Inteligență Artificială

Bogdan Alexe
bogdan.alexe@fmi.unibuc.ro

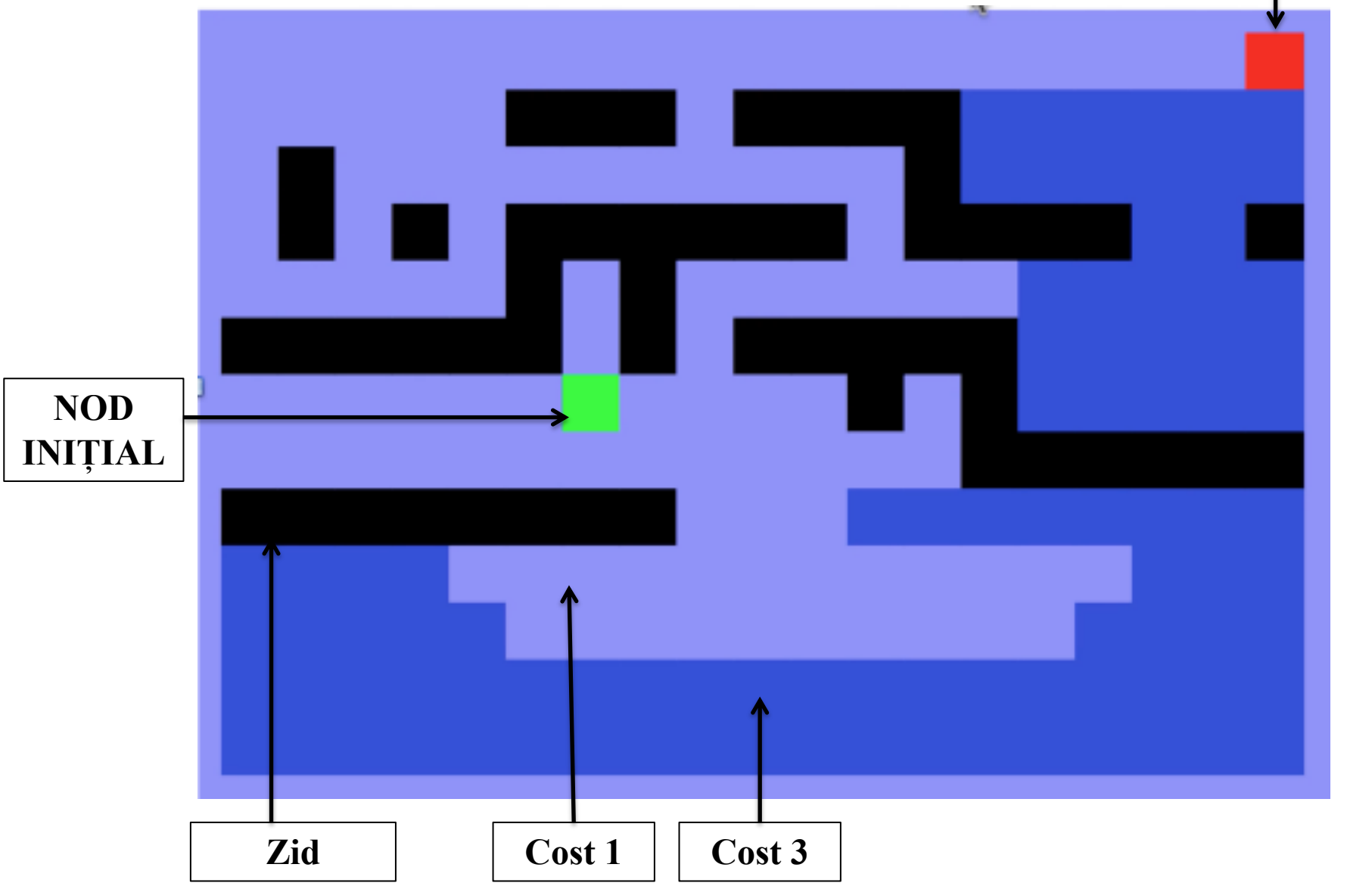
Secția Tehnologia Informației, anul III, 2022-2023
Cursul 11

Recapitulare – cursul trecut

1. Căutare neinformată
   - Căutare în adâncime incrementală (iterative deepening search)
   - Căutare uniformă după cost (uniform-cost search)

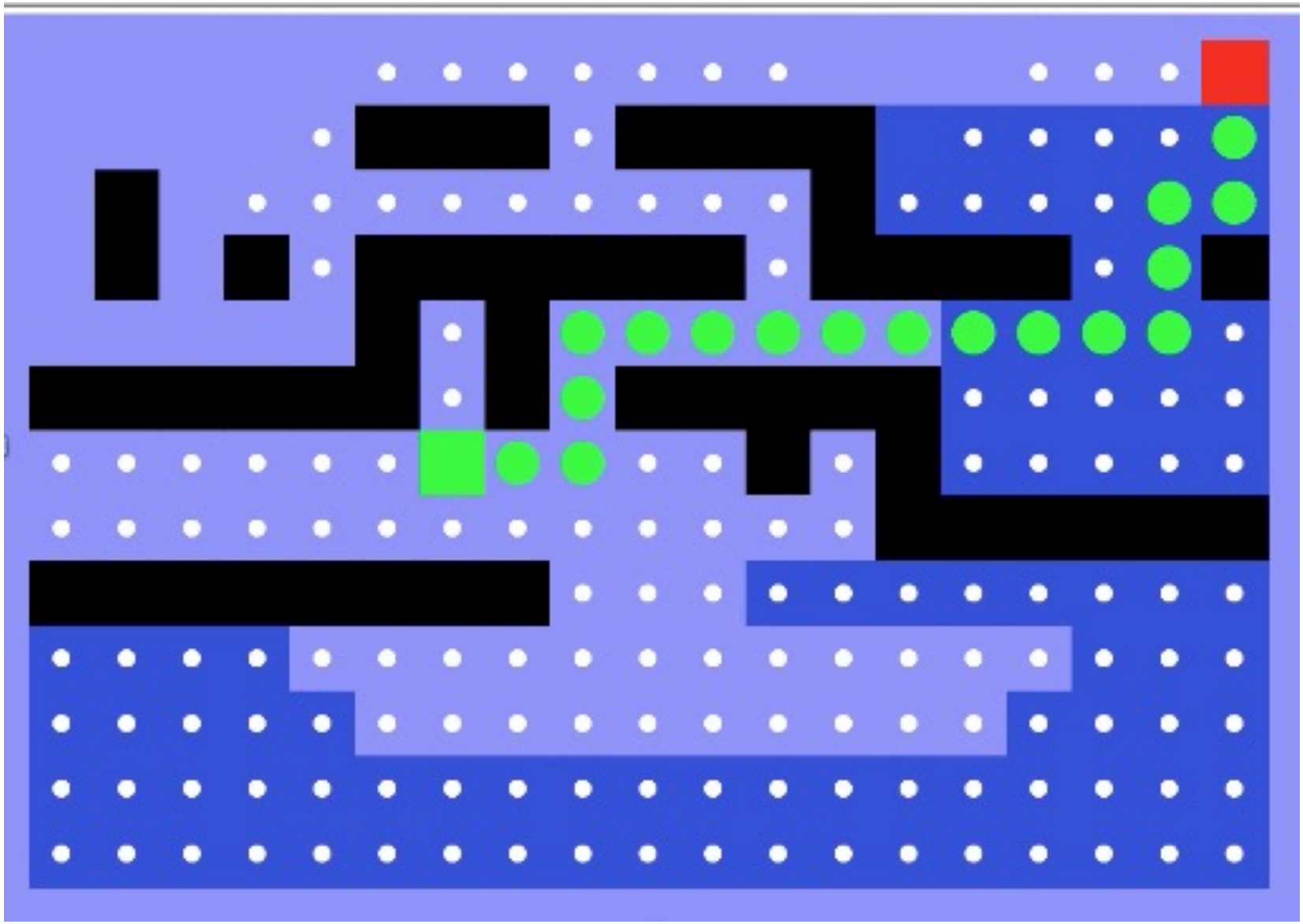
2. Căutare informată
   - căutare Greedy
   - algoritmul A*
   - euristici admisibile, dominante și banale

Labirint - demo



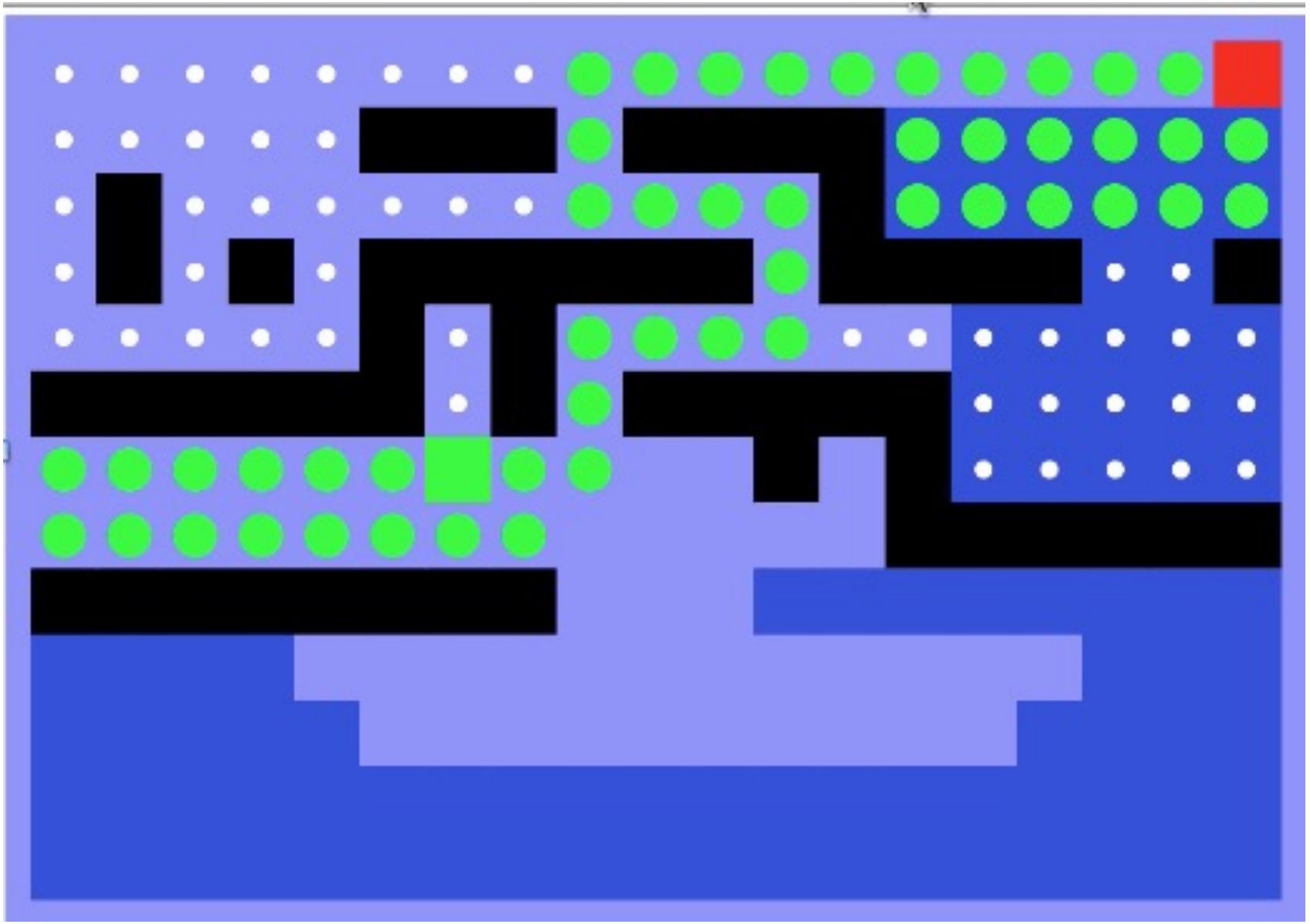
Labirint – demo Bread First search

Labirint – demo Bread First search



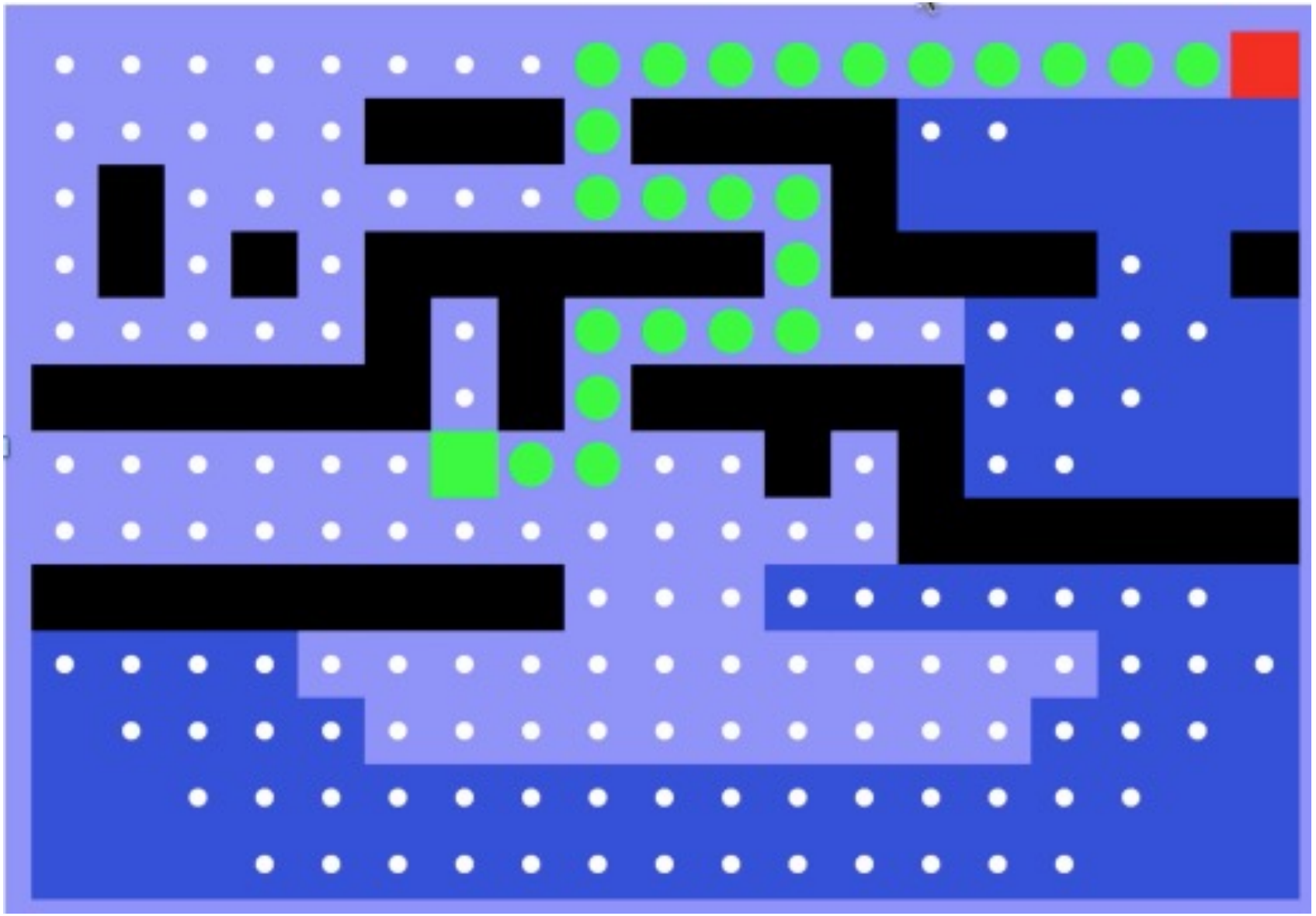
Labirint – demo Depth First search

Labirint – demo Depth First search



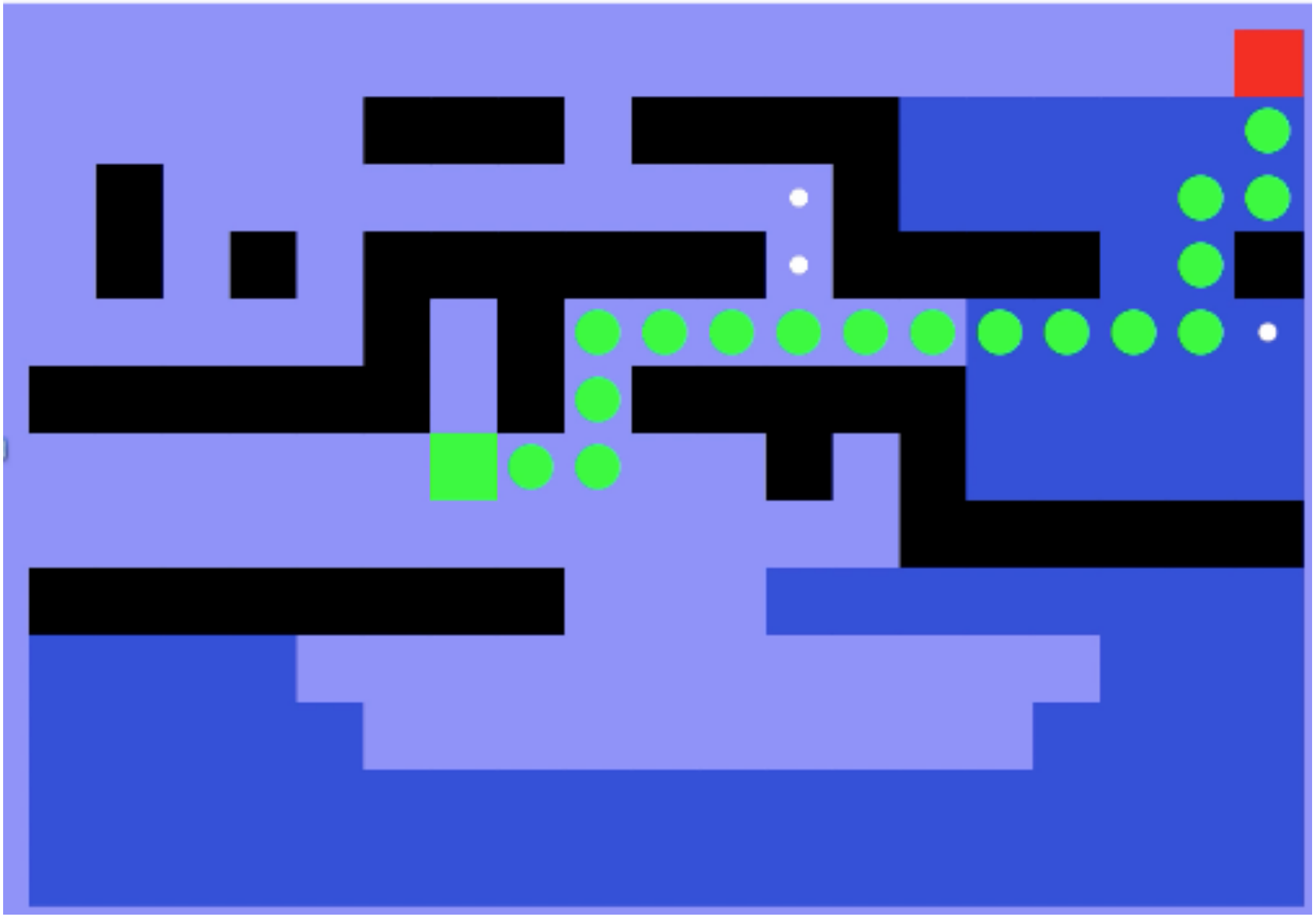
Labirint – demo Uniform Cost Search

Labirint – demo Uniform Cost Search



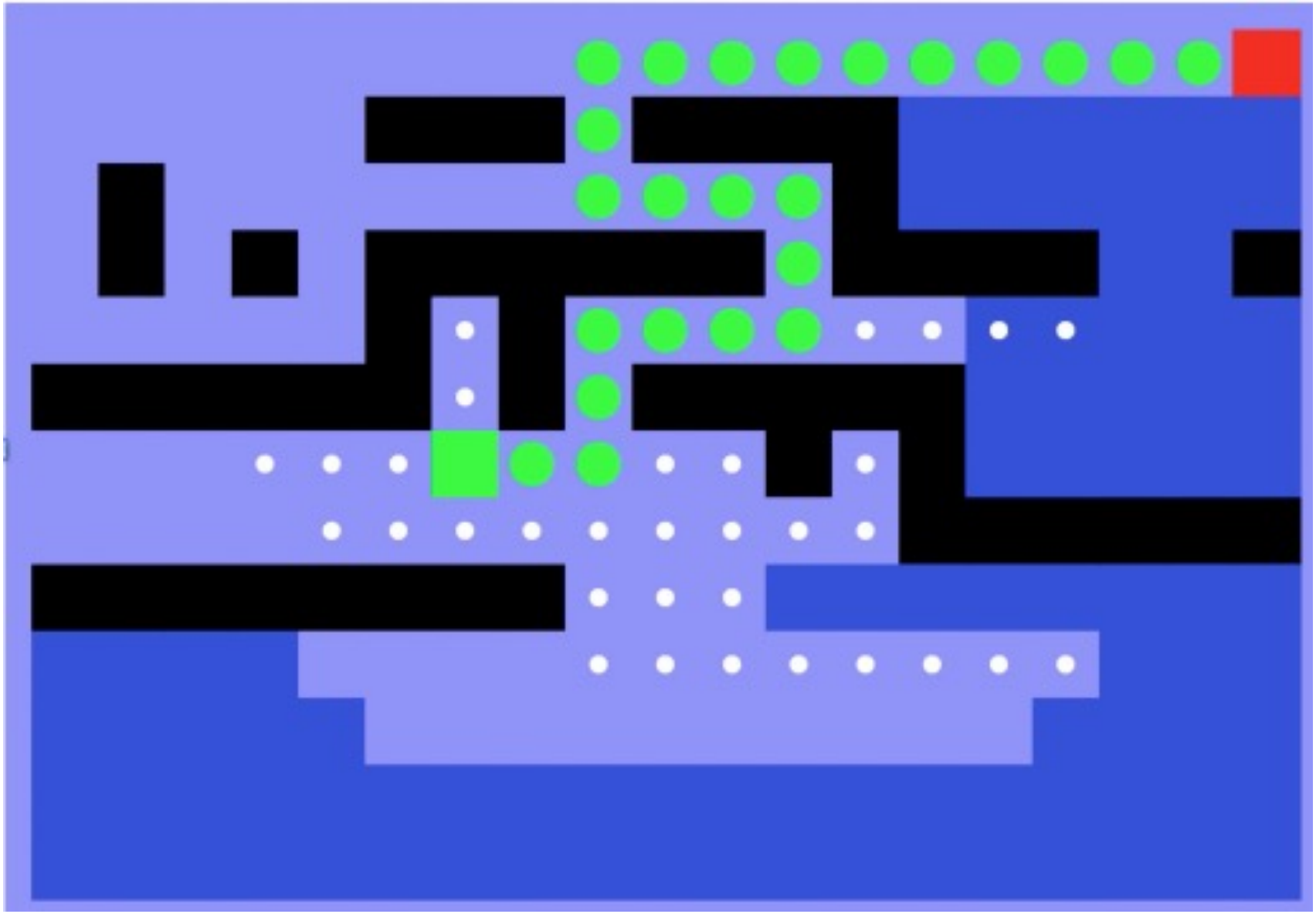
Labirint – demo Greedy search

Labirint – demo Greedy search



Labirint – demo A*

Labirint – demo A*



Cuprinsul cursului de azi

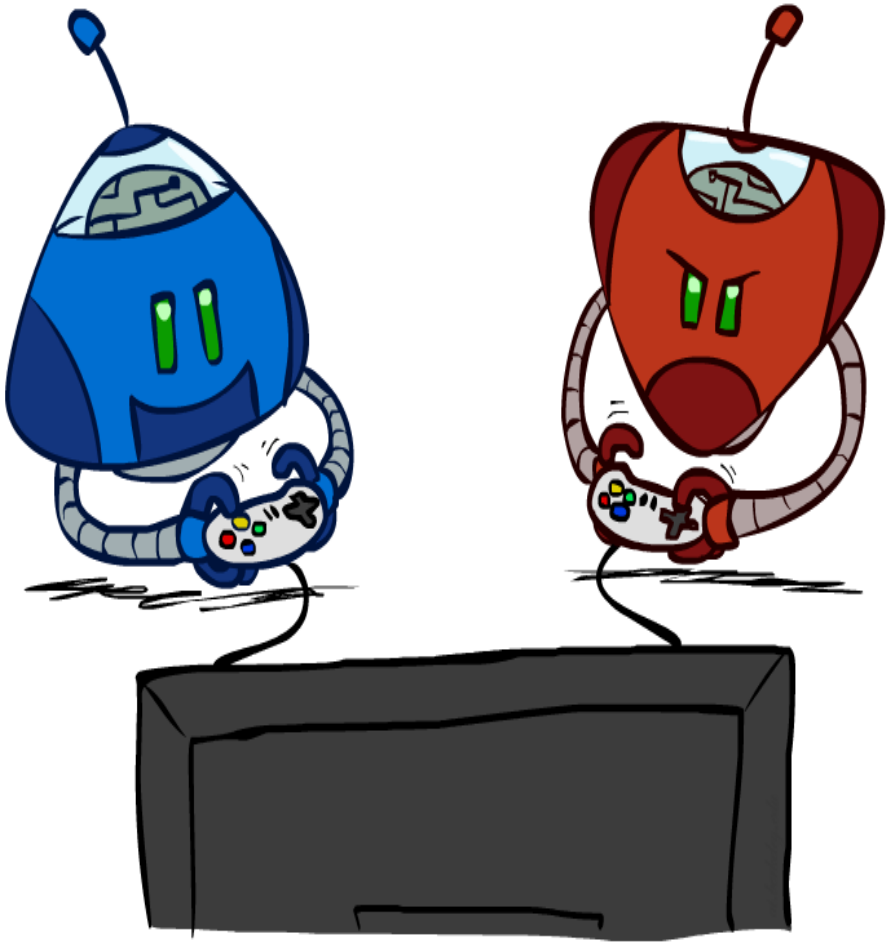
1. Căutare adversarială:
   - jocuri adversariale
   - algoritmul MINIMAX
   - algoritmul α-β retezare (α-β prunning)

Un software al DeepMind a depășit o nouă barieră. DeepMind, compania de cercetare în domeniul inteligenței artificiale deținută de Alphabet (compania-mamă a Google), a dezvoltat un program care poate învăța jocuri complexe fără a cunoaște în prealabil regulile lor.

Cei de la DeepMind s-au remarcat anterior prin progresele revoluționare înregistrate în dezvoltarea unor programe care să învețe jocuri precum Go sau Shogi (un alt joc de strategie chinezesc), precum și în cazul șahului sau a unor jocuri video dezvoltate de compania Atari. Însă în toate cazurile anterioare realizările s-au bazat pe cunoașterea de la început a regulilor jocurilor respective.

Noul software MuZero dezvoltat de DeepMind a reușit însă acest lucru fără a cunoaște de la început regulile, programatorii companiei bazându-se pe un principiu numit „căutare anticipată" (look-ahead search). Acesta evaluează un număr de mutări potențiale în funcție de maniera în care un adversar ar răspunde la ele. Deși exista un număr incredibil de mare de permutări posibile în jocuri complexe precum șahul, MuZero acordă prioritate celor mai relevante și probabile mișcări, învățând din partidele de succes și evitându-le pe cele pierzătoare.

Căutare adversarială



Proprietăți ale mediilor în care funcționează agenții

- Complet observabil vs parțial observabil
- Determinist vs stochastic
- Discret vs continuu
- Episodic vs secvențial
- Static vs dinamic
- Agent vs multiagent

Proprietăți ale mediilor în care funcționează agenții

- Complet observabil vs parțial observabil: la orice moment în timp mediul este complet observabil dacă senzorii agentului îi dau acces complet la starea mediului la fiecare moment și îi oferă toate aspectele relevante în luarea deciziei optime.
  - șah vs poker

- Determinist vs stochastic: dacă următoarea stare a mediului este în întregime determinată de starea curentă și de acțiunile agentului, mediul este determinist.
  - șah vs table (dau cu zarurile)

Proprietăți ale mediilor în care funcționează agenții

- Discret vs continuu: dacă numărul de percepții și de acțiuni ale agentului sunt finite atunci mediul este discret.
  - șah vs condusul unei mașini

- Episodic vs secvențial: într-un mediu episodic, experiența agentului este împărțită în episoade independente. Un episod constă în percepție urmată de o singură acțiune. Episoadele următoare nu depind de acțiunile episoadelor anterioare.
  - corectarea unui test grilă vs șah

Proprietăți ale mediilor în care funcționează agenții

- Static vs dinamic: dacă mediul se schimbă cât timp agentul decide ce acțiuni să facă mediul este dinamic.
  - rezolvarea unui rebus vs condusul unei mașini

- Agent vs multiagent: dacă sunt mai mulți agenți mediul este multiagent (sisteme multiagent competitive vs sisteme multiagent cooperative).
  - corectarea unui test grilă vs șah

Jocul de dame

Complet observabil vs parțial observabil
Determinist vs stochastic
Discret vs continuu
Episodic vs secvențial
Static vs dinamic
Agent vs multiagent

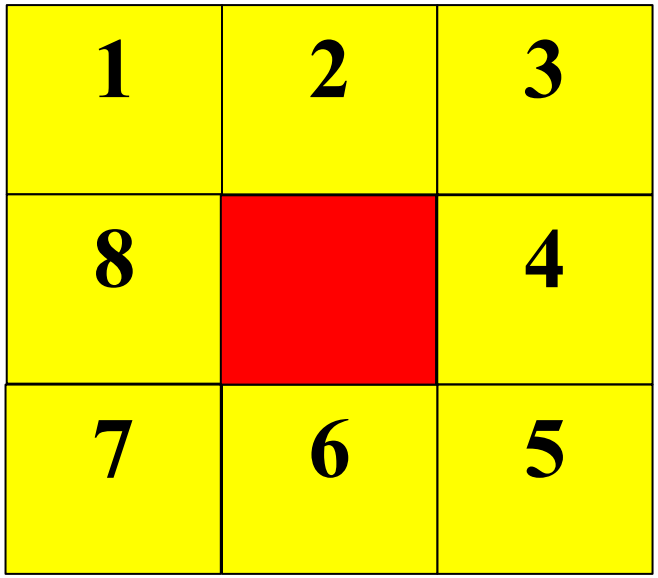
Complet observabil
Determinist
Discret
Secvențial
Static
Multiagent competitiv

Jocul de poker

Complet observabil vs parțial observabil
Determinist vs stochastic
Discret vs continuu
Episodic vs secvențial
Static vs dinamic
Agent vs multiagent

Parțial observabil
Stochastic
Discret
Secvențial
Static
Multiagent competitiv

8 - puzzle



Complet observabil vs parțial observabil
Determinist vs stochastic
Discret vs continuu
Episodic vs secvențial
Static vs dinamic
Agent vs multiagent

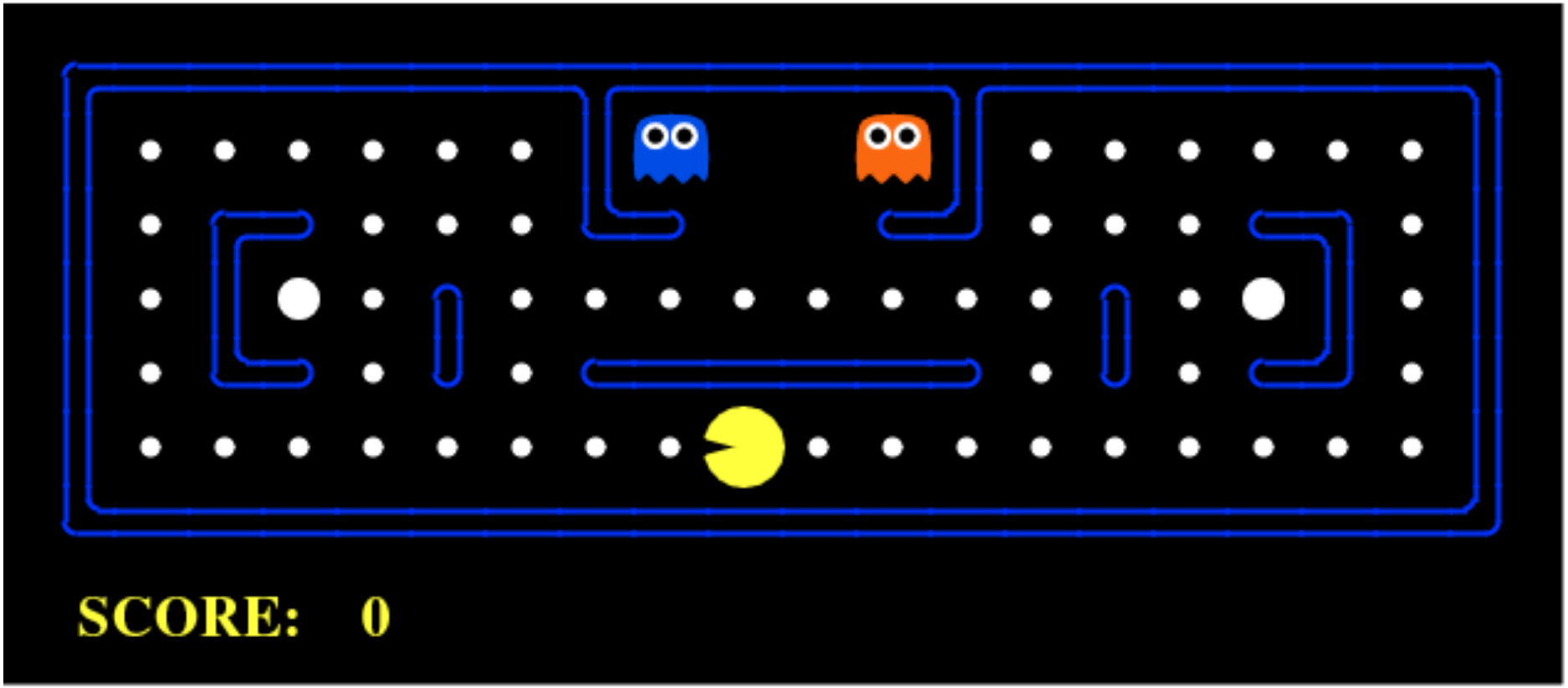
Complet observabil
Determinist
Discret
Secvențial
Static
Agent

Mașină autonomă

Complet observabil vs parțial observabil
Determinist vs stochastic
Discret vs continuu
Episodic vs secvențial
Static vs dinamic
Agent vs multiagent

Parțial observabil
Stochastic
Continuu
Secvențial
Dinamic
Multiagent cooperativ

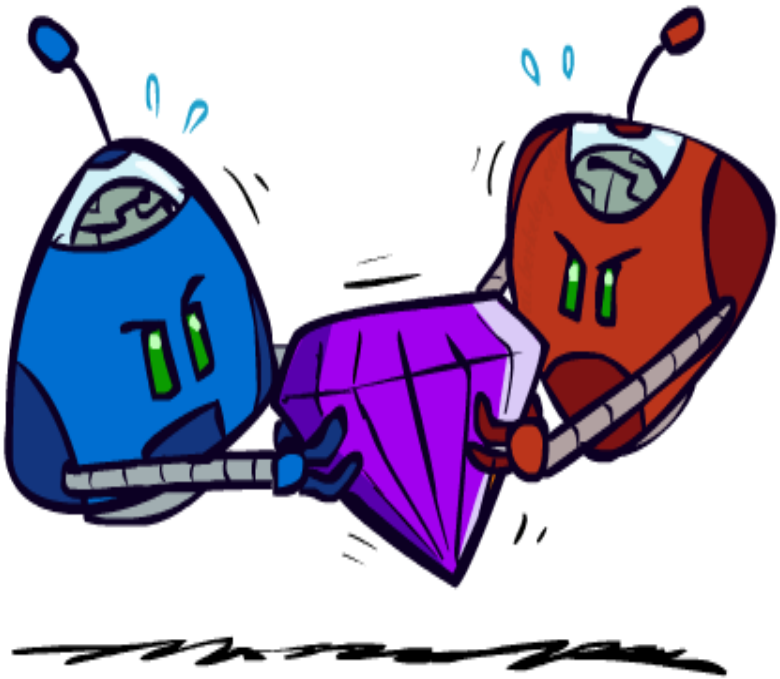
PAC MAN



Complet observabil vs parțial observabil
Determinist vs stochastic
Discret vs continuu
Episodic vs secvențial
Static vs dinamic
Agent vs multiagent

Complet observabil
Stochastic
Discret
Secvențial
Dinamic
Multiagent competitiv

Jocuri adversariale



Jocuri

Medii multi-agent:

- fiecare agent trebuie să considere acțiunile celorlalți agenți
- adversar imprevizibil, incontrolabil
- mediu competitiv

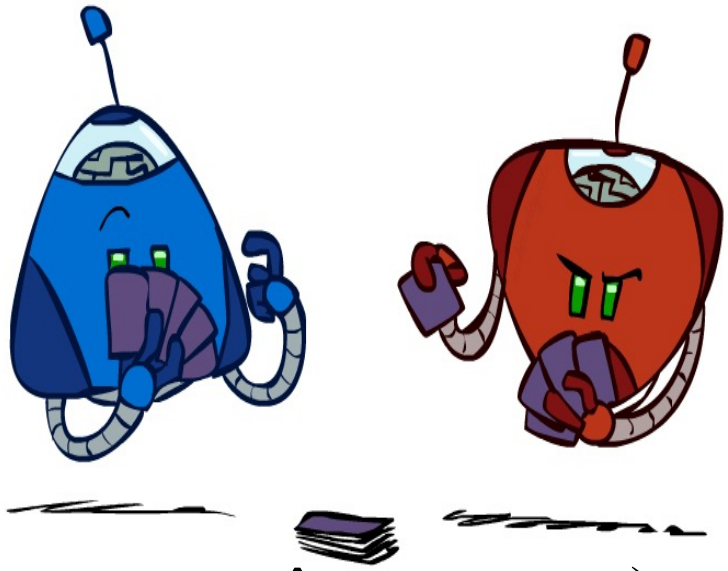
➔Căutare adversarială = joc

Soluția este o strategie (policy) care specifică mutarea pentru fiecare mutare posibilă a adversarului

Probleme de căutare adversarială sunt diferite de probleme de căutare, unde soluția este o secvență de acțiuni care garantează obținerea scopului.

Tipuri de jocuri

- multe tipuri de jocuri!



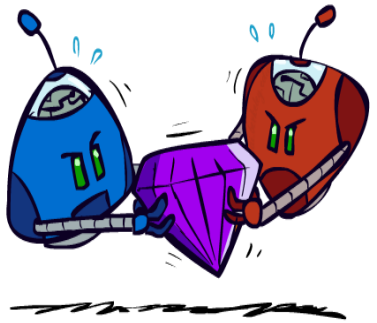
- dimensiuni:
  - deterministic sau stocastic?
  - unul, doi, sau mai mulți jucători?
  - joc de sumă zero?
  - complet sau parțial observabil? (se cunoaște întreaga stare)

- vrem să găsim algoritmi pentru a calcula o strategie ce recomandă o mutare din fiecare stare

Tipuri de jocuri

	determinist	stocastic
Complet observabil	sah, dame, go	table, monopoly
Parțial observabil	avioane	poker, catan

- dimensiuni:
  - deterministic sau stocastic?
  - unul, doi, sau mai mulți jucători?
  - joc de sumă zero?
  - complet sau parțial observabil? (se cunoaște întreaga stare)

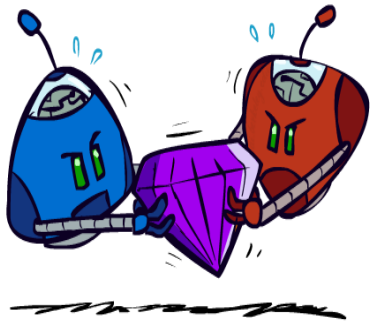


- vrem să găsim algoritmi pentru a calcula o strategie ce recomandă o mutare din fiecare stare

Tipuri de jocuri

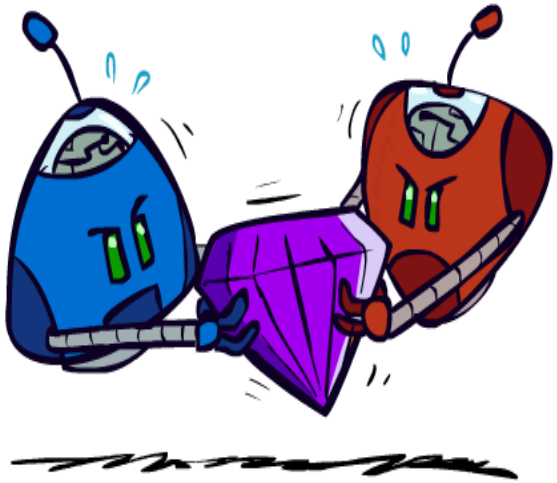
	determinist	stocastic
Complet observabil	sah, dame, go	table, monopoly
Parțial observabil	avioane	poker, catan

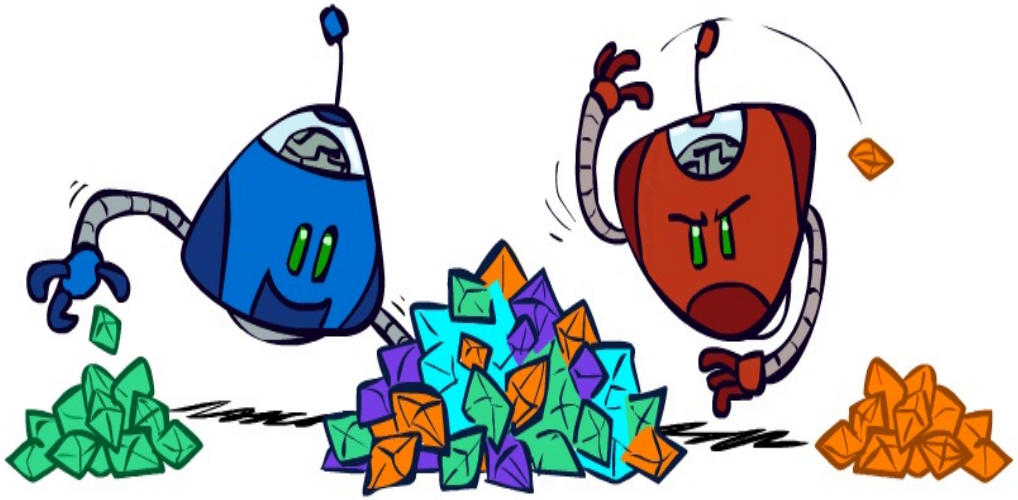
- dimensiuni:
  - deterministic sau stocastic?
  - unul, doi, sau mai mulți jucători?
  - joc de sumă zero?
  - complet sau parțial observabil? (se cunoaște întreaga stare)



- vrem să găsim algoritmi pentru a calcula o strategie ce recomandă o mutare din fiecare stare

Jocuri cu sumă zero





- Jocuri cu sumă zero
  - agenții au utilități (evaluari ale rezultatelor) opuse
  - un agent dorește maximizarea utilității (agentul MAX), celălalt minimizarea (agentul MIN)
  - mediu adversarial pur competitiv
- Jocuri generale
  - agenții au utilități independente
  - cooperare, indiferență, competiție

O posibilă formulare a problemei

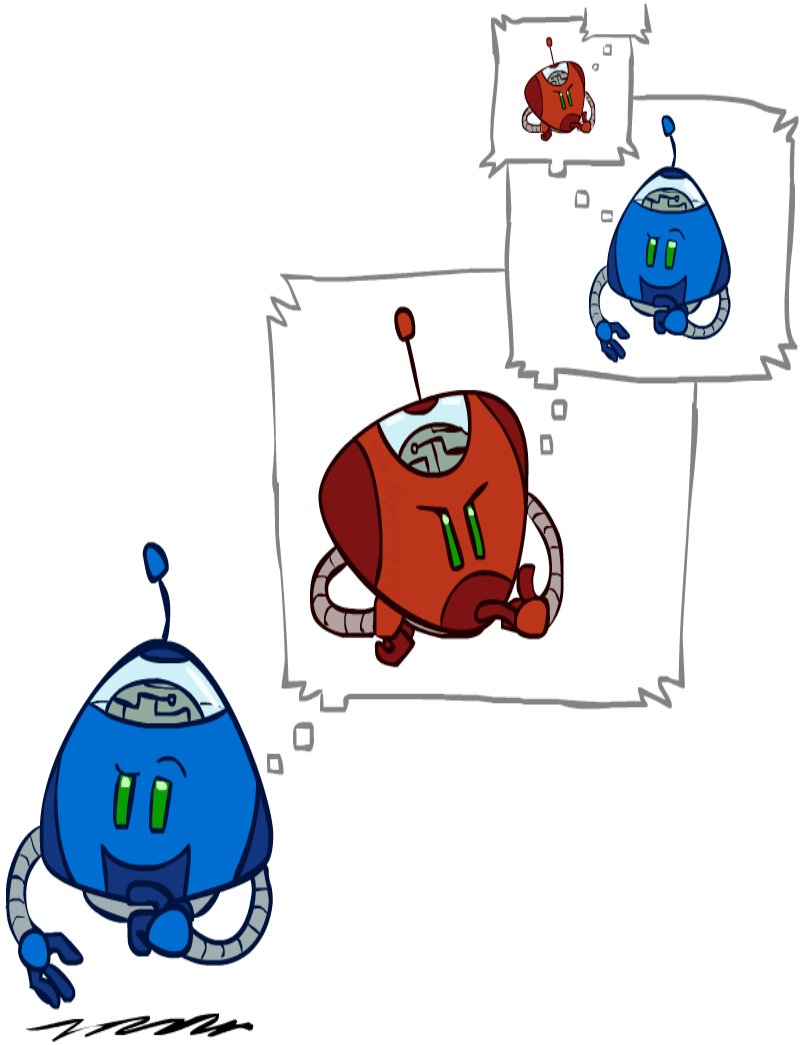
Jocuri cu doi jucători (MAX și MIN); la finalul jocului punctele sunt ale câștigătorului, penalitățile sunt ale pierzătorului (joc de sumă zero)

- stare initiala: s_0
- jucătorul care trebuie să mute într-o stare: PLAYER(s)
- acțiuni (mutări) legale într-o stare: ACTIONS(s)
- modelul de tranziție (funcția succesor): RESULT(s,a)
- test terminal : TERMINAL-TEST(s)
- o funcție de utilitate – asociază o valoare numerică stărilor terminale: UTILITY(s)

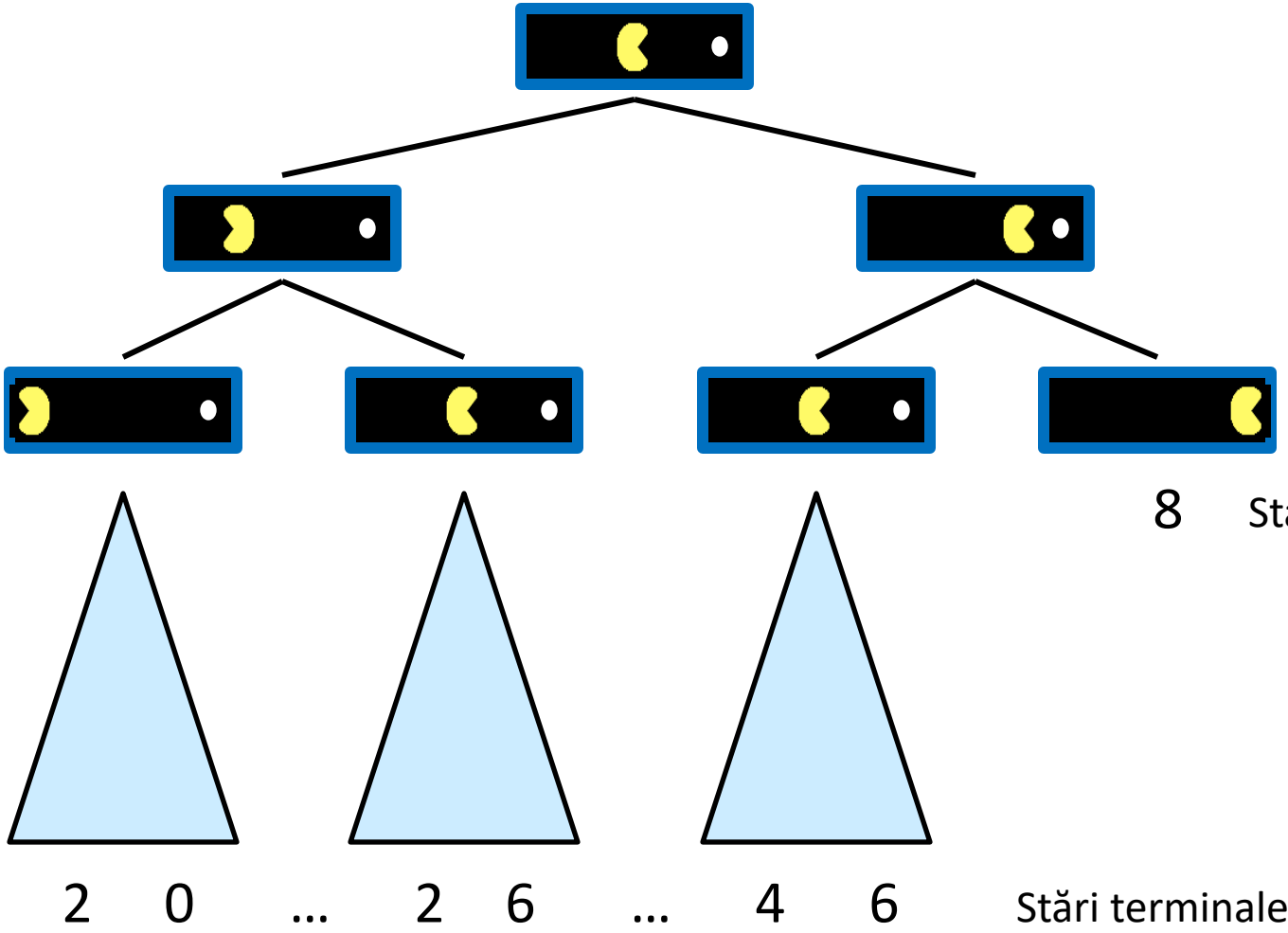
Șah: funcția de utilitate: 0 – remiză, +1 câștig, -1 pierdere.

Există și jocuri cu funcție de utilitate diferită.

Găsirea unei strategii optime



Arbore de căutare într-un mediu cu un singur agent



Fiecare mutare: -1 punct

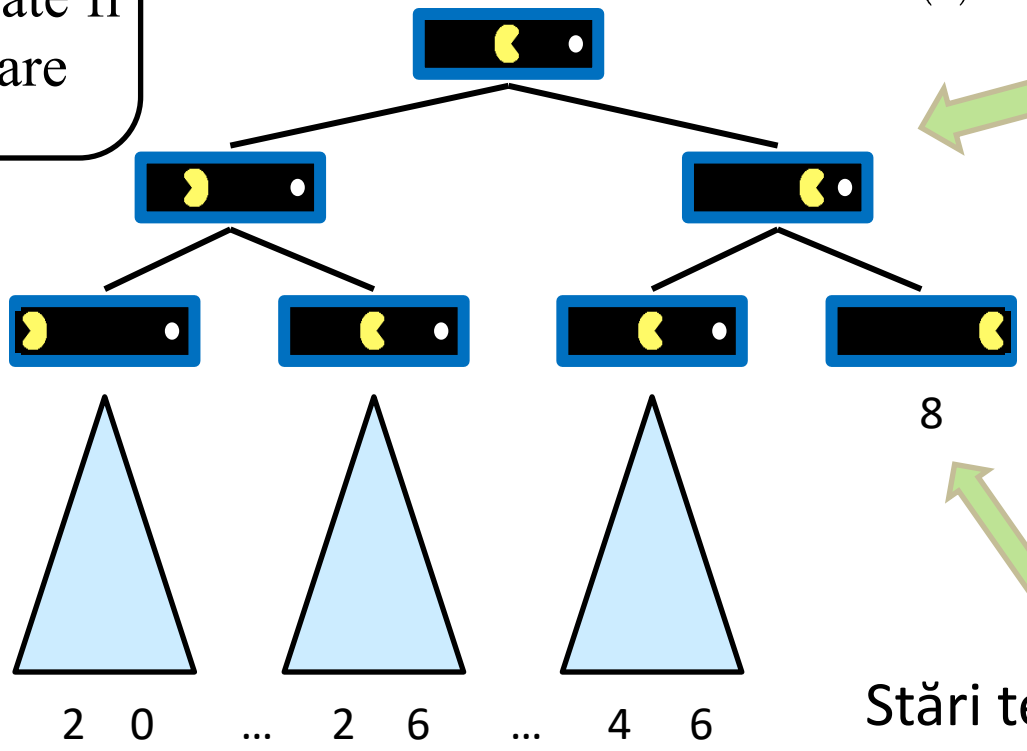
Mânânc punctul alb: + 10 puncte

Valoarea unei stări

Valoarea unei stări: cea mai bună utilitate (rezultat) care poate fi atinsă din acea stare

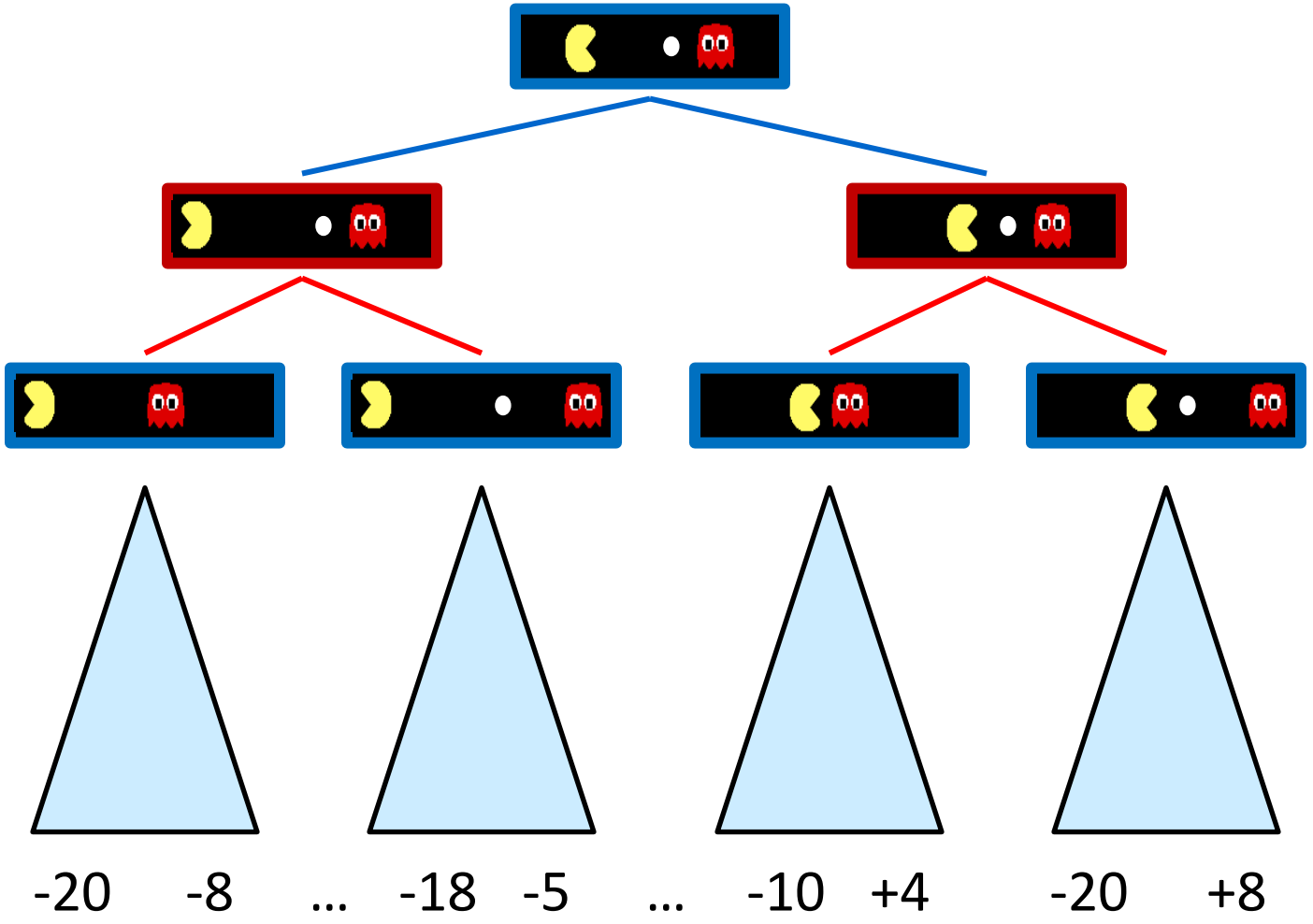
Stări neterminale:

$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$



Stări terminale:
V(s) = cunoscută, dată de regulile jocului

Arbore de joc în medii cu doi agenți, deterministic, mutări pe rând



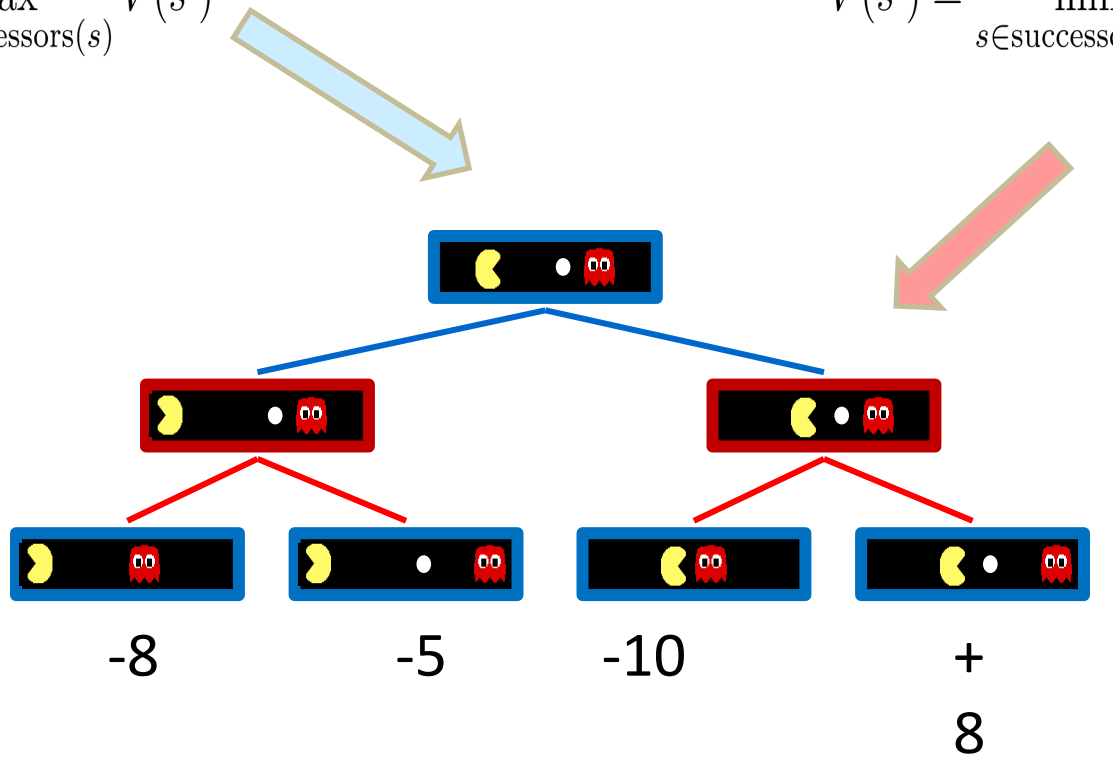
Valoare MiniMax a unei stări

Stări controlate de MAX

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

Stări controlate de MIN:

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

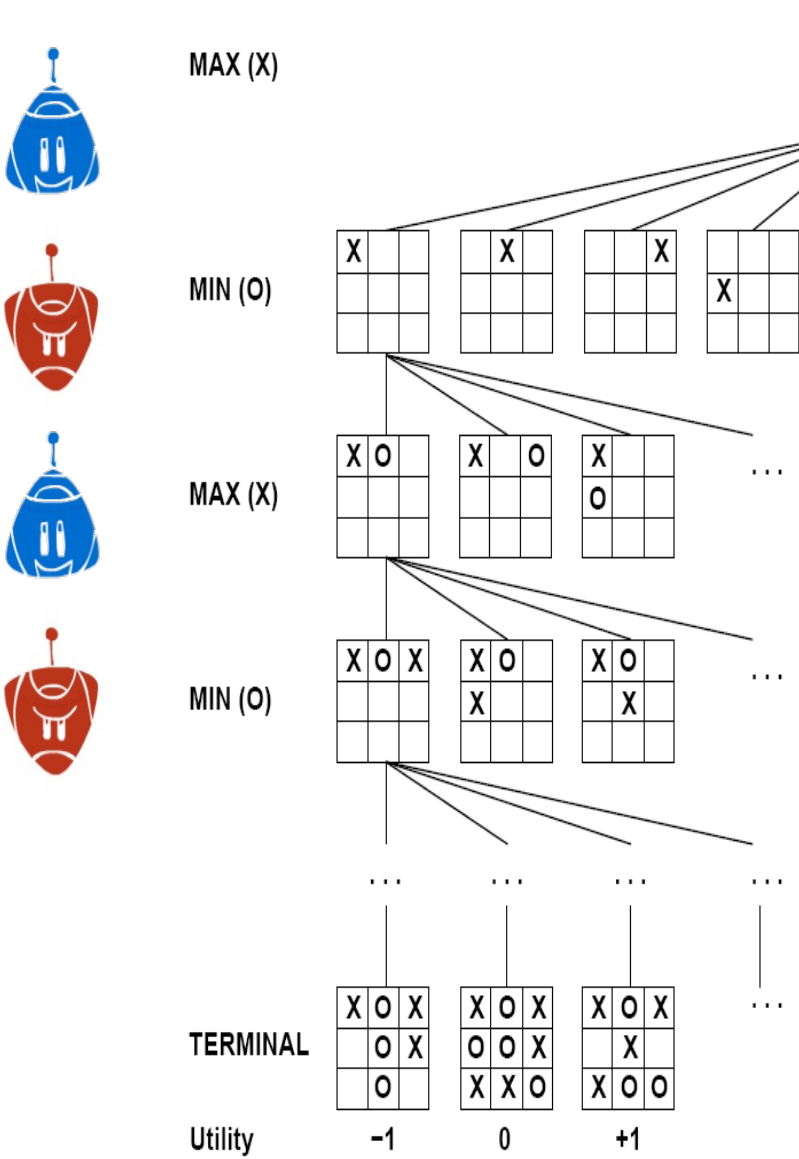


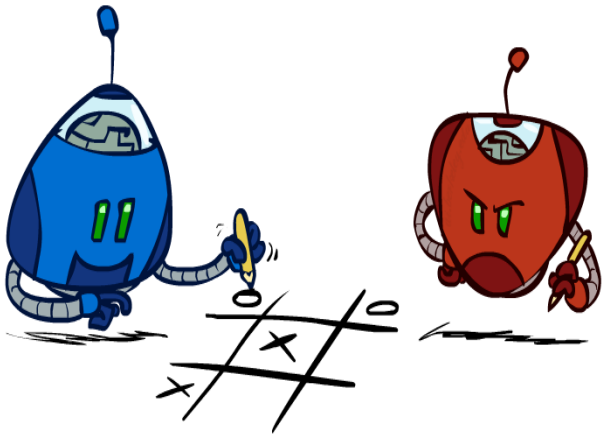
Stări terminale: V(s) = cunoscută, dată de regulile jocului

Arbore de joc

- nodurile corespund situațiilor (stărilor)
- situația inițială a jocului dată de nodul rădăcină
- arcele corespund mutărilor
- frunzele corespund situaților terminale (s-a terminat jocul)
- 2 jucători: MAX (de obicei este cel care face prima mutare) și MIN

Arbore de joc pentru X și 0

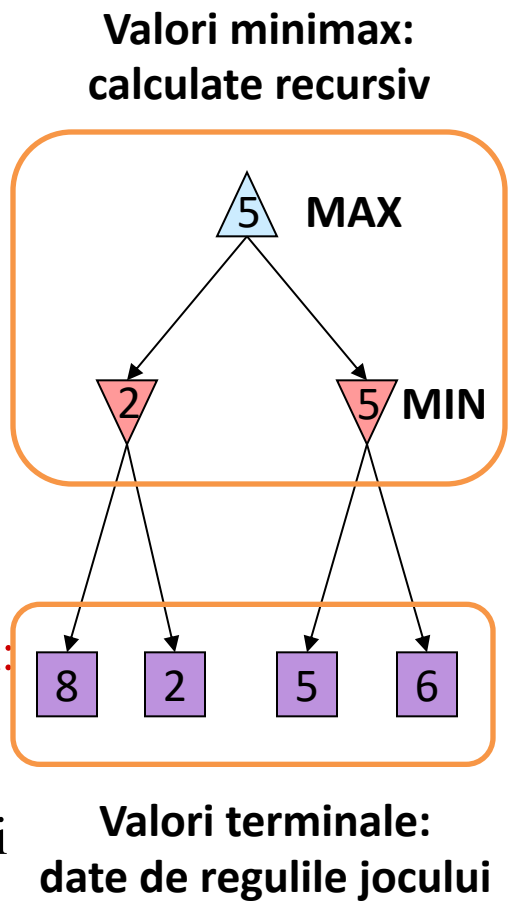




Algoritmul MiniMax pentru căutare adversarială

- Jocuri de sumă zero cu doi jucători, deterministe, cu informație perfectă
  - X și 0, șah, dame
  - un jucător (MAX) maximizează utilitatea
  - celălalt jucător (MIN) minimizează utilitatea

- Căutare MiniMax:
  - arbore de căutare în spațiul stărilor
  - jucătorii mută pe rând
  - calculăm pentru fiecare nod valoarea minimax: cea mai bună utilitate care poate fi obținută împotriva unui adversar care joacă perfect
  - determină strategia optimă corespunzătoare lui MAX, care este cea mai bună primă mutare



Algoritmul MiniMax - descriere

1. Generează întreg arborele de joc până la stările terminale

2. Aplică funcția de utilitate fiecărei stării terminale - obține valoarea 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 fii săi;

   - dacă starea-părinte este un nod de tip MIN, atribuie-i minimul dintre valorile avute de fii săi;

4. Ajuns în nodul-rădăcină, alege pentru MAX acea mutare care conduce la valoarea maximă. Mutarea se numește ***decizia minimax*** - maximizează utilitatea, în ipoteza că oponentul joacă perfect cu scopul de a o minimiza.

Algoritmul MiniMax - formalizare

- jucătorul care trebuie să mute într-o stare: PLAYER(s)
- acțiuni (mutări) legale într-o stare: ACTIONS(s)
- modelul de tranziție (funcția succesor): RESULT(s,a)
- test terminal : TERMINAL-TEST(s)
- o funcție de utilitate – asociază o valoare numerică stărilor terminale: UTILITY(s)

$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \end{cases}$$

Implementare MiniMax

```
def value(stare):
    dacă stare este o stare terminală: returnează utilitatea stării
    dacă următorul agent care mută este MAX: returnează max-value(stare)
    dacă următorul agent care mută este MIN: returnează min-value(stare)
```

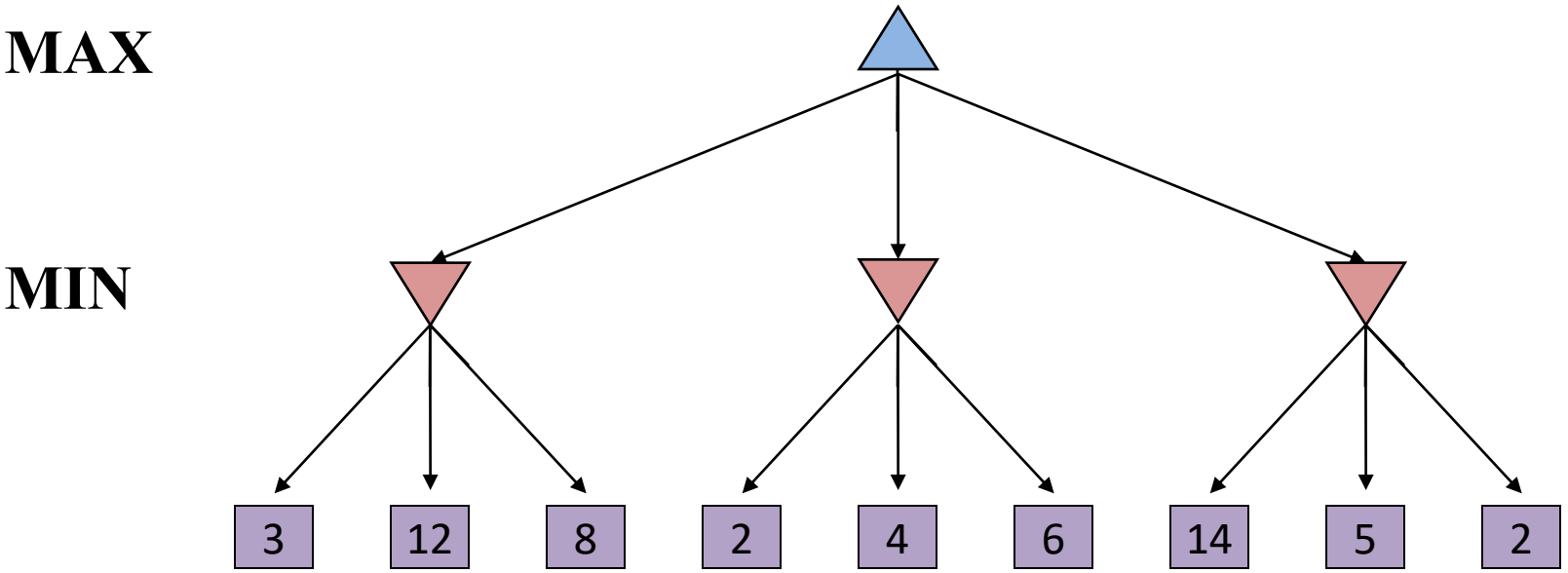
```
def max-value(stare):
    initializează v = -∞
    pentru fiecare successor al starii:
        v = max(v, value(successor))
    returnează v
```

```
def min-value(state):
    initializează v = +∞
    pentru fiecare succesor al stării:
        v = min(v, value(successor))
    returnează v
```

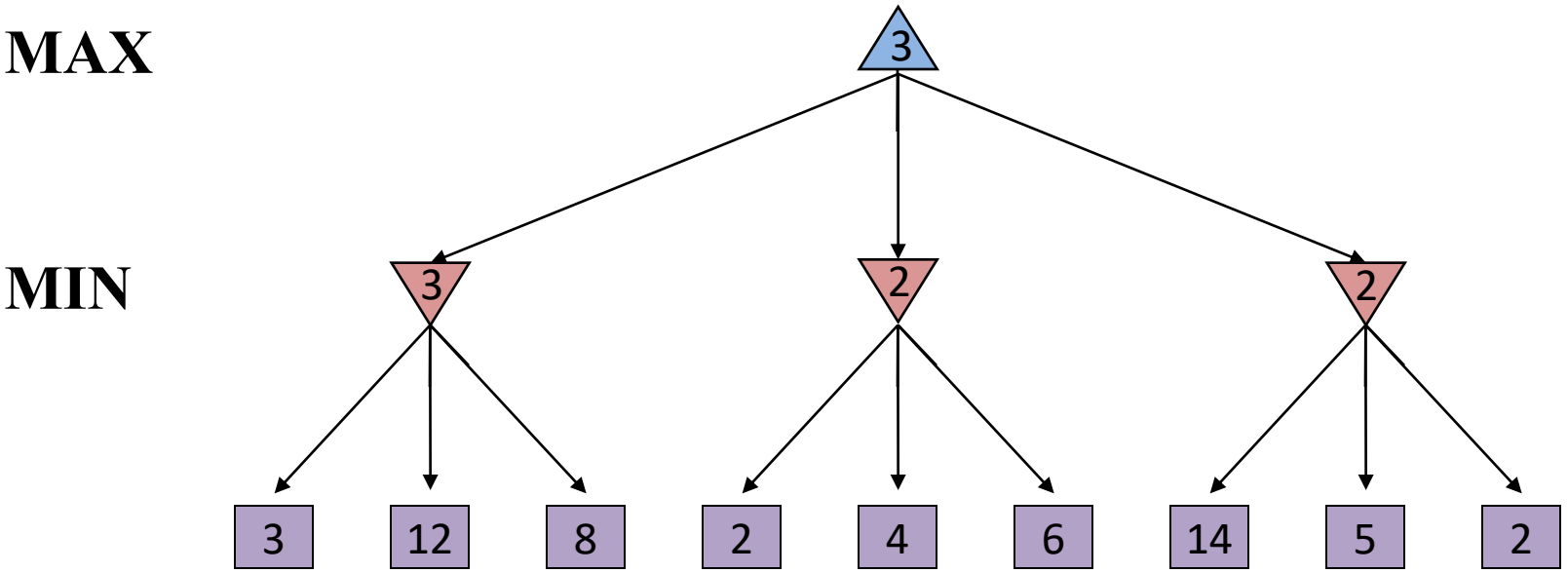
$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

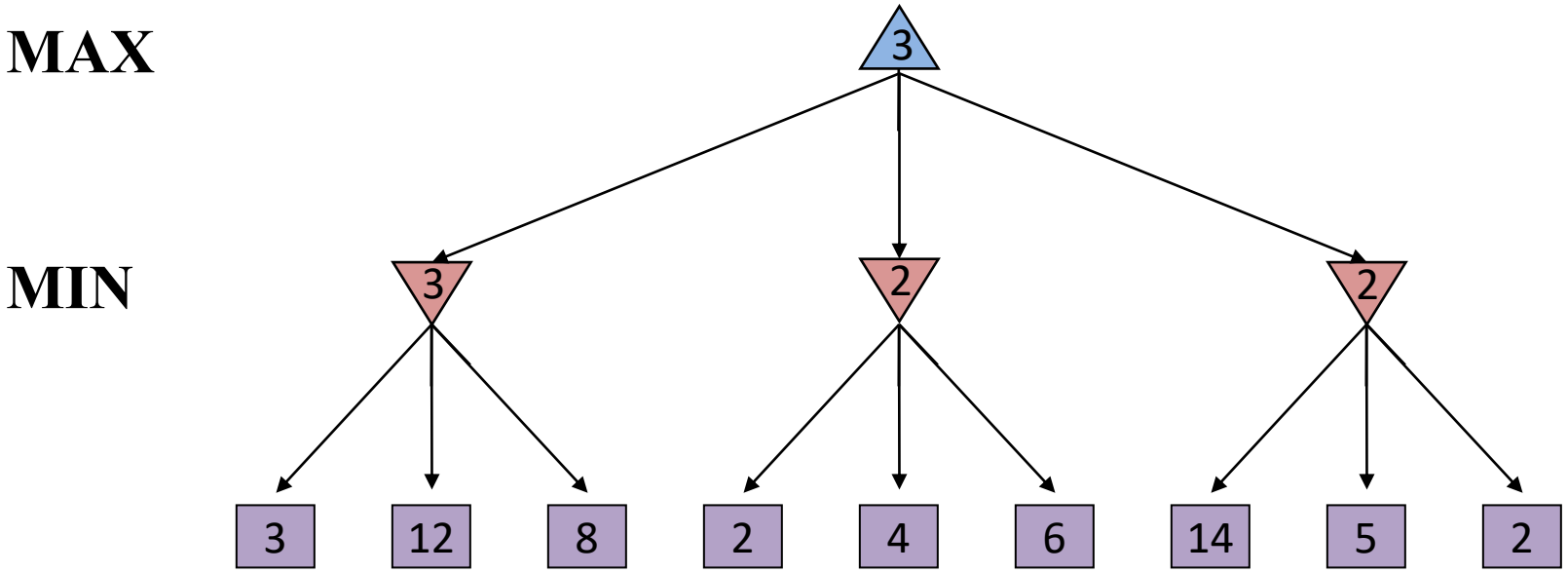
Exemplul MiniMax



Exemplul MiniMax



Exemplul MiniMax

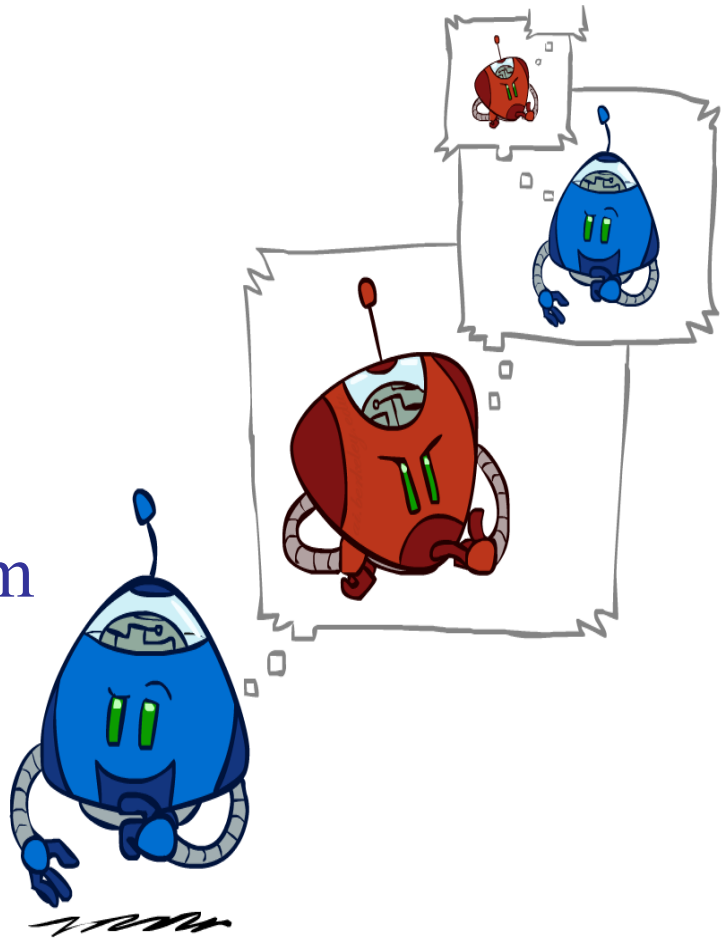


- valorile pozițiilor de la ultimul nivel sunt determinate de către funcția de utilitate și se numesc valori statice (se realizează evaluări statice).
- valorile minimax ale nodurilor interne sunt calculate în mod dinamic, în manieră bottom-up, nivel cu nivel, până când este atins nodul-rădăcină.
- valoarea rezultată este 3 și prin urmare cea mai bună mutare a lui MAX din poziția curentă este mutarea la stânga. Cel mai bun răspuns al lui MIN este mutarea la stânga. Această secvență a jocului poartă denumirea de variație principală. Ea definește jocul optim de tip minimax pentru ambele părți.
- se observă ca valoarea pozițiilor de-a lungul variației principale nu variază. Prin urmare, ***mutările corecte sunt cele care conservă valoarea jocului.***

Proprietăți ale algoritmului MiniMax

- Cât de eficient este MiniMax?
  - Căutare exhaustivă în manieră depth-first
  - timp: $O(b^m)$
  - spațiu: O(bm)

- Exemple: pentru șah, $b \approx 35$, $m \approx 100$
  - o soluție exactă este imposibilă
  - dar, oare chiar trebui să explorăm întreg arborele?



Limitări date de resurse ale algoritmului MiniMax

1. **Generează întreg arborele de joc până la stările terminale**

2. Aplică funcția de utilitate fiecărei stării terminale - obține valoarea 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 fii săi;

   - dacă starea-părinte este un nod de tip MIN, atribuie-i minimul dintre valorile avute de fii săi;

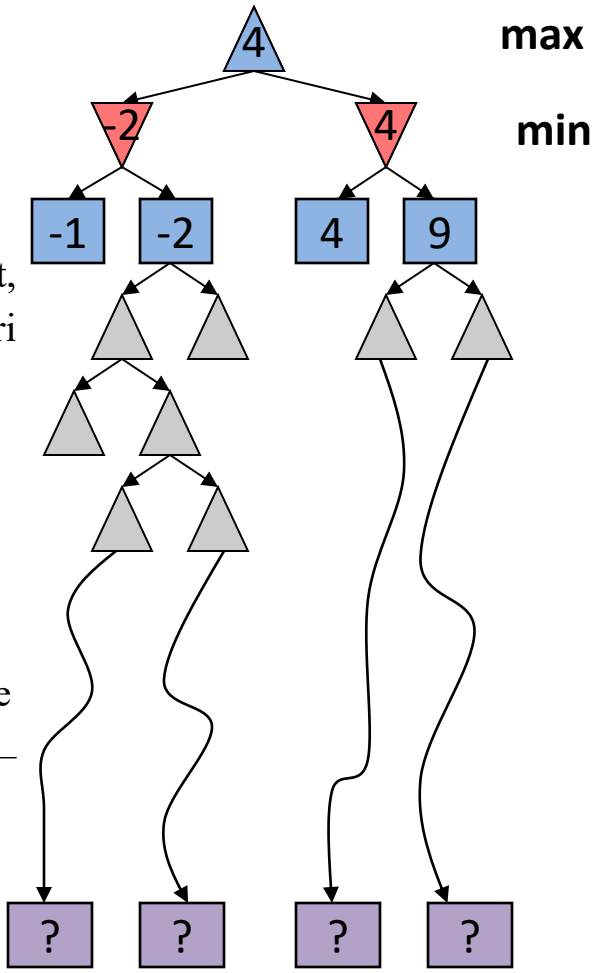
4. Ajuns în nodul-rădăcină, alege pentru MAX acea mutare care conduce la valoarea maximă. Mutarea se numește ***decizia minimax*** - maximizează utilitatea, în ipoteza că oponentul joacă perfect cu scopul de a o minimiza.

Limitări date de resurse

- problemă: în jocuri reale, nu putem genera tot arborele până la frunze din cauza resurselor limitate (timp + spațiu)

- soluție posibilă: căutare în adâncime limitată
  - caută până la o anumită adâncime în arborele de joc
  - înlocuiește utilitățile stărilor terminale (care sunt calculate exact, după regulile jocului) cu o funcție de evaluare a utilității unor stări neterminale (această funcție aproximează utilitatea)

- exemplu:
  - avem la dispoziție 100 de secunde petru a realiza mutarea;
  - putem explora 10000 noduri / secundă;
  - în 100 de secunde putem explora 1 Milion de noduri pentru mutare
  - algoritmul α-β retezare (îl prezentăm azi) ajunge la adâncime 8 – performanță bună în șah;



- garanția de optimalitate dispare;
- cu cât pot analiza mai multe noduri cu atât mutarea este mai bună;
- uneori pot folosi și un algoritm de căutare incrementală în adâncime

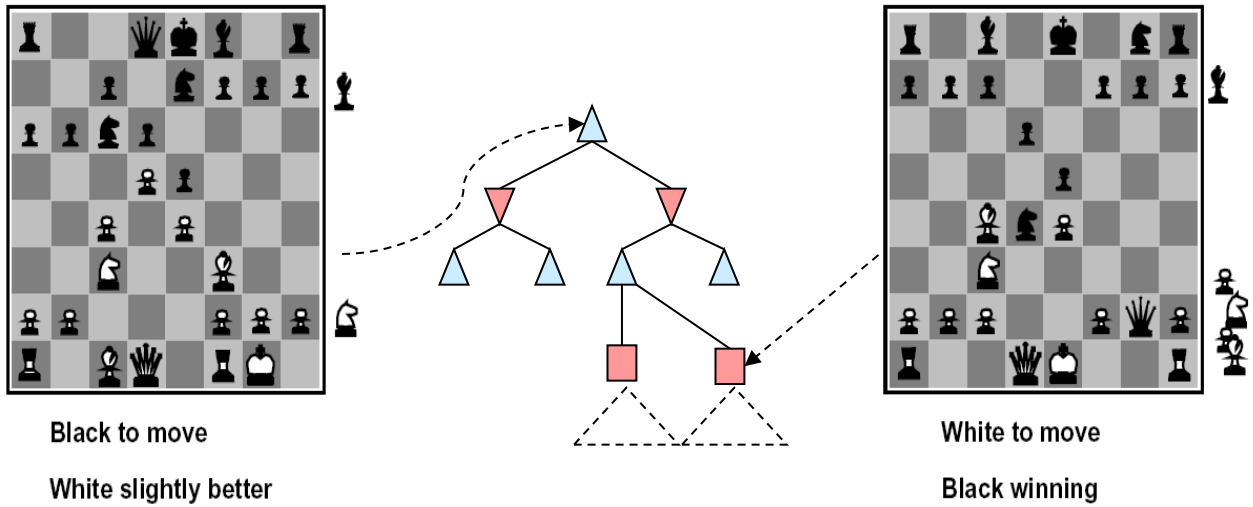
Influența adâncimii pentru funcția de evaluare

- Funcțiile de evaluare sunt imperfecte

- Cu cât merg mai adâncime în arbore și apelez funcția de evaluare a unei stări neterminale mai târziu cu atât am șanse să greșesc mai puțin.

Funcții de evaluare - șah

- Funcțiile de evaluare asociază un scor stărilor neterminale (este folosită de căutarea în adâncime limitată, căutarea în adâncime iterativă)



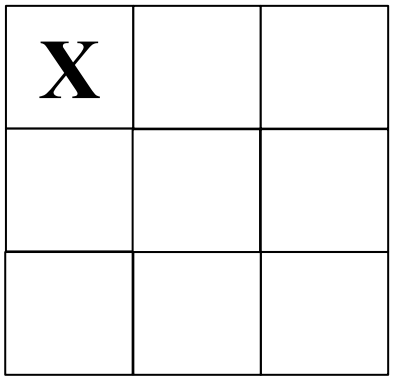
- o funcție de evaluare ideală returnează valoarea MiniMax a stării.
- diverse funcții de evaluare specifice pentu fiecare joc: pentru șah se consideră funcții liniare în care ponderăm piesele de pe tablă

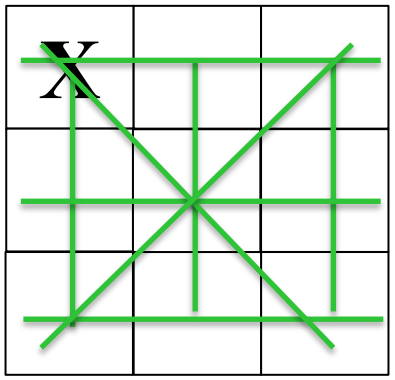
$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

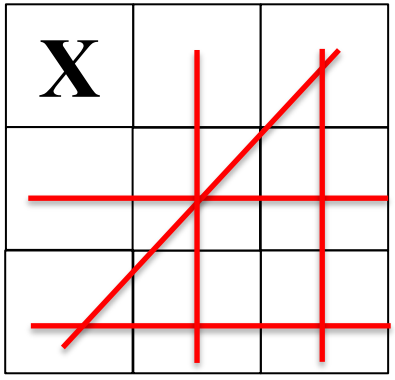
- unde, $f_1(s)$ = (#regine_albe – #regine_negre), $w_1 = 100$, etc.

Funcții de evaluare – X și 0

- o funcție de evaluare ideală returnează valoarea MiniMax a stării.
- diverse funcții de evaluare specifice pentru fiecare joc:



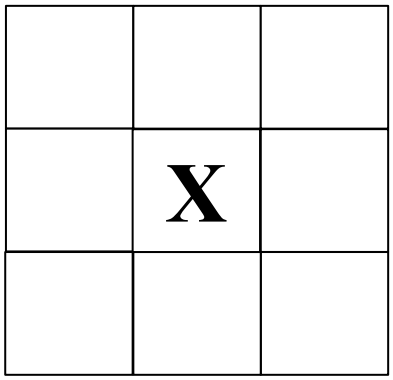


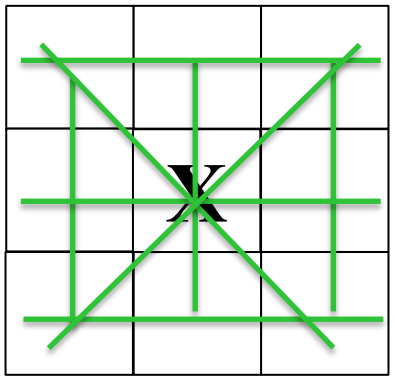


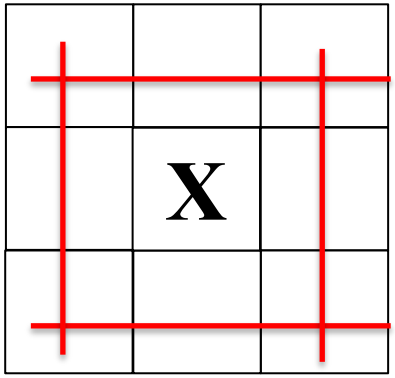
- exemplu de funcție de evaluare pentru X și 0: diferența dintre numărul de soluții disponibile pentru MAX și numărul de soluții disponibile pentru MIN
- în exemplul de mai sus: 8-5 = 3

Funcții de evaluare – X și 0

- o funcție de evaluare ideală returnează valoarea MiniMax a stării.
- diverse funcții de evaluare specifice pentru fiecare joc:



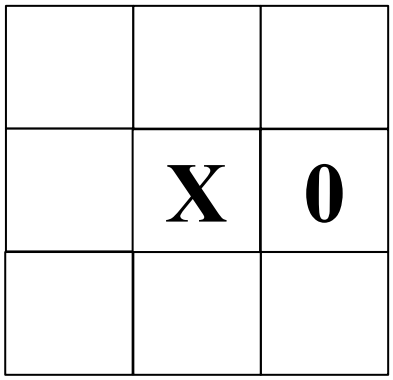


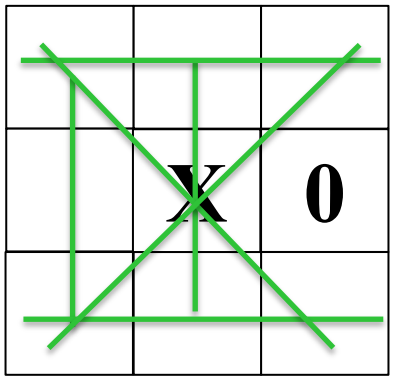


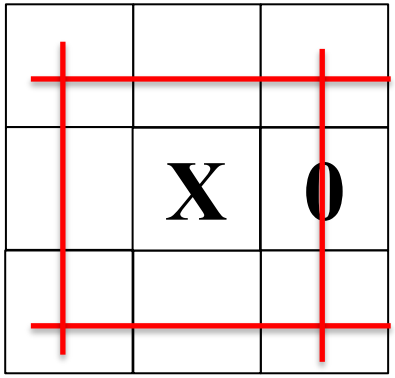
- exemplu de funcție de evaluare pentru X și 0: diferența dintre numărul de soluții disponibile pentru MAX și numărul de soluții disponibile pentru MIN
- În exemplul de mai sus: 8-4 = 4

Funcții de evaluare – X și 0

- o funcție de evaluare ideală returnează valoarea MiniMax a stării.
- diverse funcții de evaluare specifice pentru fiecare joc:



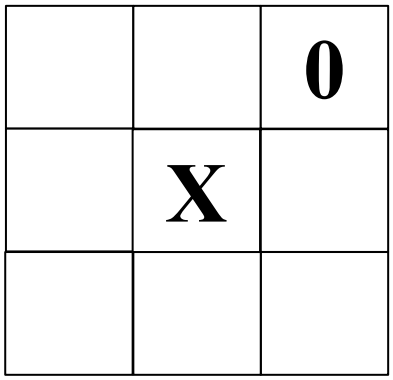


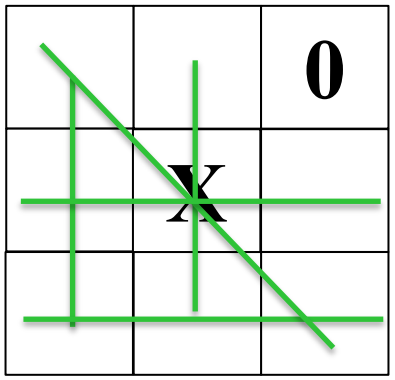


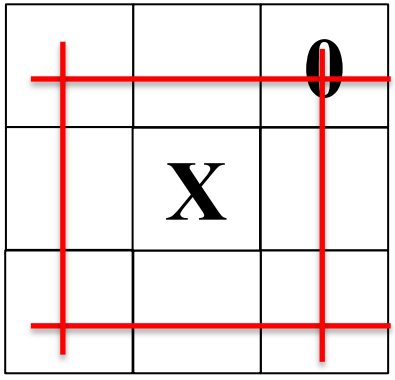
- exemplu de funcție de evaluare pentru X și 0: diferența dintre numărul de soluții disponibile pentru MAX și numărul de soluții disponibile pentru MIN
- În exemplul de mai sus: 6-4 = 2

Funcții de evaluare – X și 0

- o funcție de evaluare ideală returnează valoarea MiniMax a stării.
- diverse funcții de evaluare specifice pentru fiecare joc:

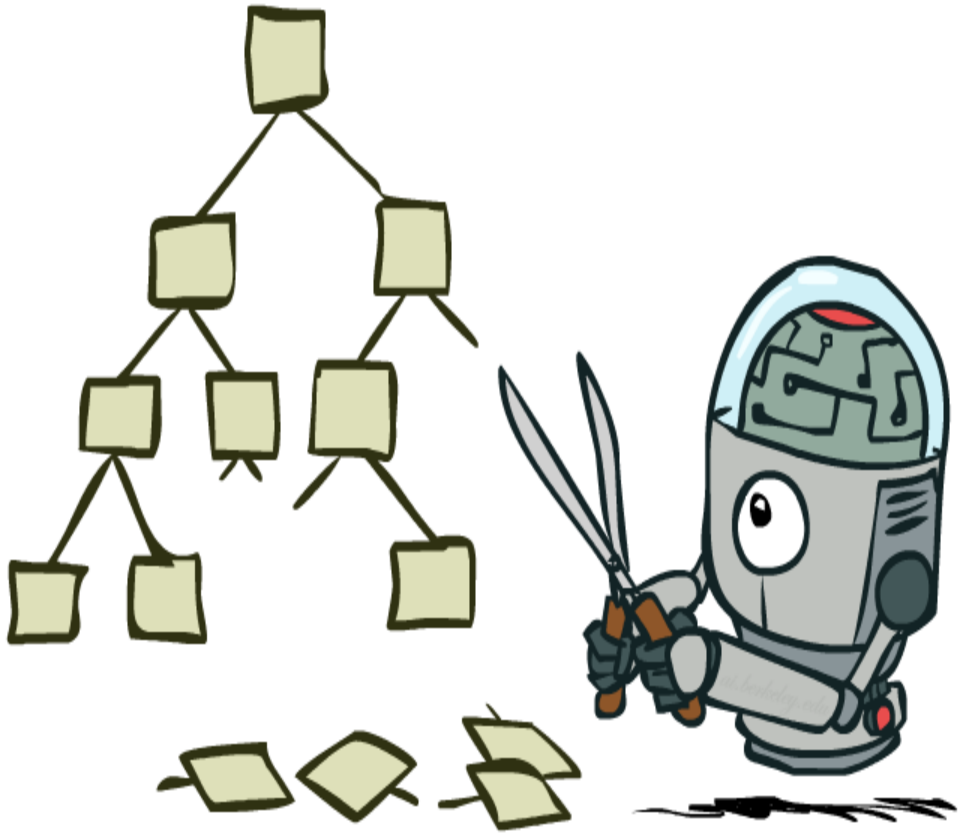




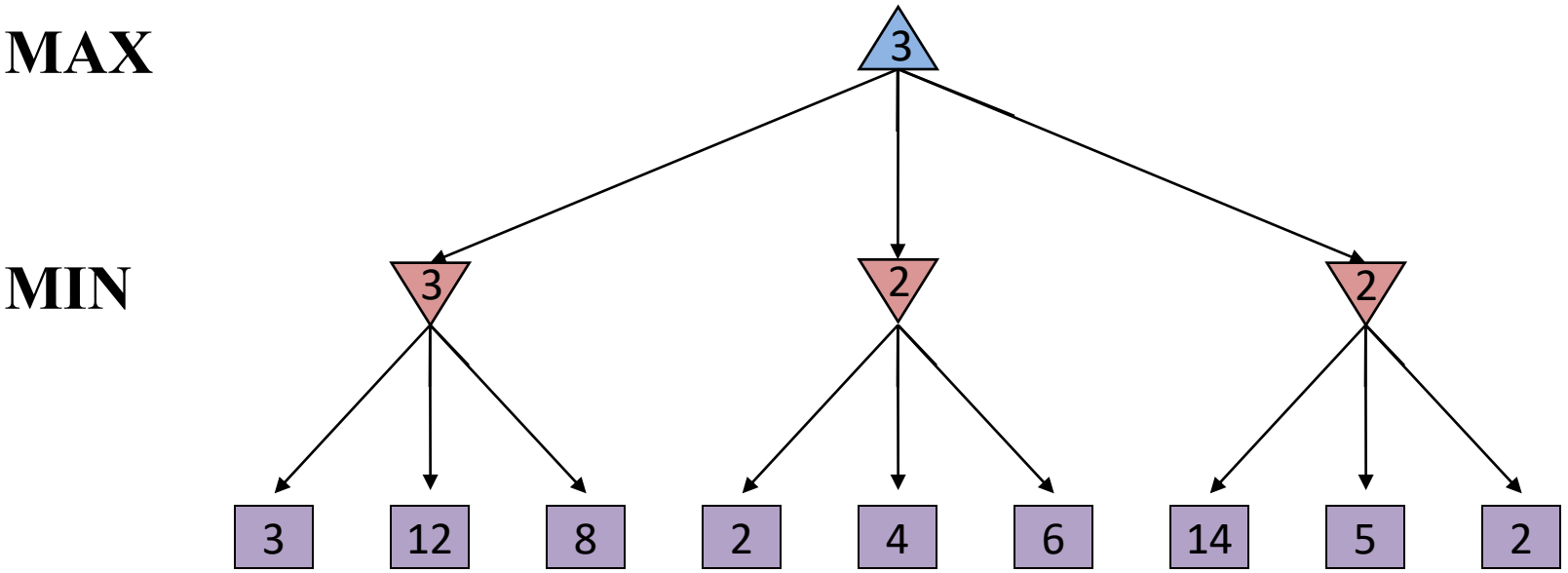


- exemplu de funcție de evaluare pentru X și 0: diferența dintre numărul de soluții disponibile pentru MAX și numărul de soluții disponibile pentru MIN
- În exemplul de mai sus: 5-4 = 1

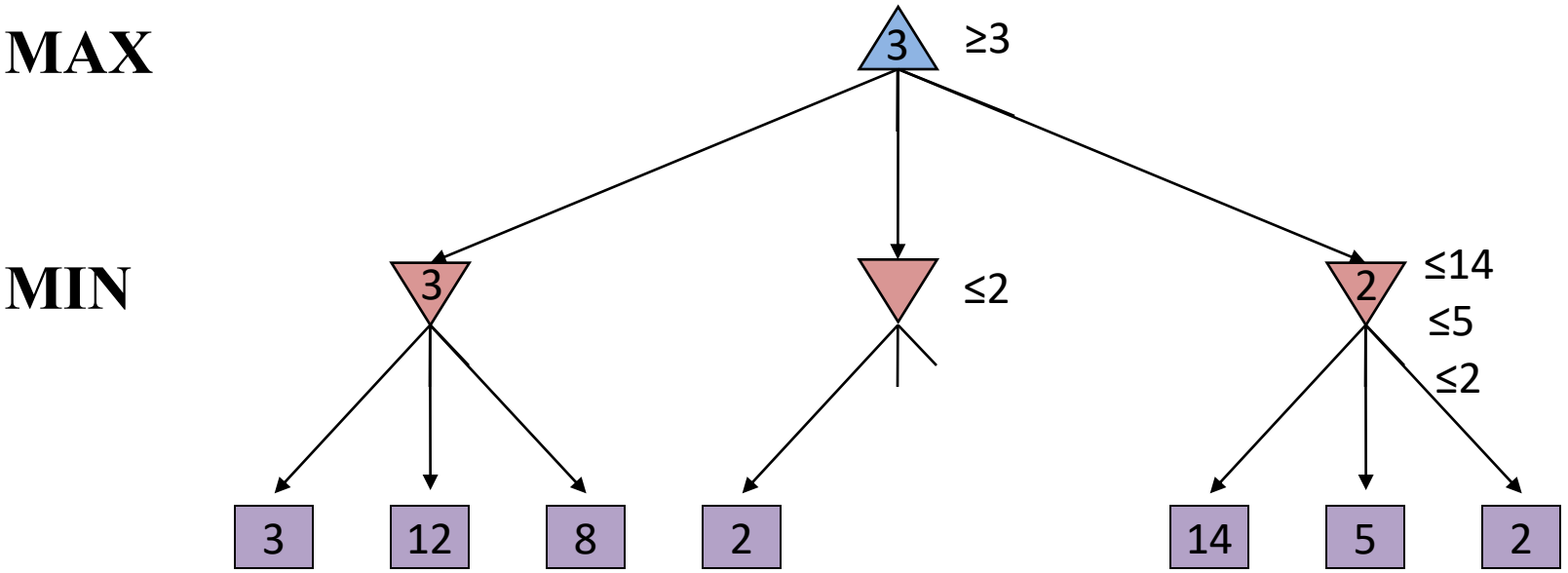
Retezări (Pruning) în arborele de joc



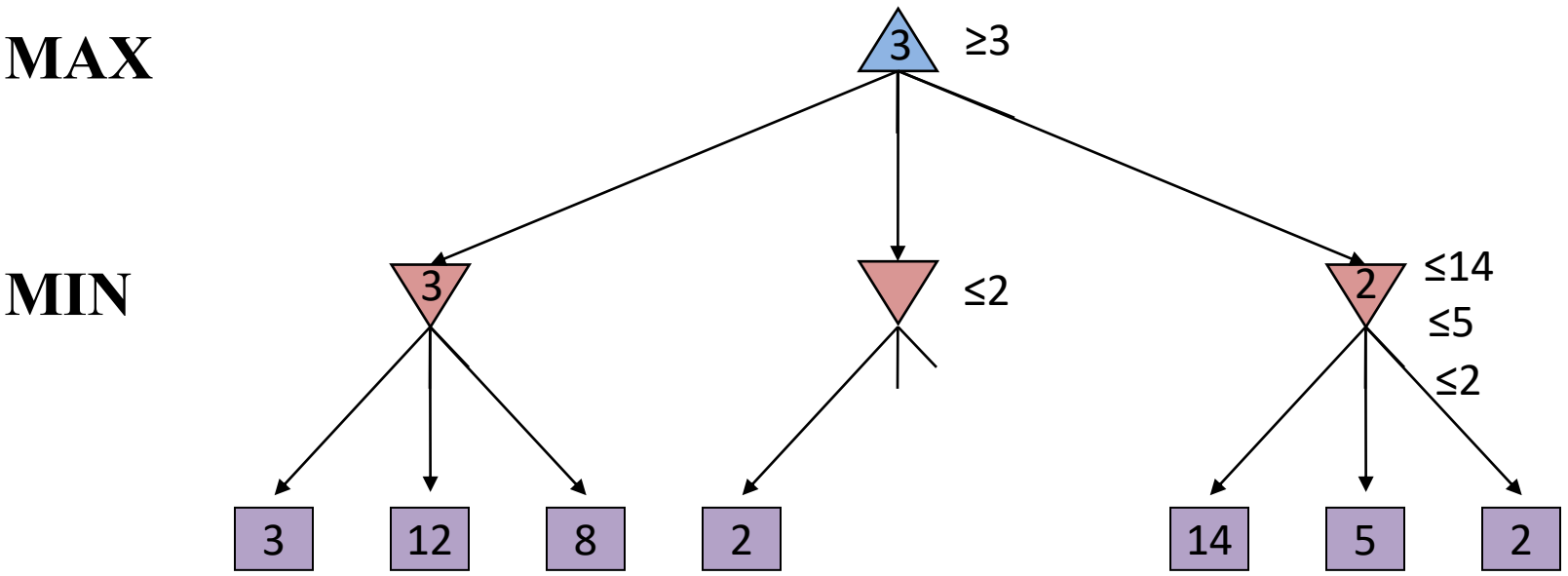
Exemplu MiniMax



Accelerarea algoritmului MiniMax



Retezare alfa-beta

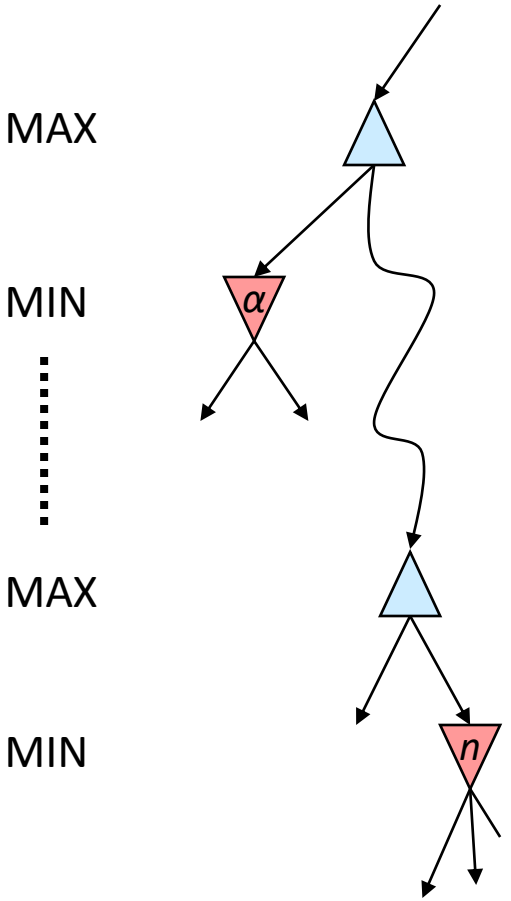


Tehnica de alfa-beta retezare, când este aplicată unui arbore de tip Minimax standard, va întoarce aceeași mutare pe care ar furniza-o și Algoritmul MiniMax, dar într-un timp mai scurt, întrucât realizează o retezare a unor ramuri (subarbori) ale arborelui care nu pot influența decizia finală și care nu mai sunt vizitate.

Retezare alfa-beta (alfa-beta pruning)

- Cazurile când se aplică (pentru noduri de tip MIN)
  - calculăm pentru jucătorul MIN valoarea nodului *n*
  - iterăm după toți succesorii nodului *n* de la stânga la dreapta
  - valoarea estimată a nodului *n* va scădea pe măsură ce îi vizităm toți succesorii
  - cine este interesat de valoarea nodului *n*? MAX
  - fie α cea mai bună valoare curentă (garantată) pe care MAX o poate obține după ce a vizitat subarborii din stânga subarborelui actual vizitat pornind de la rădăcină
  - dacă valoarea curentă estimată a lui *n* devine mai mică decât α, MAX nu va alege acest drum, deci putem să renunțăm (să retezăm) să evaluăm valorile pentru celelalte noduri succesor ale lui *n*.

- Cazurile când se aplică (pentru noduri de tip MAX)
  - simetric, se definește în mod similar valoarea β ca fiind cea mai bună valoare (garantată) pe care MIN o poate obține.
  - MIN nu mai trebuie să mai ia în considerare nicio valoare internă mai mare sau egală cu β care este asociată oricărui nod intern *n* de tip MAX. Putem tăia (reteza) acea ramură a arborelui de joc.



Implementare Alfa-Beta retezare

α: opțiunea curentă cea mai bună a lui MAX (MAX vrea să maximizeze α)
β: opțiunea curentă cea mai bună a lui MIN (MIN vrea să minimizeze β)
Întotdeauna vom avea $\alpha \leq \beta$

```
def max-value(stare, α, β):   //nodul n e de tip MAX, MAX are α, MIN are β
    initializeaza v = -∞
    pentru fiecare succesor al stării:          //pentru fiecare succesor al lui n
        v = max(v, value(successor, α, β))   //calculăm valoarea v și actualizăm max
        if v ≥ β return v    //dacă v depășește limita β a lui MIN nu mai continuăm
                             // întoarcem în acest caz v  = o estimare a lui n
        α = max(α, v)        // actualizăm valoarea α
    return v
```

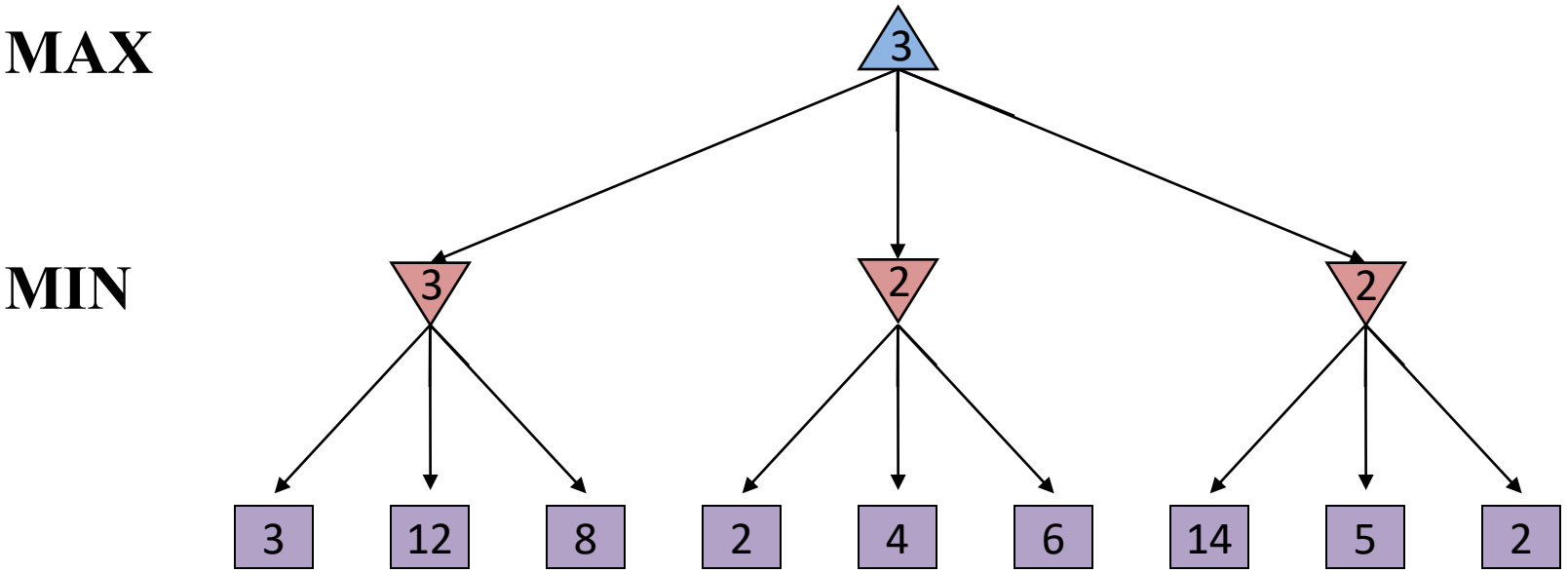
Implementare Alfa-Beta retezare

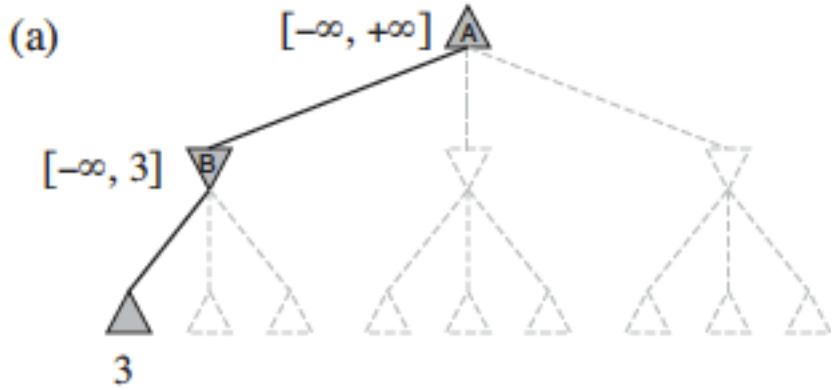
α: opțiunea curentă cea mai bună a lui MAX (MAX vrea să maximizeze α)
β: opțiunea curentă cea mai bună a lui MIN (MIN vrea să minimizeze β)
Întotdeauna vom avea $\alpha \leq \beta$

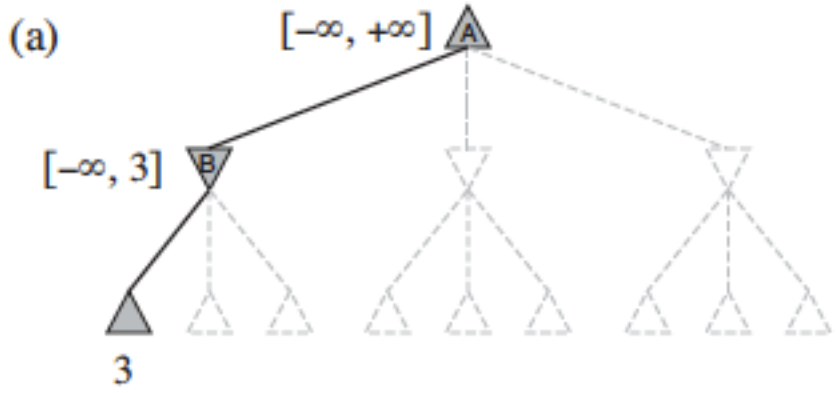
```
def max-value(stare, α, β):
    initializeaza v = -∞
    pentru fiecare succesor al stării:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

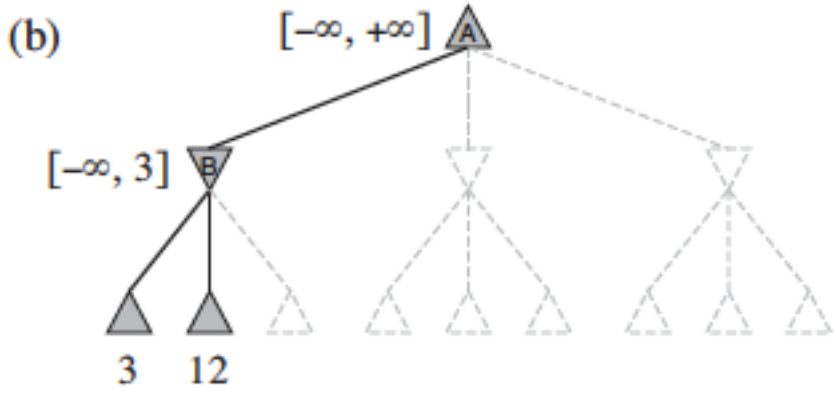
```
def min-value(stare , α, β):
    initializeaza v = +∞
    pentru fiecare succesor al stării:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

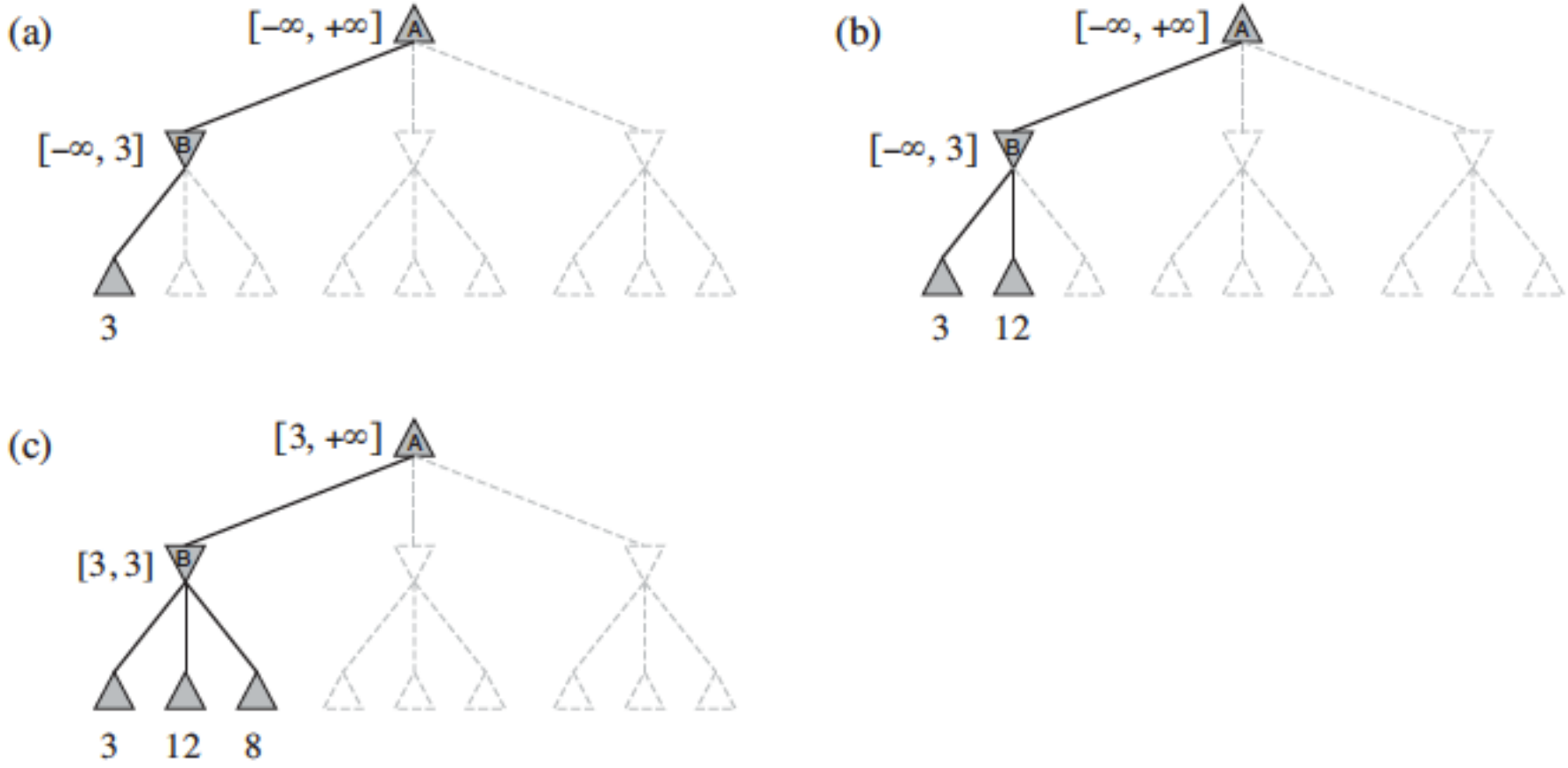
Exemplu MiniMax

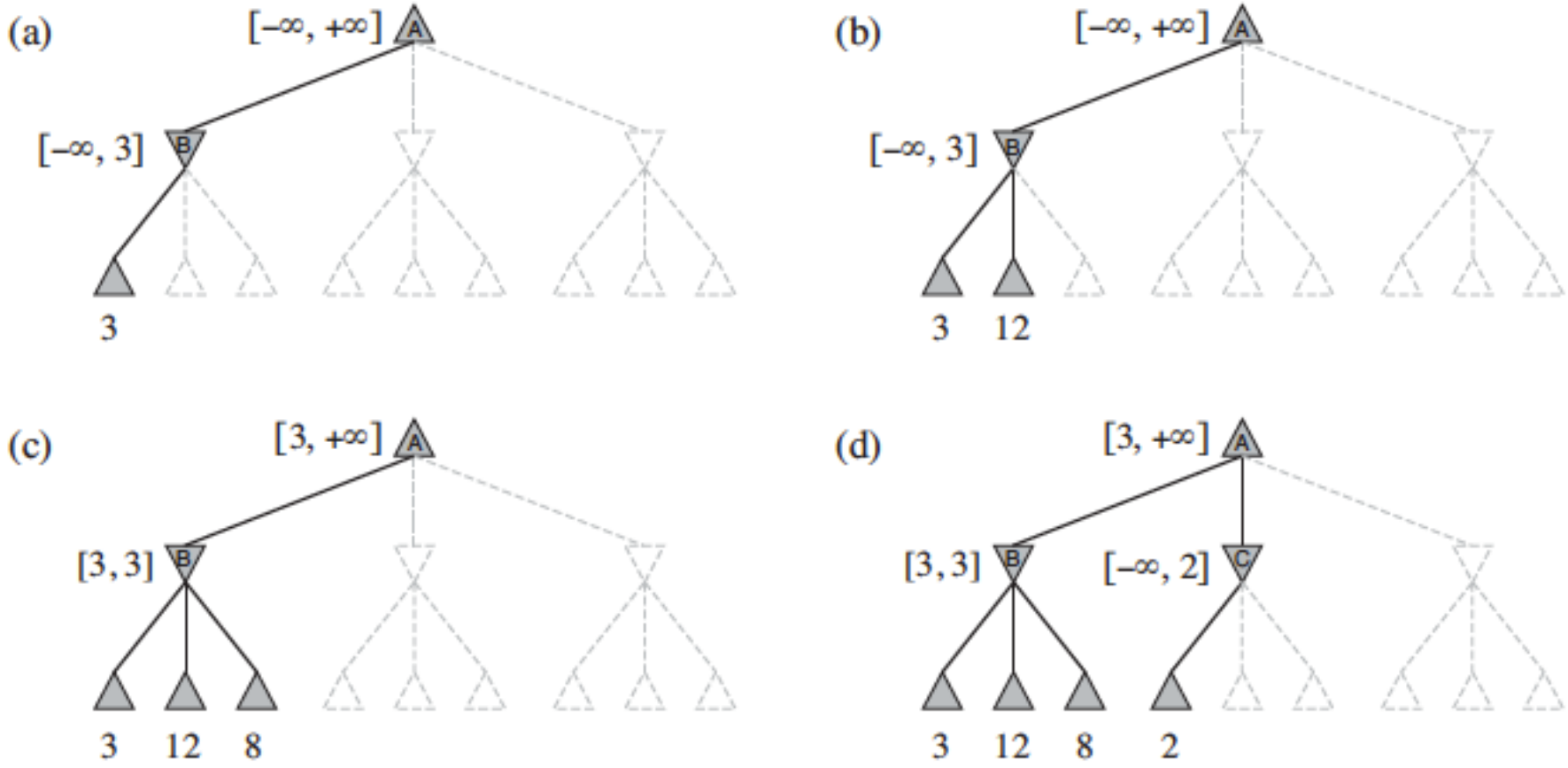


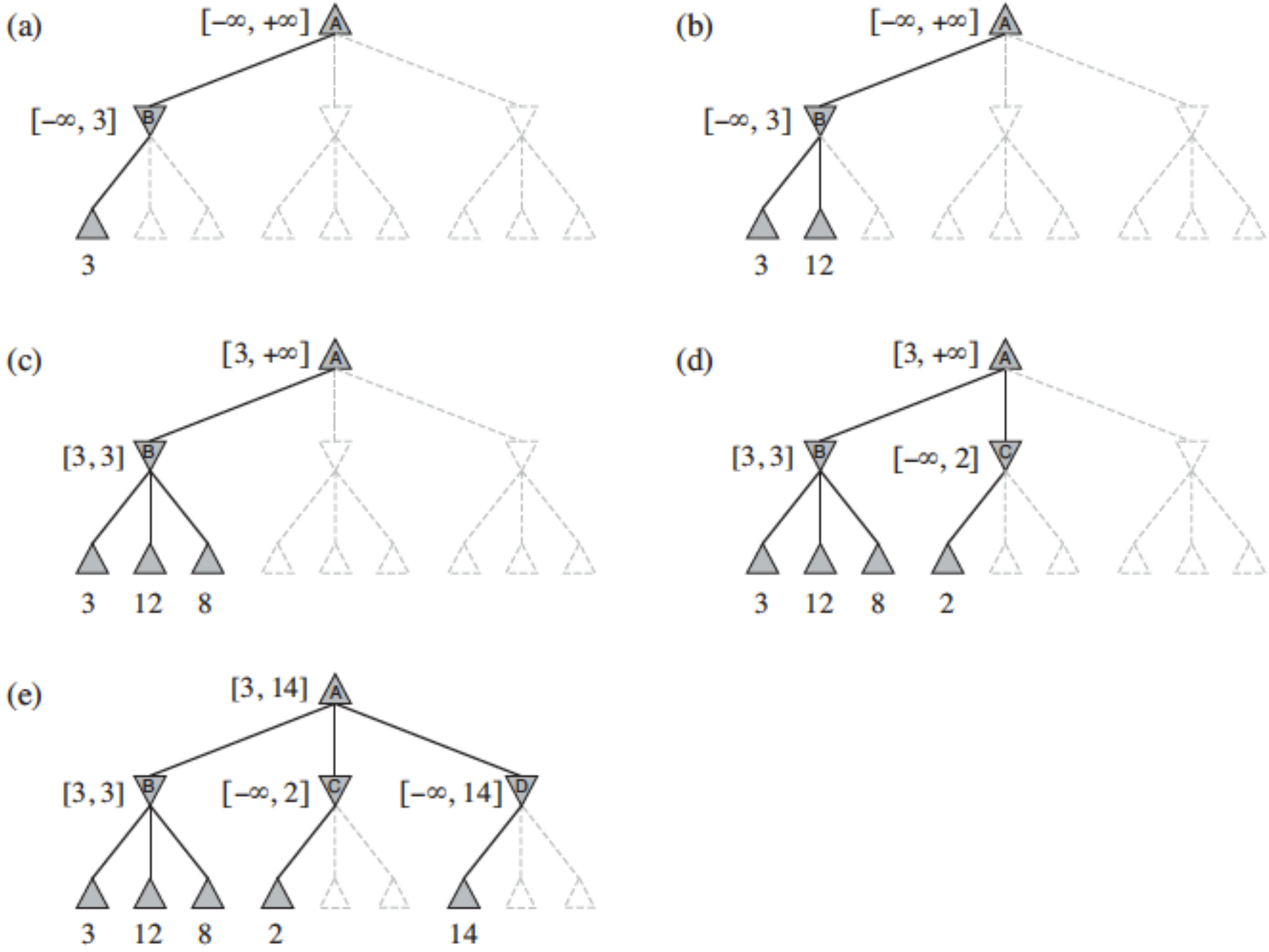

(a)
[−∞, +∞]
A
[−∞, 3]
B
3

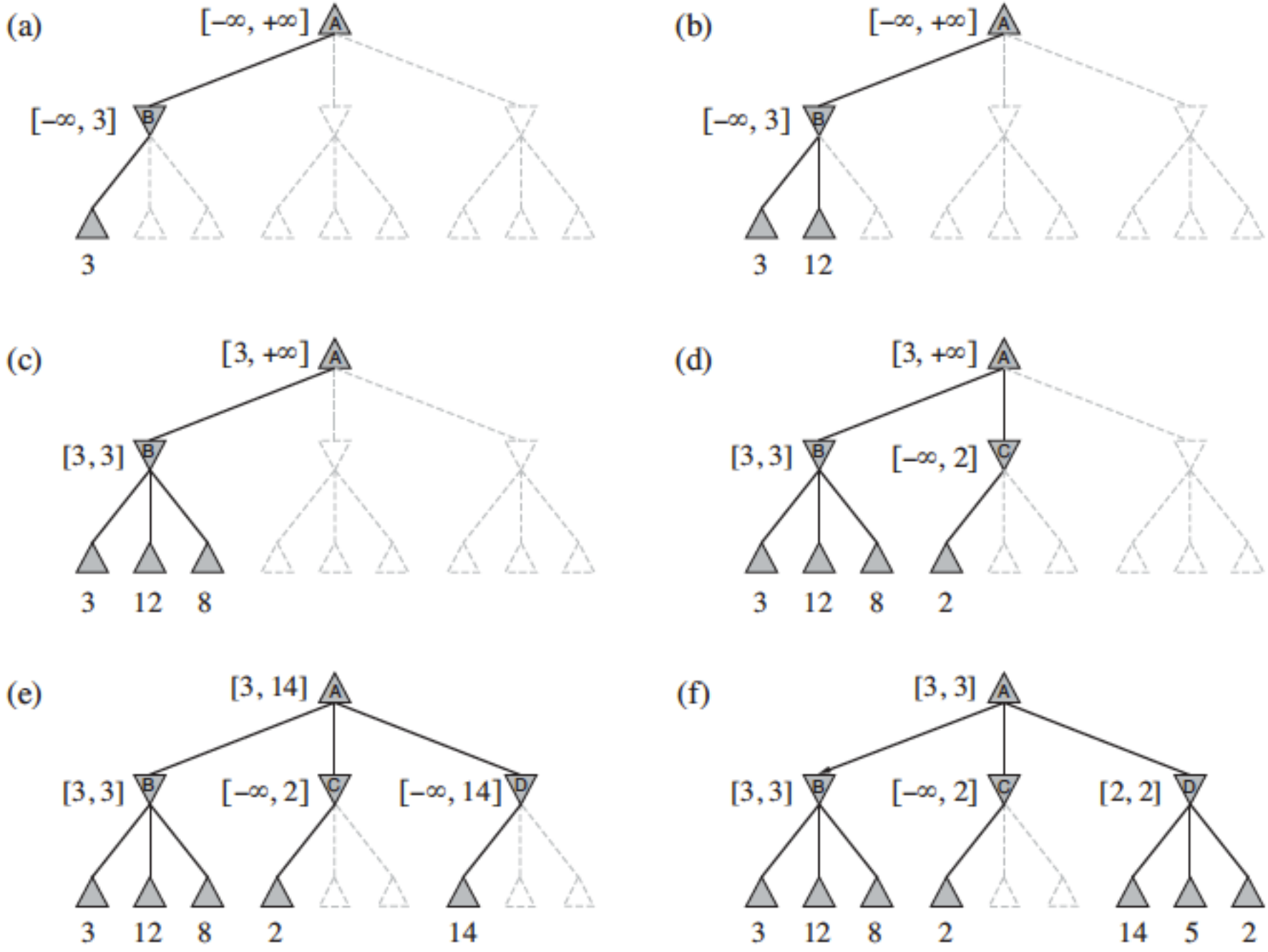

(a)
[−∞, +∞]
A
[−∞, 3]
B
3


(b)
[−∞, +∞]
A
[−∞, 3]
B
3
12

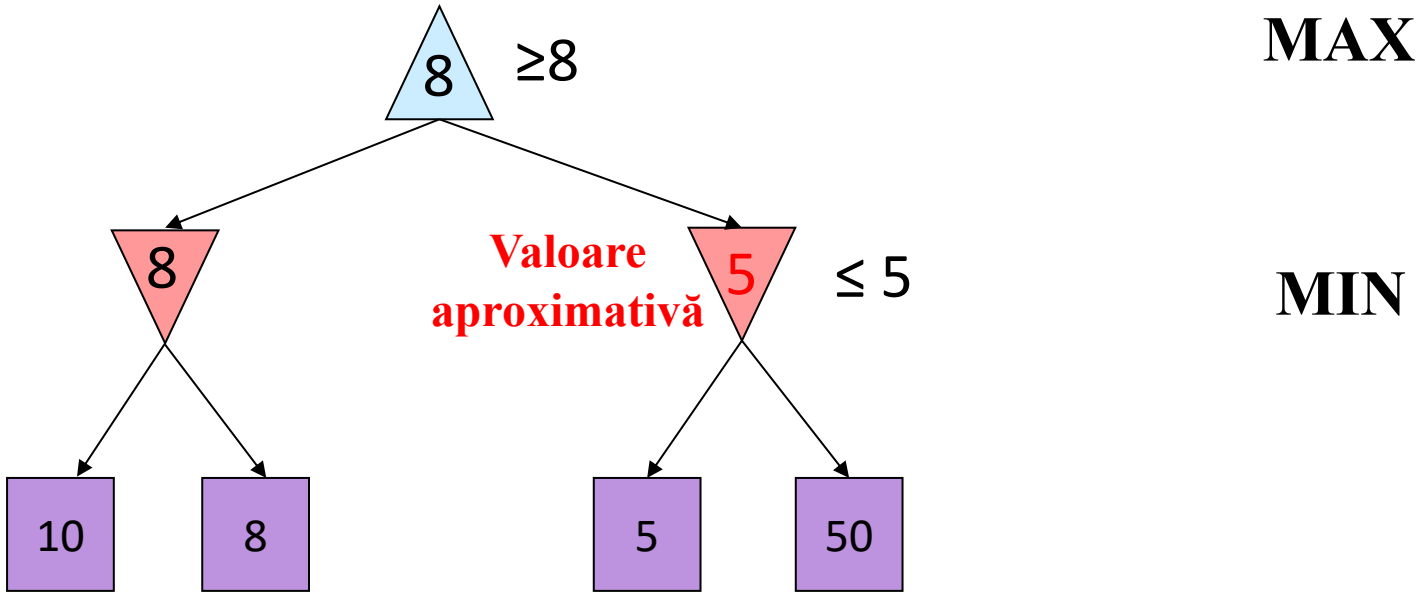

(a)
[−∞, +∞]
A
[−∞, 3]
B
3
(b)
[−∞, +∞]
A
[−∞, 3]
B
3
12
(c)
[3, +∞]
A
[3, 3]
B
3
12
8


(a)
[−∞, +∞]
A
[−∞, 3]
B
3
(b)
[−∞, +∞]
A
[−∞, 3]
B
3
12
(c)
[3, +∞]
A
[3, 3]
B
3
12
8
(d)
[3, +∞]
A
[3, 3]
B
[−∞, 2]
C
3
12
8
2

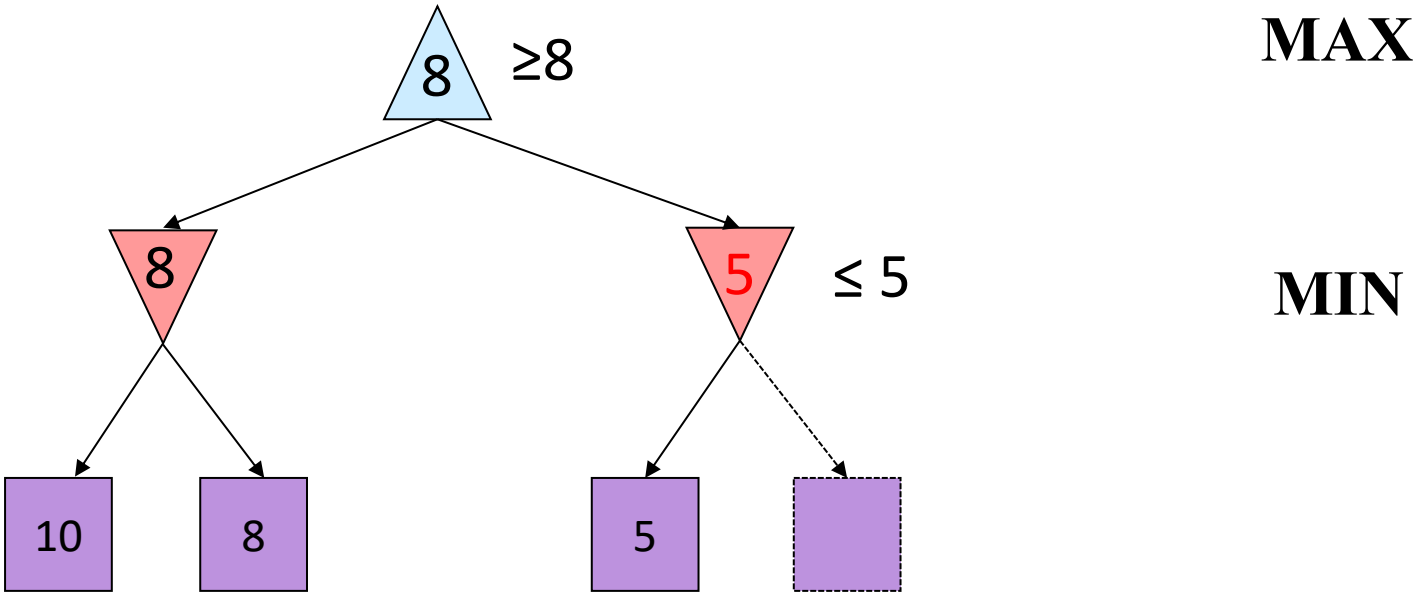

(a)
[−∞, +∞]
A
[−∞, 3]
B
3
(b)
[−∞, +∞]
A
[−∞, 3]
B
3
12
(c)
[3, +∞]
A
[3, 3]
B
3
12
8
(d)
[3, +∞]
A
[3, 3]
B
[−∞, 2]
C
3
12
8
2
(e)
[3, 14]
A
[3, 3]
B
[−∞, 2]
C
[−∞, 14]
D
3
12
8
2
14


(a)
[−∞, +∞]
A
[−∞, 3]
B
3
(b)
[−∞, +∞]
A
[−∞, 3]
B
3
12
(c)
[3, +∞]
A
[3, 3]
B
3
12
8
(d)
[3, +∞]
A
[3, 3]
B
[−∞, 2]
C
3
12
8
2
(e)
[3, 14]
A
[3, 3]
B
[−∞, 2]
C
[−∞, 14]
D
3
12
8
2
14
(f)
[3, 3]
A
[3, 3]
B
[−∞, 2]
C
[2, 2]
D
3
12
8
2
14
5
2

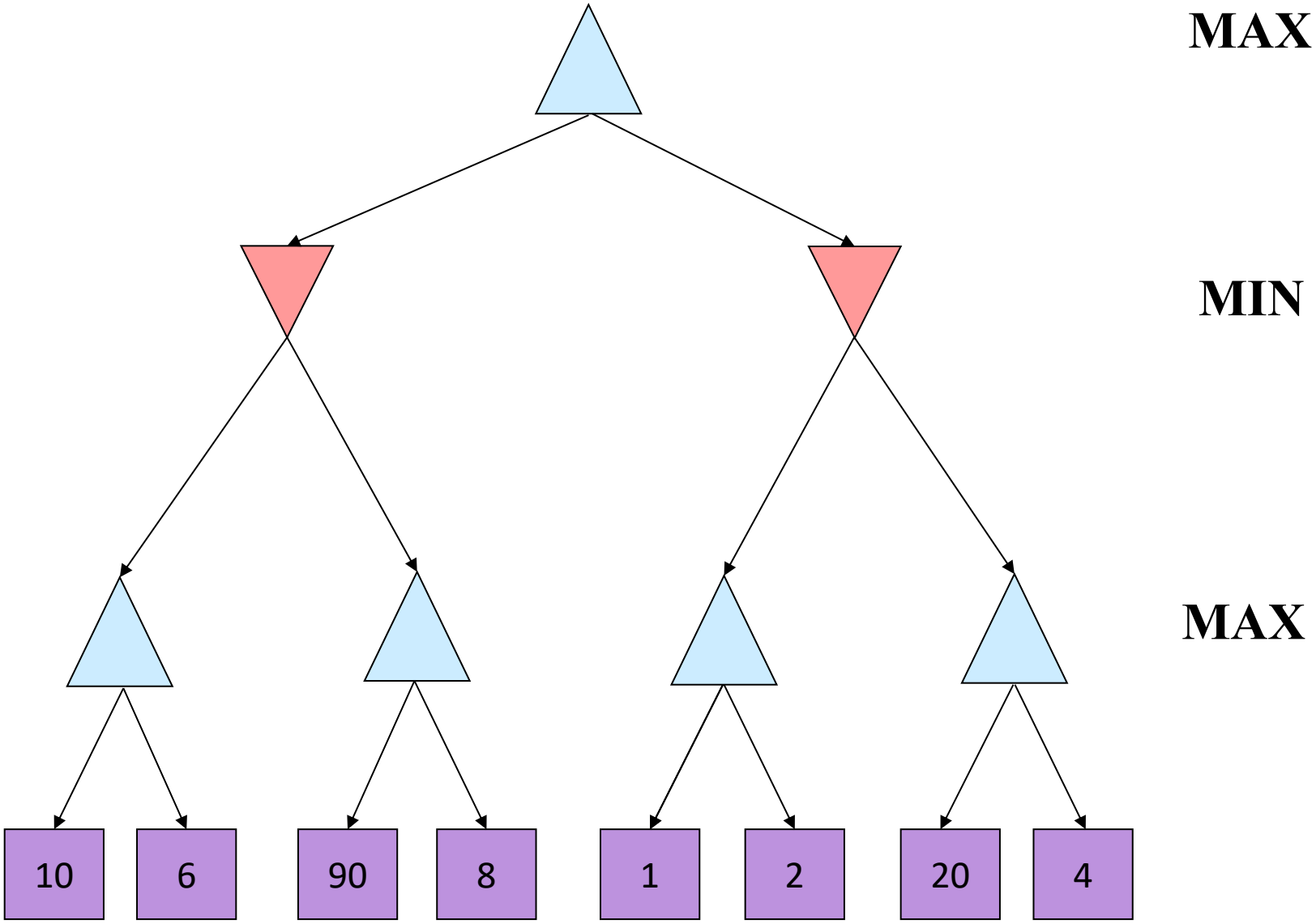
Exemplu



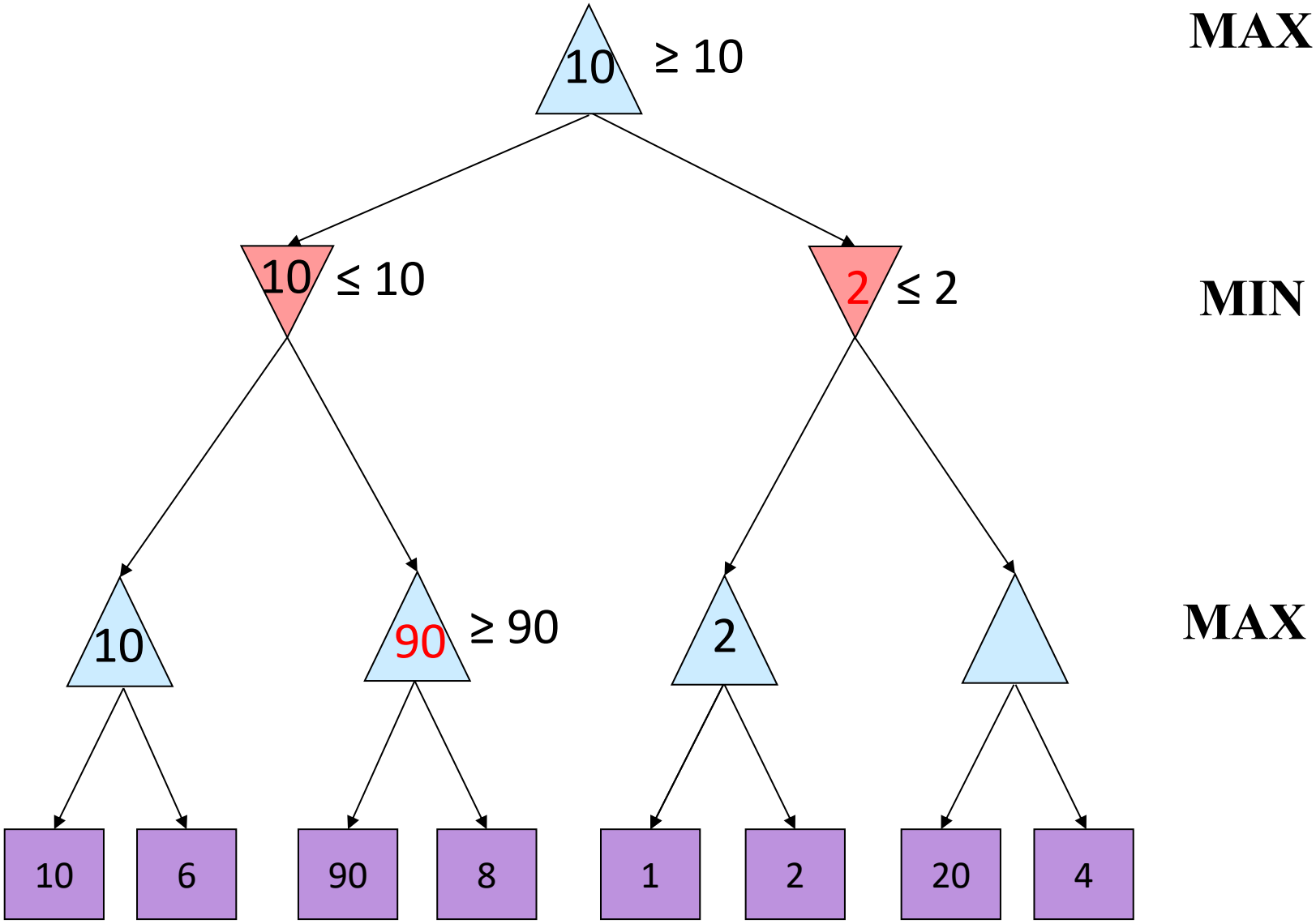
Exemplu



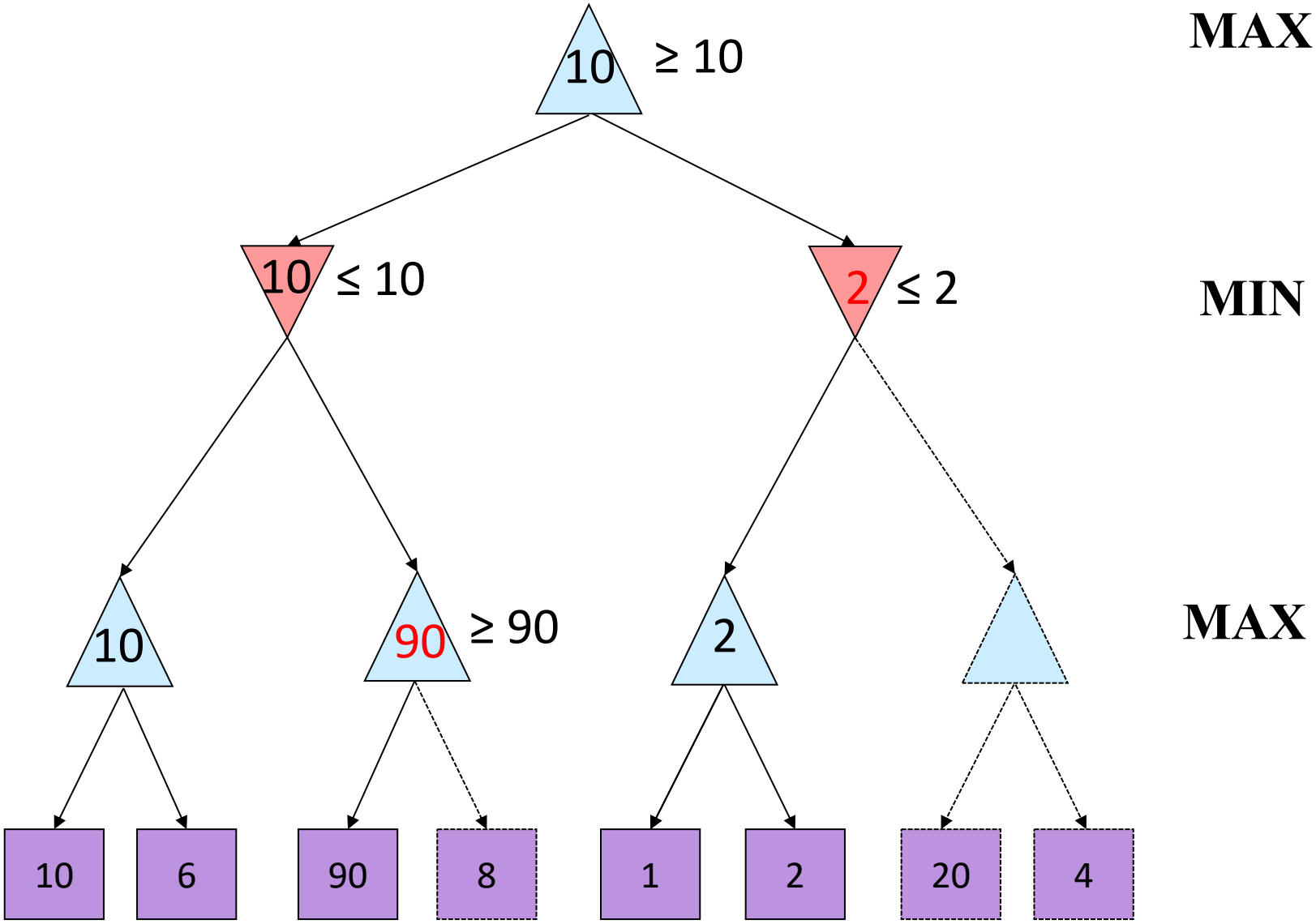
Exemplu



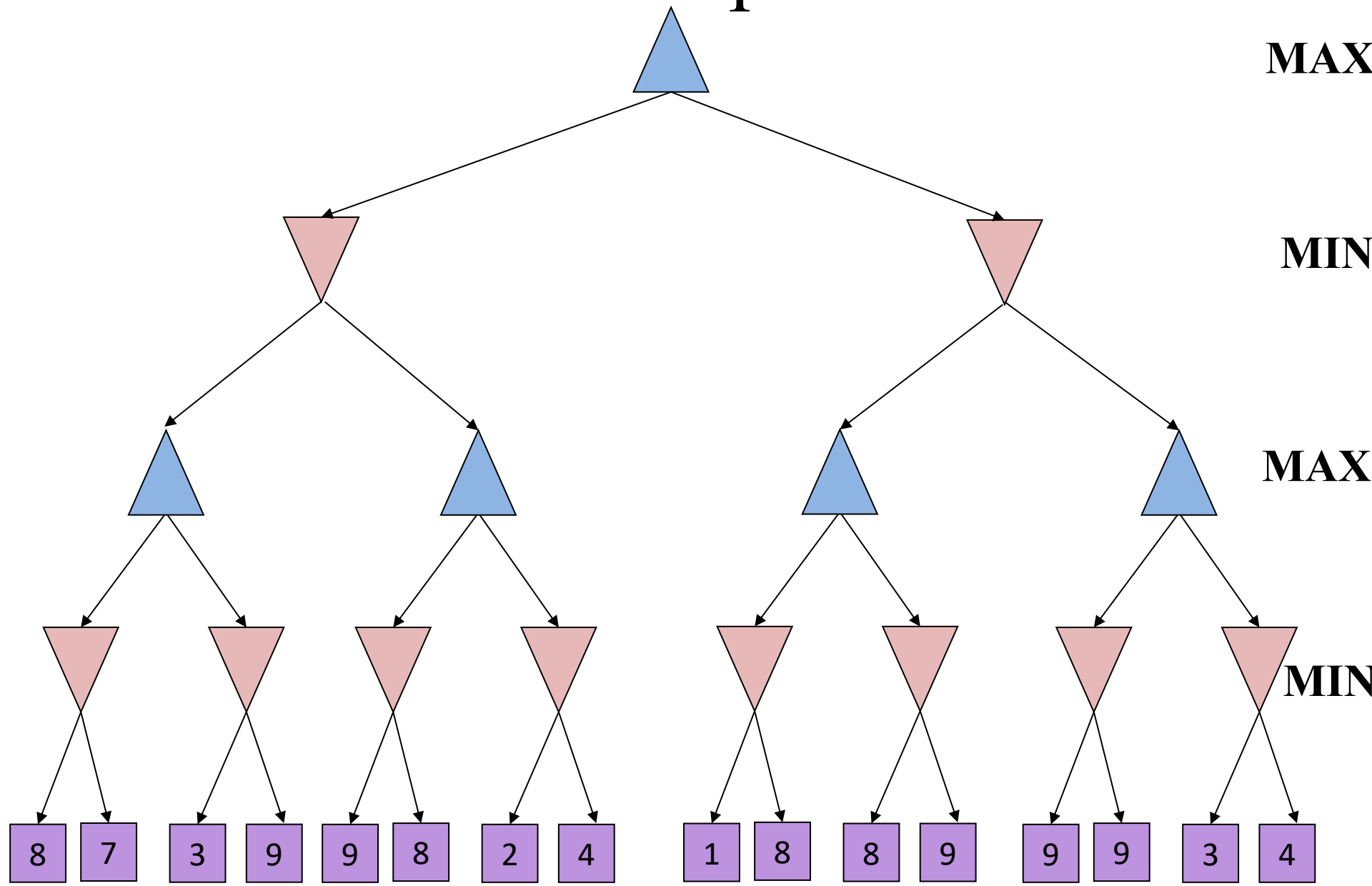
Exemplu



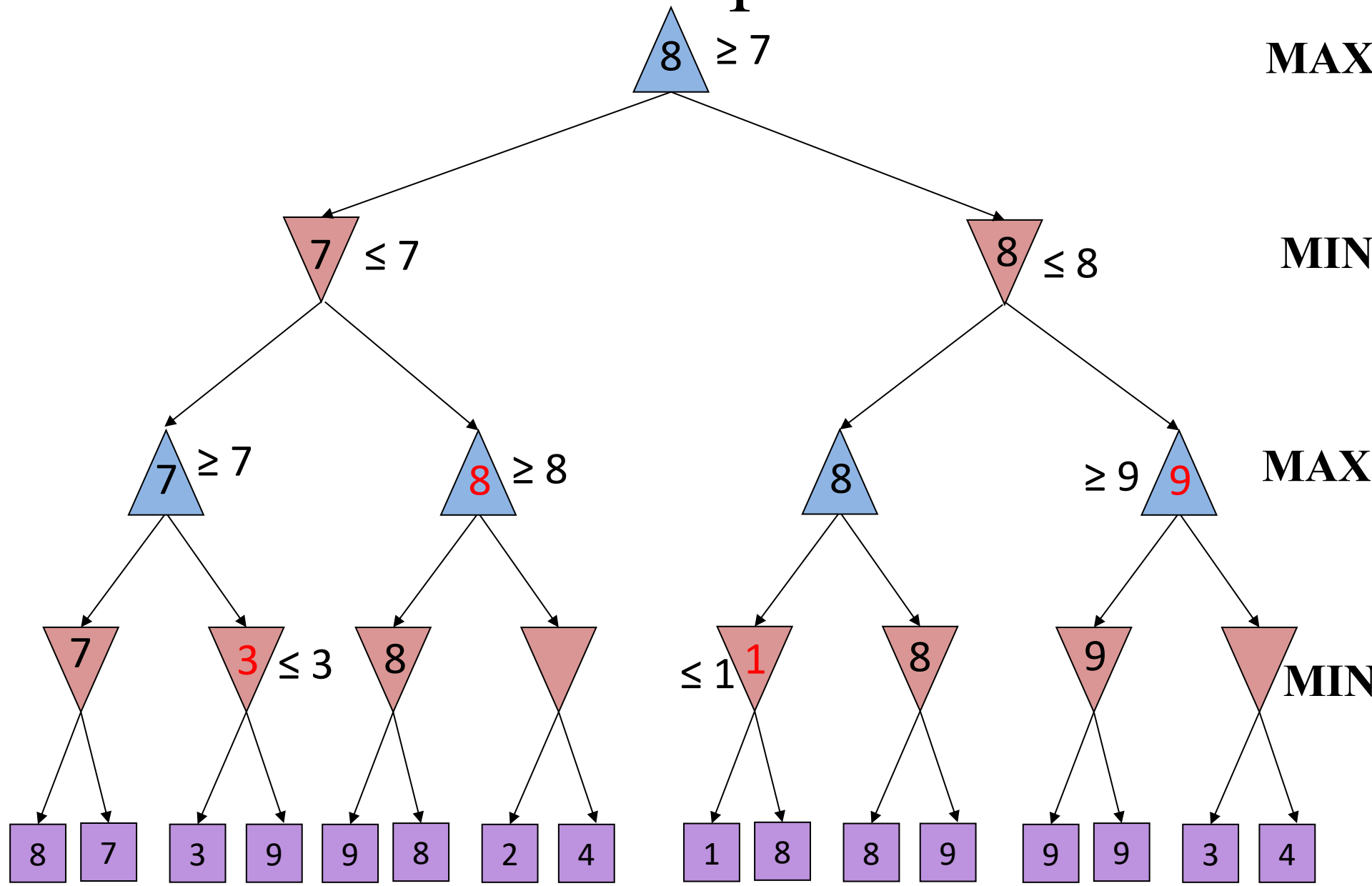
Exemplu



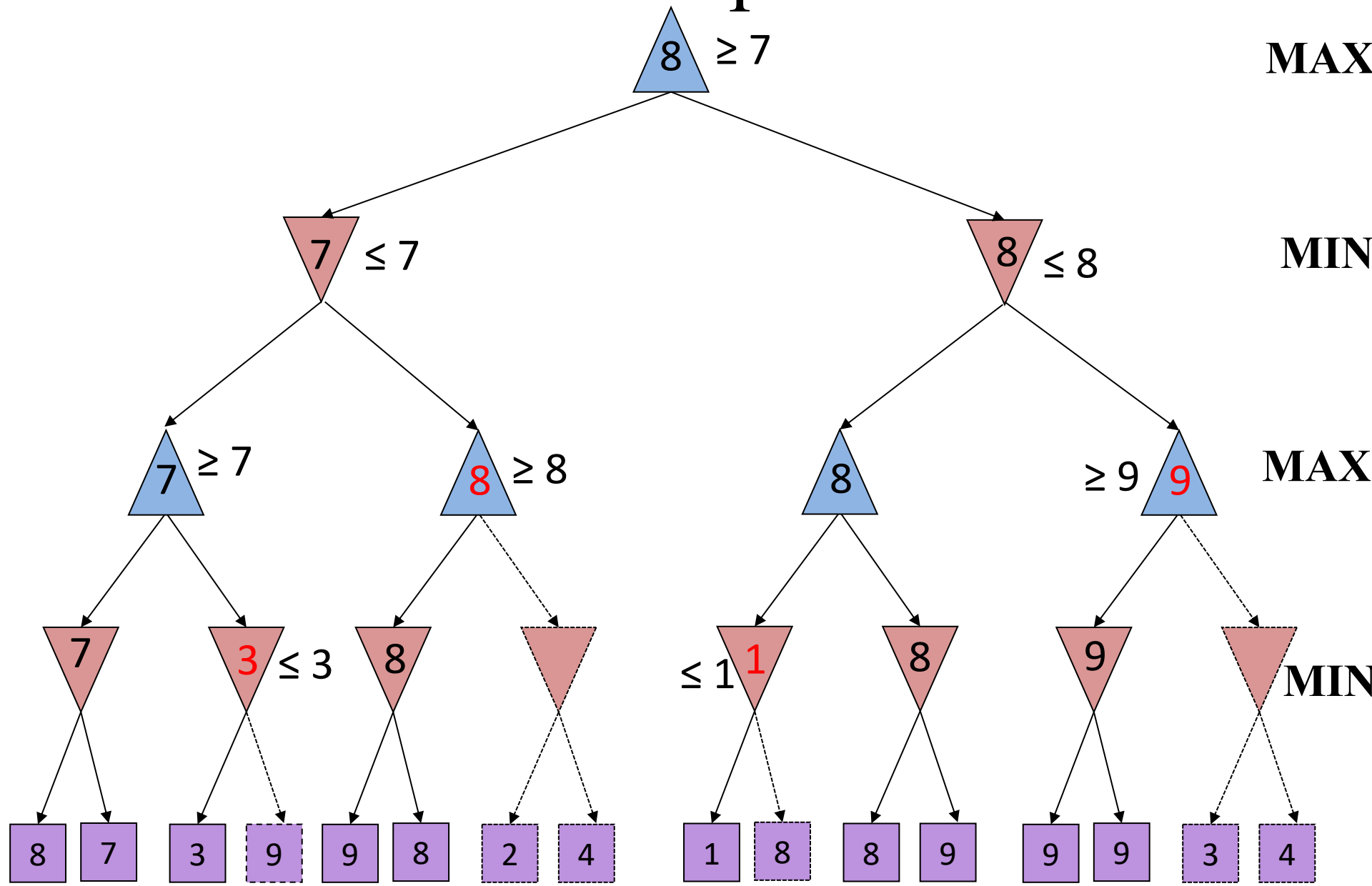
Exemplu



Exemplu



Exemplu



Proprietățile algoritmului Alpha-Beta Pruning

- procesul de retezare nu are niciun efect asupra valorii minimax calculate pentru nodul rădăcină (aceasta nu se schimbă)

- valorile nodurilor intermediare (aproximate) pot fi greșite:
  - important: nodurile fii ale rădăcinii pot avea valori greșite

  Implementarea naivă poate conduce la erori

- vizitarea în ordinea bună a nodurilor fii conduce la retezări mai rapide

- cu "ordonare perfectă":
  - complexitatea timp poate scădea la $O(b^{m/2})$
  - algoritmul poate rula pe o adâncime dublă
  - căutarea exhuastivă nu este totuși posibilă…