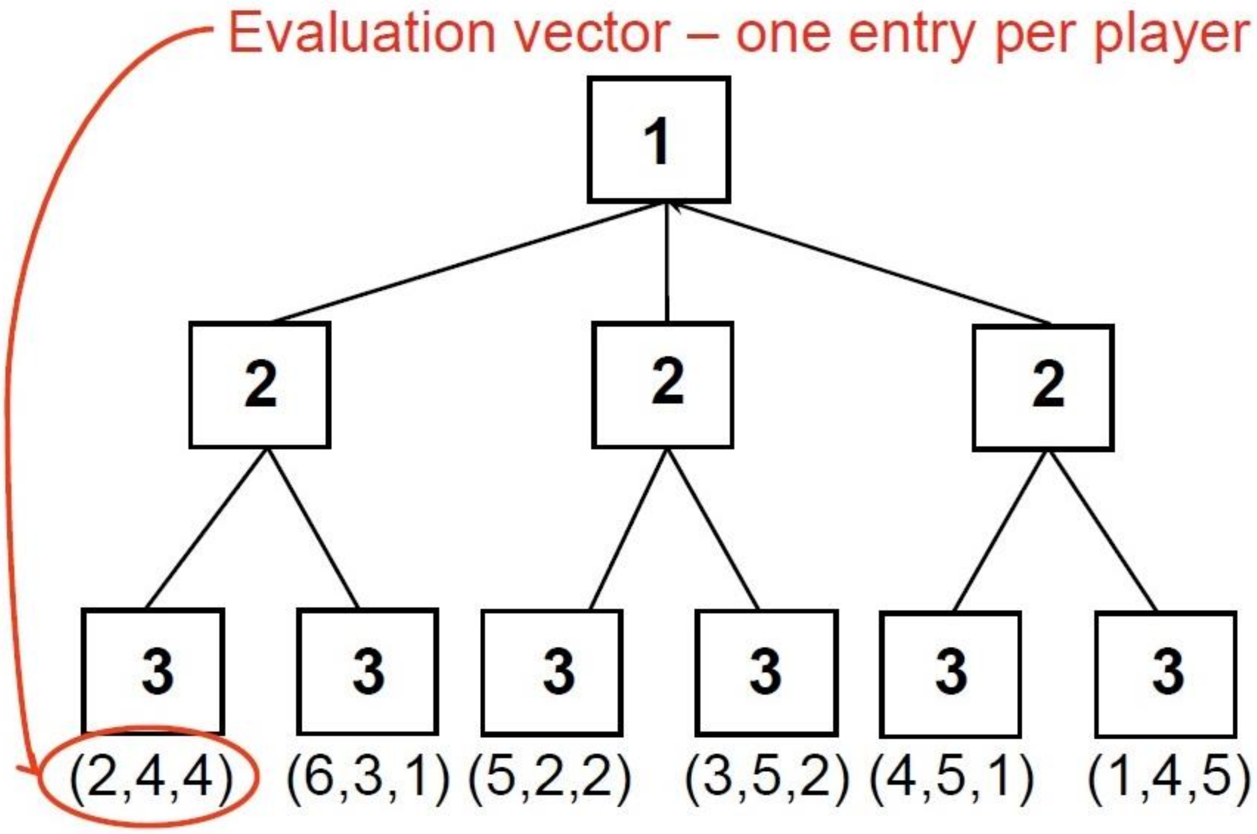
Presupunem jocuri cu informatie perfecta

Jocuri cu mai mulți jucători

- Functia de evaluare – dependenta de joc
- Regula de decizie in parcurgerea arborelui de cautare – generica

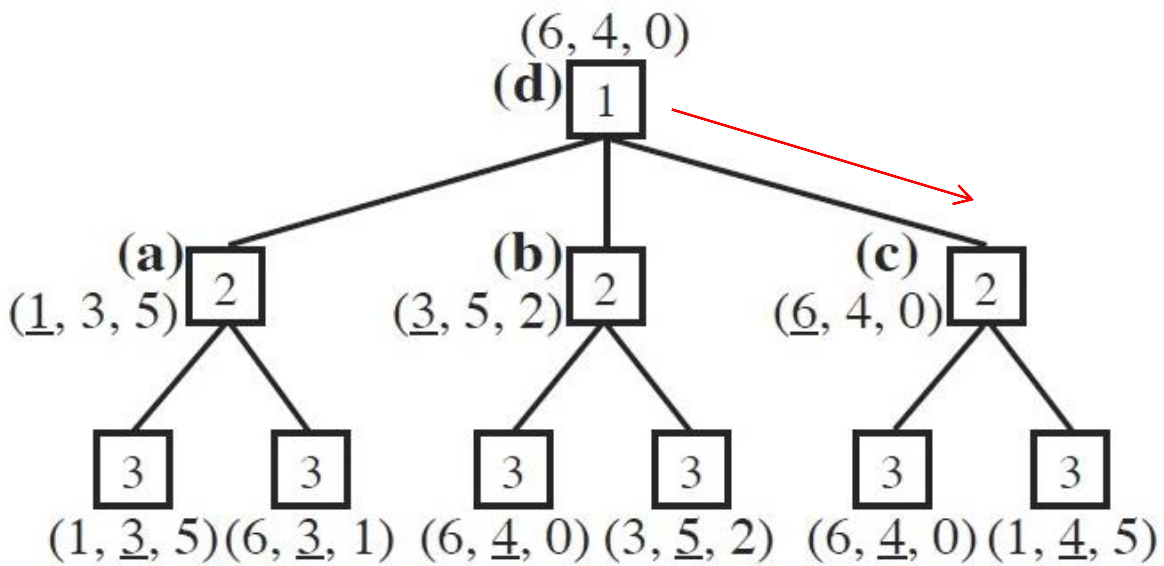


Jocuri cu mai mulți jucători



Strategie Max^n

- Generalizarea Minimax pentru n jucatori
- Frunzele arborelui de joc sunt n-tuple, in care elementul pe pozitia ***i*** este scorul jucatorului ***i***.
- Pentru nodurile din interior, valoarea Max^n a unui nod in care jucatorul ***i*** muta este valoarea Max^n a succesorului pentru care a ***i***-a componenta din vector este maxima.



Strategie Max^n

Maxn(Nod, Juc)

1. daca Nod este nod final **atunci intoarce** scor(Nod[Juc])

2. daca nivel(Nod) = n **atunci** intoarce eval(Nod[Juc])

3. altfel

3.1 P ← Prim_succesor(Nod)

3.2 Best [Juc] ← Maxn (P, Juc_Urm)

3.3 **pentru** fiecare succesor $S_j \neq P$ al lui Nod **executa**

- Curent ← Maxn(S_j, Juc_Urm))

- **if** Curent[Juc] > Best [Juc]

  then Best ← Curent

4. Intoarce Best [Juc]

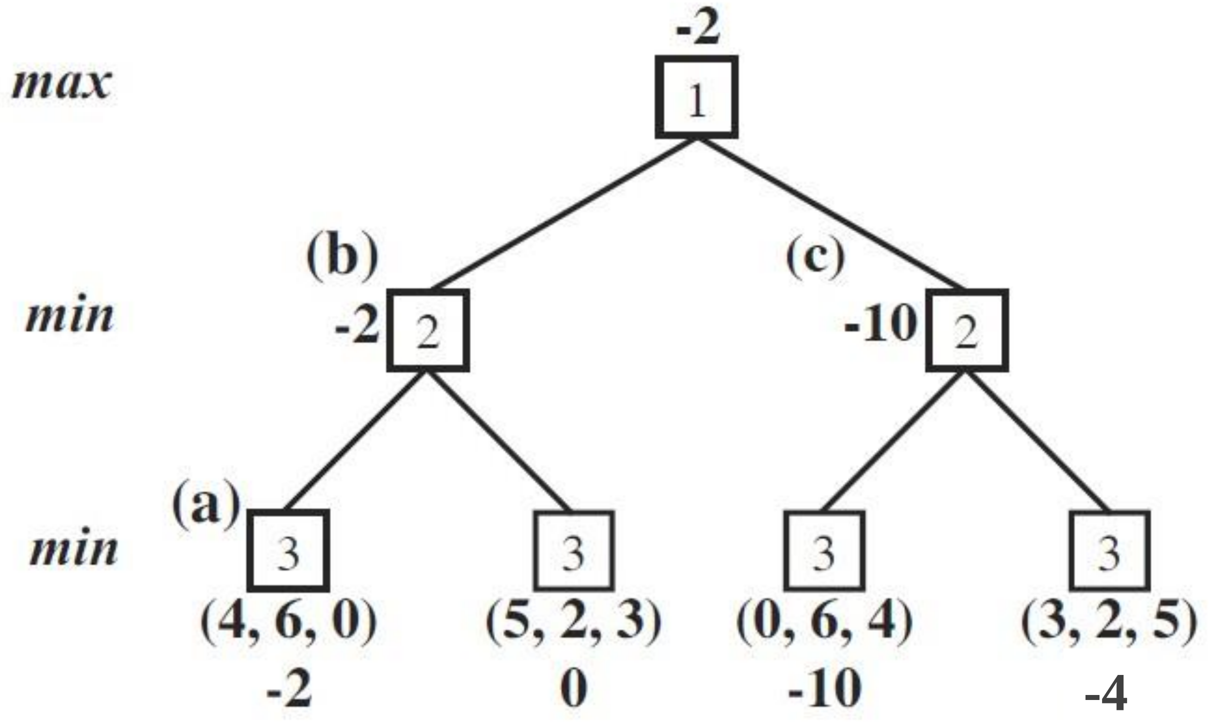
Strategie Max^n

- Pot exista multe valori egale Max^n intr-un arbore
- Rezultatul poate depinde drastic de felul in care se face alegerea
- E.g., (2,3,3) vs. (2,1,7)

- Alpha-beta in adancime nu poate fi aplicat
- Max^n – shallow pruning, Korf, 1991 (analog to alpha-beta dar cu performante proaste)

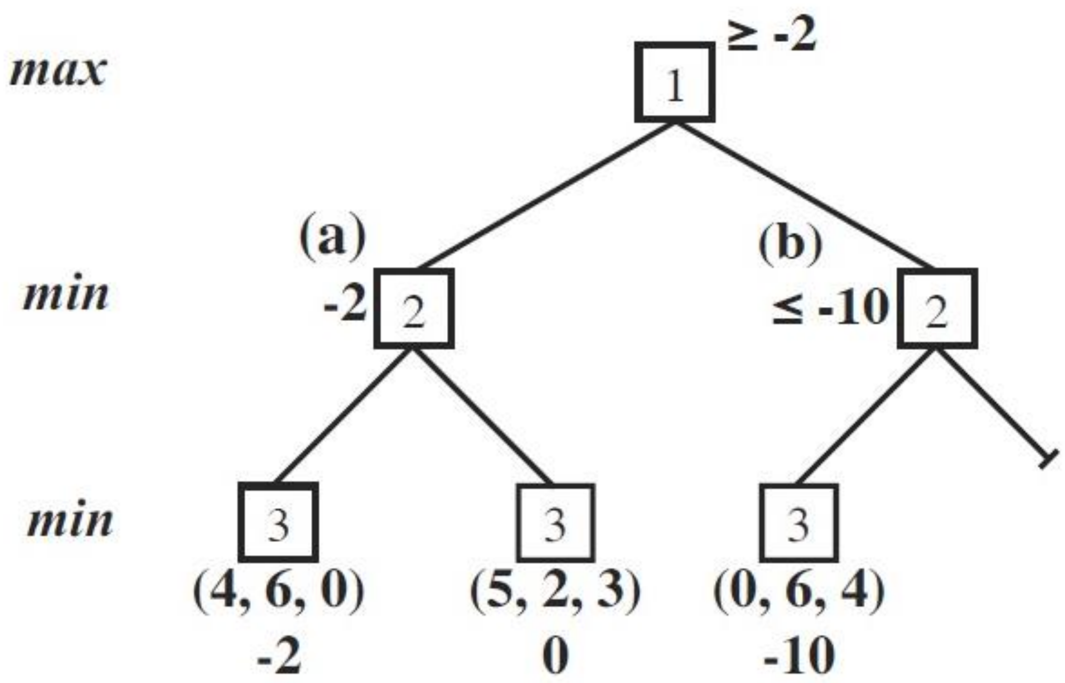
Strategie Paranoic

- **Paranoic** – reduce jocul la 2 jucatori si se poate aplica Minimax



Strategie Paranoic

- La fel ca la Minimax:
  - Exista o unica valoare
  - Se poate utiliza Alpha-Beta



Strategie Paranoic

- Pe masura ce numarul jucatorilor creste beneficiul adus de taiere scade
- Pt jocuri cu 3-6 jucatori, adancimea este cu 20-50% mai mare decat la Max^n
- Presupunerea Paranoic este foarte pesimista
- Exista cazuri in care greseste; in aceasta situatie, cu cat se cauta mai adanc cu atat este mai proasta estimarea

1.3 Monte Carlo Tree Search

- MCTS - algoritm probabilistic care utilizeaza o serie de simulari aleatoare pentru a expanda selectiv arborele de joc
- Reprezinta o metoda buna pentru luarea deciziilor în probleme cu un spatiu de cautare mare
- 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 dintr-o anumita stare pana la sfarsitul jocului, obtinandu-se castig/pierdere sau un scor
- Fiecarui nod parcurs i se asociaza un merit
- In varainta cea mai simpla acest merit este un procent de castig = de cate ori s-a castigat daca s-a pornit unda din acel nod

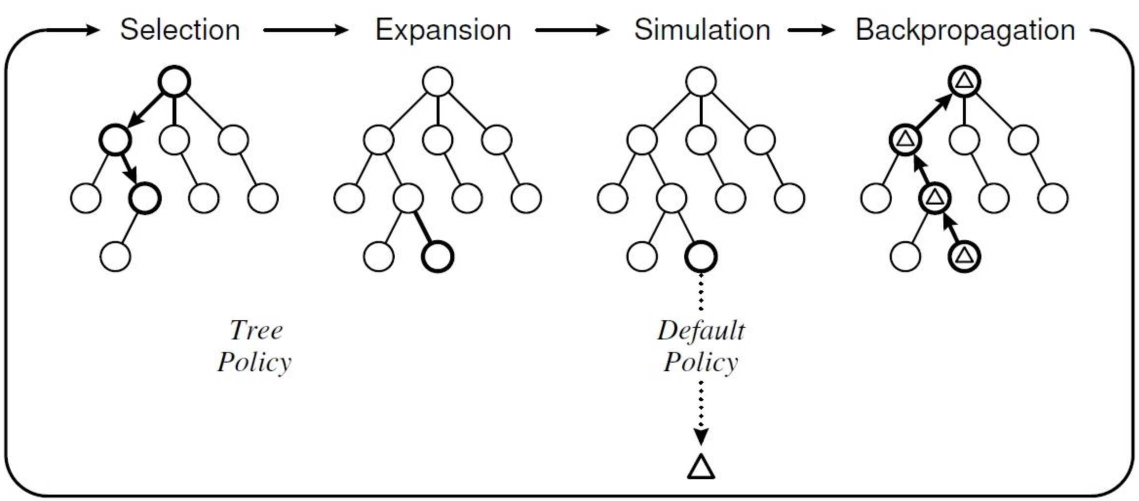
Monte Carlo Tree Search

- 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

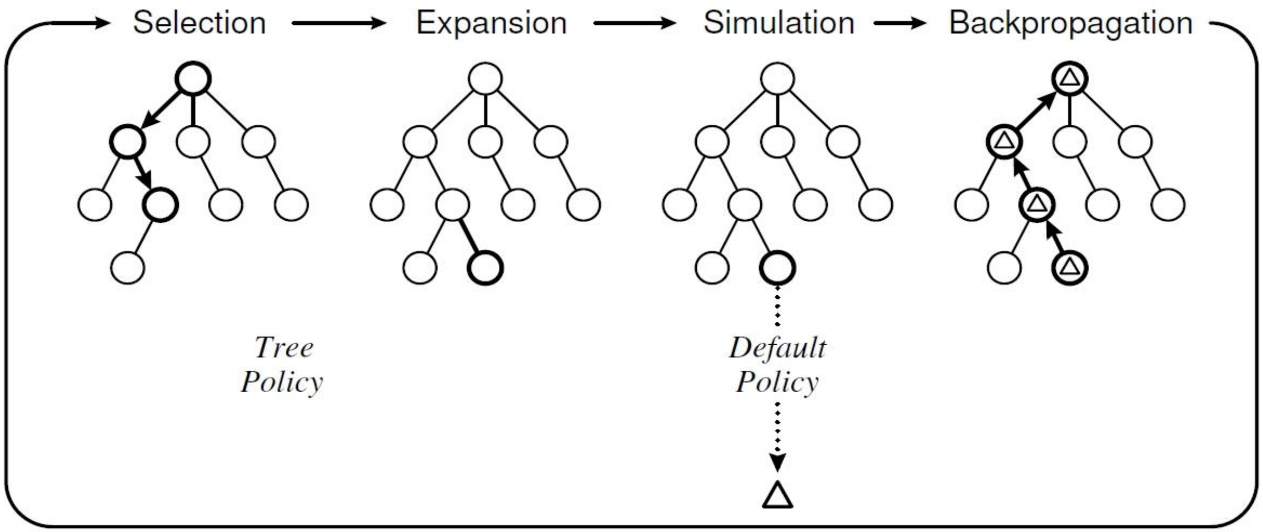
Pentru fiecare iteratie se aplica 4 pasi:

- **Selectie** – Pornind de la radacina o politica de selectie a copiilor este aplicata recursiv pana cand se gaseste cel mai interesant nod neexpandat (un nod E care are un copil ce nu este inca parte a arborelui)
- **Expandare** – Un nod copil (sau mai multe noduri copii) a lui E este adaugate in arbore cf. actiunilor disponibile



Monte Carlo Tree Search

- **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

Creaza nodul radacina v_0 cu starea s_0

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

$v_l \leftarrow$ TreePolicy(v_0)

$\Delta \leftarrow$ DefaultPolicy(v_l)

BackUp(v_l, Δ)

intoarce a(BestChild(v_0))

- **TreePolicy** – construieste un nod frunza din nodurile aflate deja in arborele de cautare; nodul la care se ajunge cu TreePolicy este v_l
- **DefaultPolicy** – joaca jocul dintr-o stare neterminala v_l pentru a produce o estimare a valorii / recompensa (pana stare terminala)
- **a(BestChild(v_0))** – actiunea care selecteaza cel mai bun copil a lui v_0

Algoritm MCTS(Radacina) **intoarce** cea mai buna miscare

Creaza nodul radacina v_0 cu starea s_0

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

$v_l \leftarrow$ TreePolicy(v_0)

$\Delta \leftarrow$ DefaultPolicy(v_l)

BackUp(v_l, Δ)

intoarce a(BestChild(v_0))

TreePolicy(v) **intoarce** un nod

cat timp v este nod neterminal **executa**

daca v nu este complet expandat **atunci** v $\leftarrow$ Expand(v); **break**

altfel v $\leftarrow$ BestChild(v)

intoarce v

Expand(v) intoarce un nod

alege a$\in$actiunile neincercate inca din A(v) /* A(v) act legale in v */

adauga un copil nou v' la v cu v' = next(v,a) /* aleator sau in fct de merit */

intoarce v'

Algoritm MCTS(Radacina) intoarce cea mai buna miscare

Creaza nodul radacina v_0 cu starea s_0

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

$v_l \leftarrow$ TreePolicy(v_0)

$\Delta \leftarrow$ DefaultPolicy(v_l)

BackUp(v_l**,** Δ)

intoarce a(BestChild(v_0))

DefaultPolicy(s) intoarce recompensa

cat timp s este nod neterminal **executa**

alege aleator $a \in A(s)$ /* sau cf unei politici de selectie */

$s \leftarrow$ next(s,a)

intoarce recompensa pentru starea s

BackUp(v, Δ)

cat timp v nu este null **executa**

$N(v) \leftarrow N(v) + 1$ /* numar vizitari nod v */

$Q(v) \leftarrow Q(v) + \Delta(v)$ /* recompensa nod */

$v \leftarrow p$ /* p parintele lui v */

Monte Carlo Tree Search

Ce strategii se folosesc pt fiecare pas?

- **Selectia**
- Problema exploatarii vs explorare
  - Selectam stari din care s-a castigat des si au fost parcurse de multe ori
  - Ori selectam stari cu putine simulari anterioare

- Diferite strategii propuse in literatura

Monte Carlo Tree Search

Exemplu de strategie pentru selectie

Fie I multimea de noduri succesoare nodului curent **p**. Se selecteaza copilul **K** a nodului **p** care satisface formula

$$K \in \arg\max_{i \in I} \left(\overset{\text{Exploatare}}{\frac{Q(i)}{N(i)}} + C * \overset{\text{Explorare}}{\sqrt{\frac{2 * \ln N(p)}{N(i)}}} \right)$$

Q(i) **-** recompensa nodului i

N(i) – numarul de vizitari a nodului i

N(p) – numarul de vizitari a nodului p (parinte)

C – coeficient experimental

BestChild(v) intoarce K

Monte Carlo Tree Search

Expandarea

Se genereaza un nou de unde se va incepe simularea

Simularea

- Total aleator, sau pseudoaleator, sau in functie de o politica

Backpropagation

- Diferite metode

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

$$\Delta(v) = V_p$$

V_{med} – media valorilor nodurilor copii

W_{med} – ponderea acestei medii

V_r – mutarea cu cel mai mare numar de simulari

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

Monte Carlo Tree Search

- Converge spre valorile Minimax, dar lent
- Cu toate acestea mai eficient decat AlfaBeta
- Este un algoritm de tip anytime
- Algoritmul nu gaseste întotdeauna cea mai buna mutare, dar are, în general, un succes rezonabil în cazul alegerii mutarilor care duc la sanse mari de câstig
- Poate sa nu detecteze o ramura care conduce la pierdere (din cauza componentei aleatoare)
- Light playouts – aleator
- Heavy playouts – euristici/politici pentru selectarea miscarii urmatoare (nu strict aleator)

Monte Carlo Tree Search

MoGo

A participat in 30 de turnee intre 2006 si 2010 (9 x 9 GO)

A castigat contra profesionistilor

- MoGo invinge pe campionul Myungwan Kim, august 2008, utilizand MCTS
- **FUEGO -** 2009 – a invins mai multi campioni la 9 x 9 GO

Monte Carlo Tree Search

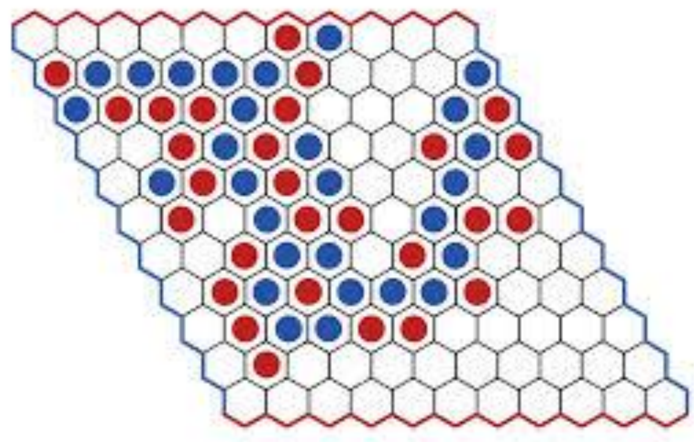
2009 – **MoHex** devine campion la jocul **Hex**

11 x 11

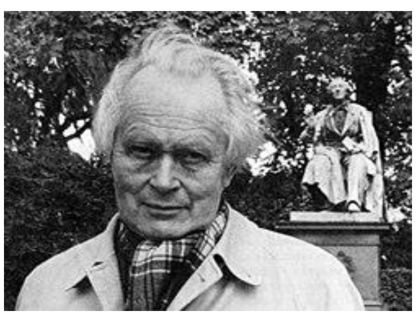
13 x 13

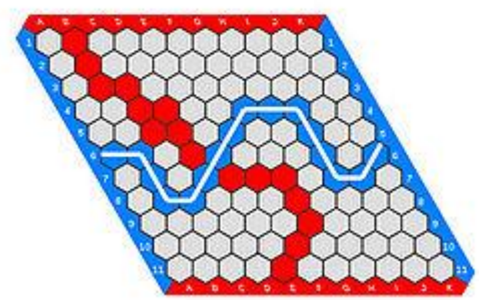
14 x 14

19 x 19



Joc inventat in 1940
de Piet Hein and John Nash





1.4 Invatarea functiei de evaluare pentru selectie

- Functia de evaluare – depinde de parametrii starii si o serie de ponderi ale acestora

$$Eval_w(u)$$

- Daca avem evaluari corecte ale unor stari $Eval^*(u)$ putem aplica **metoda scaderii gradientului** pentru a reduce **eroarea**

$$L = (Eval_w(u) - Eval^*(u))^2$$

$$\Delta w = \alpha\big(Eval^*(u) - Eval_w(u)\big) * \nabla Eval_w(u)$$

Q-Learning

Temporal credit assignment problem

Q-Learning

- **Functie actiune-valoare** = atribuie o utilitate estimata executarii unei actiuni intr-o stare
- Q: A x S $\rightarrow$ U
- Foloseste o matrice (stare,actiune) - valoare

Q-Learning

Foloseste **ecuatia lui Bellman**

$$Q(a,s) \leftarrow Q(a,s) + \alpha(R(s) + \delta \max_{a'} Q(a', s') - Q(a,s))$$

calculata dupa fiecare tranzitie din s in s'.

α - viteza de invatare

Q-Learning

1. Initializeaza **δ** si matricea **R**
2. Initializeaza **Q** la 0 sau aleator
3. Pentru fiecare episod repeta
   - Selecteaza starea initiala **s** aleator
   - **cat timp** **s** nu este stare scop **repeta**
     - obtine recompensa **R(s)**
     - selecteaza o actiune **a** din starea **s**
     - executa trecerea in **s'** cu **a**
     - $Q(a,s) \leftarrow$

       $$Q(a,s) + \alpha(R(s) + \delta \max_{a'} Q(a', s') - Q(a,s))$$
     - $s \leftarrow s'$

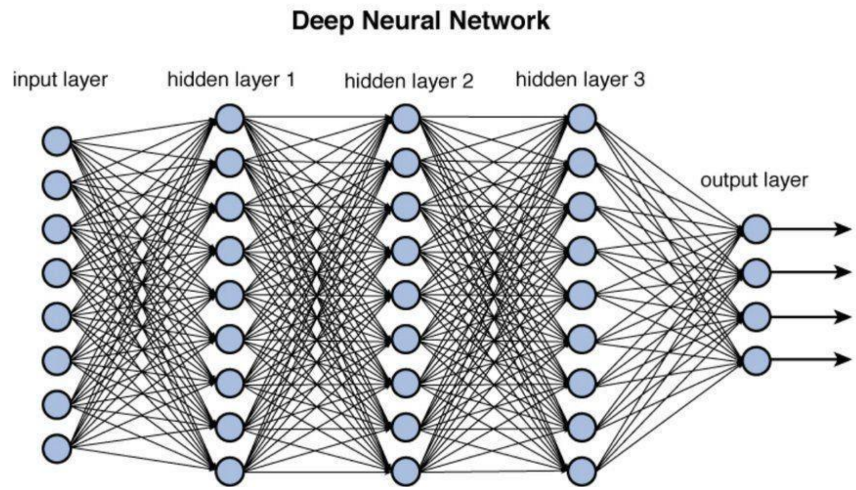
Q-Learning

Utilizare Q

1. s $\leftarrow$ starea initiala
2. gaseste actiunea **a** care maximizeaza Q(a,s)
3. Executa **a**
4. **Repeta** de la 2 **pana** stare scop

Deep Q-Learning

- Combina Q-Learning cu DNN
- Reteaua neurala invata estimarea perechilor stare-actiune numai pe baza descrierii starii si a recompensei, fara a sti regulile de joc

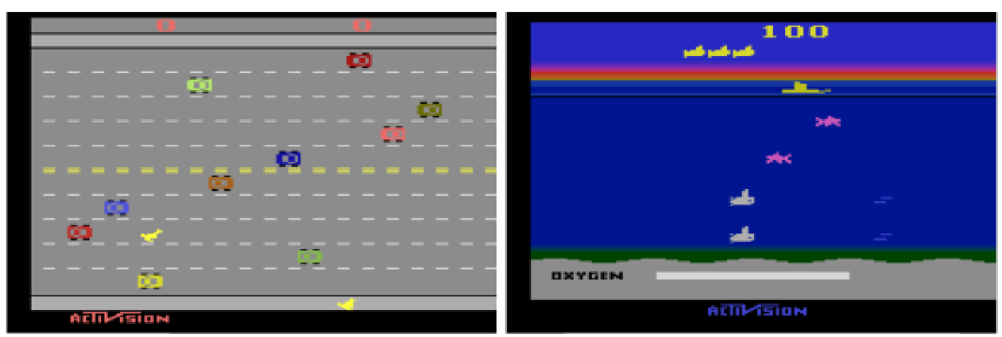


Atari Deep Q-Learning

- Deep learning – extrage caracteristici de nivel inalt din imagini

Atari

- O retea convolutionala este antrenata sa invete evaluarea stare-actiune pt a aplica RL
- Intrare: pixelii din imagine
- Iesire: valori Q



Two classic Atari 2600 video games, “Freeway” and “Seaquest”, from the Arcade Learning Environment

Atari Deep Q-Learning

- Selecteaza actiunea
- Actiunea trimisa emulatorului de joc care modifica starea si scorul jocului
- Agentul primeste o imagine (vectori de pixeli) si recompensa = schimbarea scorului
- Considera o secventa de stari si observatii

$$S_t = s_1, a_1, s_2, a_2, .., a_{t-1}, s_t$$

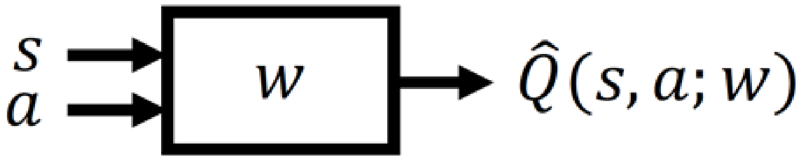
si invata strategia in functie de astfel de secvente

Atari Deep Q-Learning

- Foloseste un Q-network cu parametrii **w** care minimizeaza functia de eroare (**Loss**)

  $L(w) = \text{Estimare}\ [y - Q(s,a,w)]^2$

$$y = R(s) + \delta \max_{a'} Q(s',a')$$



AlphaGo

- Octombrie 2015 - AlphaGo joaca impotriva campionului euroepan Fan Hui si catiga cu un scor de 5-0
- In 2016 castiga in fata lui Lee Sedol, detinator a 18 titluri mondiale
- A obtinut cel mai mare scor
- A inventat mutari
- Foloseste Deep Q-Learning

AlphaGo

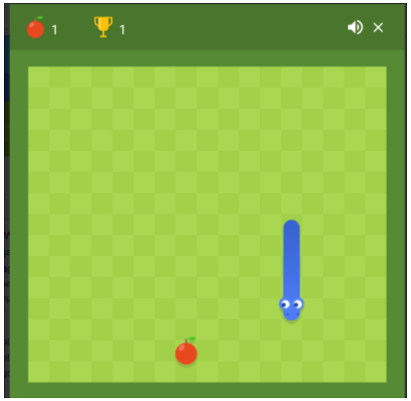
- Foloseste:

- MCTS

- 2 retele neurale adanci: una pentru a invata politica de selectie si una pentru a invata evaluarea nodurilor
- Algoritmul de Q-Learning – initializat cu politica invatata si imbunatatita in functie de recompensele viitoare

Un exemplu simplu

- **Jocul Snake**
- Se dau sistemului parametrii legati de stare si recompensa
- Nu necesita reguli de joc pt invatarea politicii
- Trebuie sa maximizeze recompensa



Un exemplu simplu

- **Tabela Q**

Stare	Right	Left	Up	Down
1	0	0.31	0.12	0.87
2	0.98	-0,21	0.01	0.14

Se actualizeaza cu Q-Learning si ecuatia lui Bellman

$$Q(a,s) \leftarrow Q(a,s) + \alpha(R(s) + \delta \max_{a'} Q(a', s') - Q(a,s))$$

1. Initializeaza **Q** aleator
2. Initializeaza **δ** si matricea **R**
3. Pentru fiecare episod repeta

    Obtine starea curenta **s**

    Executa o actiune **a** (aleator sau selectata de RNA)

    Obtine recompensa **R(s)**

    Executa trecerea in **s'** cu **a**

    $$Q(a,s) \leftarrow$$

    $$Q(a,s) + \alpha(R(s) + \delta \max_{a'} Q(a', s') - Q(a,s))$$

    $$s \leftarrow s'$$

Reprezentare stare pentru retea

11 variabile boolene

- Daca este pericol in jurul sarpelui (Right, Left, Straight)
- Daca sarpele se misca Up, Down, Left, Right
- Daca mancarea este Up, Down, Left, Right

- RNA estimeaza actiunea cea mai buna pentru o anumita stare incercand sa maximizeze recompensa pe baza functiei de eroare (**Loss**)

$$L_i(\theta_i) = [R(s) + \delta \max_{a'} Q(s',a') - Q(s,a,\theta_i)]^2$$