Laborator 6+7 (Minimax, Alpha-Beta)

Teorie

Jocurile ca probleme de cautare

Ne referim la jocuri cu **2 jucatori** (care pe rand, *alternativ*, fac cate o mutare), cu **informatie completa**, adica ambii jucatori cunosc integral starea curenta a jocului si toate mutarile posibile din acea stare (*contra-exemplu*: jocurile de carti unde fiecare jucator stie doar cartile sale, dar nu si pe ale oponentului sau ordinea cartilor de pe masa, deci are informatie *incompleta*).

Mutarile sunt **deterministe**, nu includ probabilitati (de exemplu nu depind de aruncarea unui zar).

Jocul se incheie cand sa ajunge intr-o stare „terminala” conform regulilor jocului. Aceleasi reguli determina care este rezultatul jocului (care jucator a castigat sau daca eventual a fost remiza).

Un joc va fi reprezentat printr-un ***arbore de joc*** in care nodurile corespund starilor de joc, iar arcele corespund mutarilor. Radacina arborelui este starea initiala a jocului, iar frunzele arborelui sunt starile terminale ale jocului.

Cei doi jucatori vor fi numiti MAX si MIN. Jucatorul **MAX** este cel care face **prima mutare**, apoi cei doi jucatori muta alternativ pana se termina jocul. In arborele de joc, fiecare nivel contine mutarile unui anumit jucator: radacina pentru MAX, apoi toti fiii radacinii sunt pentru MIN, apoi nivelul urmator pentru MAX, si tot asa alternand nivelurile.

La finalul jocului, se acorda puncte jucatorului castigator (sau penalizari celui care a pierdut). Pentru asta, se va folosi o ***functie de utilitate***, care acorda o valoare numerica rezultatului unui joc (de exemplu 1 pentru castig, -1 pentru pierdere, 0 pentru remiza).

Jucatorul MAX ***incearca sa castige*** sau sa-si maximizeze scorul din acel moment. Jucatorul MIN, oponentul sau, ***incearca sa minimizeze scorul lui MAX***. Deci la fiecare pas, jucatorul MIN va alege acea mutare care este cea mai nefavorabila pentru MAX.

Jucatorul MAX trebuie sa gaseasca (folosind arborele de joc) o strategie care-l va conduce la castigarea jocului, indiferent de actiunile lui MIN.

In probleme, vom avea jucatorul MAX (**calculatorul** care isi alege mutarea folosind algoritmul) versus jucatorul MIN (**omul** care introduce de la tastatura mutarea dorita).

Pasii algoritmului Minimax

„1. Generează întregul arbore de joc, până la stările terminale.

2. Aplică funcţia de utilitate fiecărei stări terminale pentru a obţine valoarea corespunzătoare stării.

3. Deplasează-te înapoi în arbore, de la nodurile-frunze spre nodul-rădăcină, determinând, corespunzător fiecărui nivel al arborelui, valorile care reprezintă utilitatea nodurilor aflate la acel nivel. Propagarea acestor valori la niveluri anterioare se face prin intermediul nodurilor- părinte succesive, conform următoarei reguli:

• dacă starea-părinte este un nod de tip MAX, atribuie-i maximul dintre valorile avute de fiii săi;

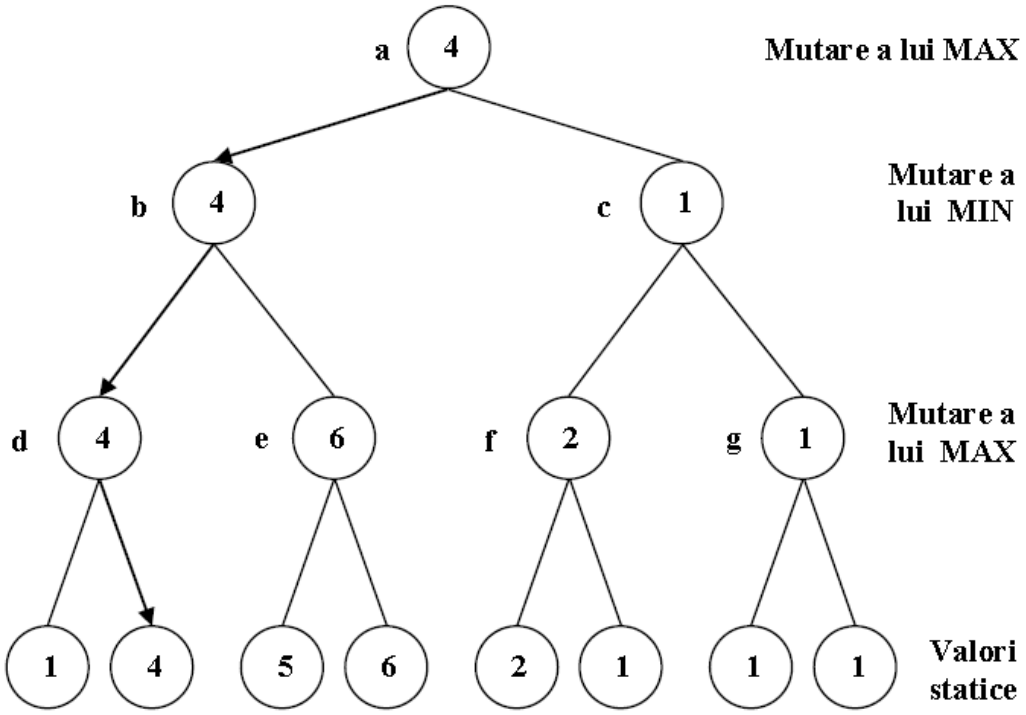
• dacă starea-părinte este un nod de tip MIN, atribuie-i minimul dintre valorile avute de fiii săi.

4. Ajuns în nodul-rădăcină, alege pentru MAX acea mutare care conduce la valoarea maximă.”

Observatie:

„Decizia luată la pasul 4 al algoritmului se numeşte *decizia minimax*, întrucât ea maximizează utilitatea, în ipoteza că oponentul joacă perfect cu scopul de a o minimiza.”

Exemplu de arbore de cautare pt alg Minimax



„Valorile poziţiilor de la ultimul nivel sunt determinate de către funcţia de utilitate şi se numesc *valori statice*. Valorile minimax ale nodurilor interne sunt calculate în mod dinamic, în manieră bottom-up, nivel cu nivel, până când este atins nodul-rădăcină. Valoarea rezultată, corespunzătoare acestuia, este 4 şi, prin urmare, cea mai bună mutare a lui MAX din poziţia *a* este *a-b.* Cel mai bun răspuns al lui MIN este *b-d*. Această secvenţă a jocului poartă denumirea de *variaţie principală*. Ea defineşte jocul optim de tip minimax pentru ambele părţi. Se observă că valoarea poziţiilor de-a lungul variaţiei principale nu variază. Prin urmare, mutările corecte sunt cele care *conservă valoarea jocului*."

In practica, nu vom genera *intreg arborele de cautare*, ci il vom extinde doar ***pana la o adancime maxima data***.

„Ideea este de a evalua aceste poziţii terminale ale căutării, fără a mai căuta dincolo de ele, cu scopul de a face economie de timp. Aceste estimări se propagă apoi în sus de-a lungul arborelui, conform principiului Minimax. Mutarea care conduce de la poziţia iniţială, nodul-rădăcină, la cel mai promiţător succesor al său (conform acestor evaluări) este apoi efectuată în cadrul jocului."

In algoritm vom inlocui *functia de utilitate* (aplicata doar starilor terminale de joc) cu ***functia de evaluare*** (care se poate aplica oricarei stari de joc, nu neaparat una terminala).

„O *funcţie de evaluare* întoarce o *estimaţie*, realizată dintr-o poziţie dată, a utilităţii aşteptate a jocului. Ea are la bază evaluarea şanselor de câştigare a jocului de către fiecare dintre părţi, pe baza calculării caracteristicilor unei poziţii. Performanţa unui program referitor la jocuri este extrem de dependentă de calitatea funcţiei de evaluare utilizate."

„***Funcţia de evaluare trebuie să îndeplinească anumite condiţii*** evidente: ea trebuie să concorde cu funcţia de utilitate în ceea ce priveşte stările terminale, calculele efectuate nu trebuie să dureze prea mult şi ea trebuie să reflecte în mod corect şansele efective de câştig."

O implementare eficientă a principiului Minimax: Algoritmul Alpha-Beta

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

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

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

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

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

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

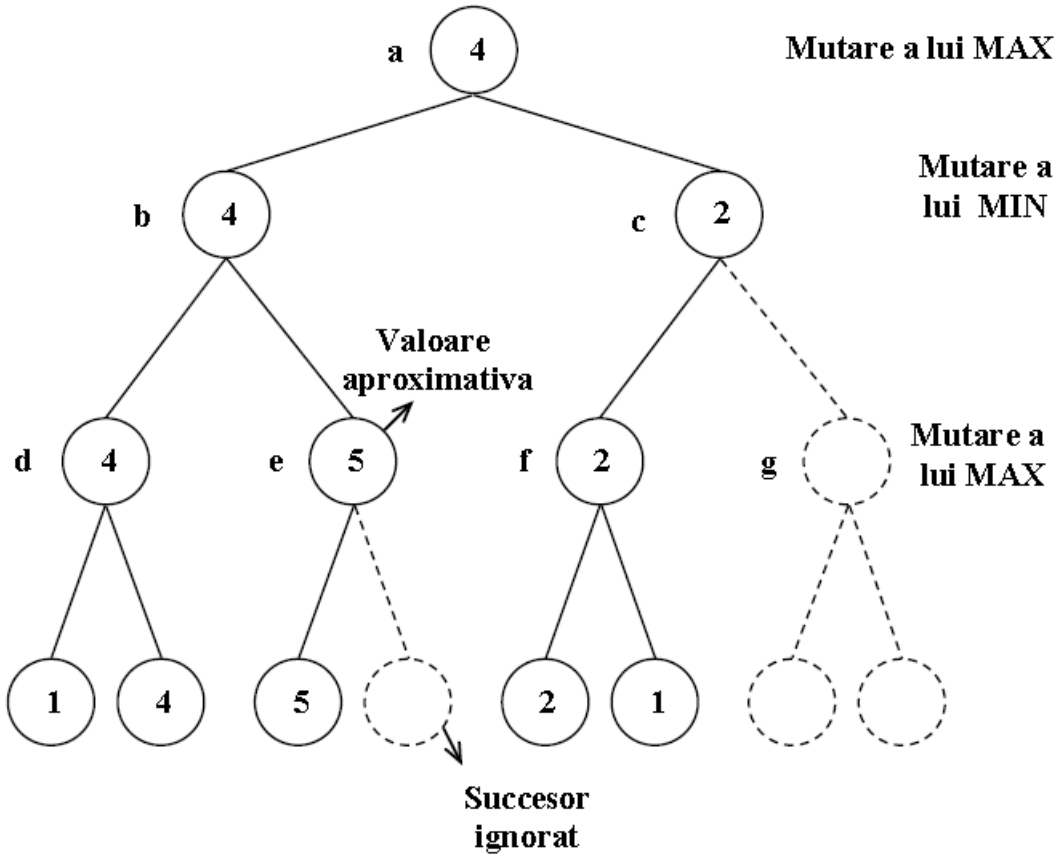
1. Căutarea poate fi oprită dedesubtul oricărui nod de tip MIN care are o valoare beta mai mică sau egală cu valoarea alpha a oricăruia dintre strămoşii săi de tip MAX.
2. Cautarea poate fi oprită dedesubtul oricărui nod de tip MAX care are o valoare alpha mai mare sau egală cu valoarea beta a oricăruia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

„Dacă, referitor la o poziţie, se arată că valoarea corespunzătoare ei se află în afara intervalului alpha-beta, atunci această informaţie este suficientă pentru a şti că poziţia respectivă nu se află de-a lungul *variaţiei principale*, chiar dacă nu este cunoscută valoarea exactă corespunzătoare ei. Cunoaşterea valorii exacte a unei poziţii este necesară numai atunci când această valoare se află între alpha şi beta.”

„Din punct de vedere formal, putem defini o *valoare de tip minimax* a unui nod intern, P, V(P, alpha, beta), ca fiind “*suficient de bună*” dacă satisface următoarele cerinţe:
V(P, alpha, beta) < alpha, dacă V(P) < alpha
V(P, alpha, beta) = V(P), dacă alpha ≤ V(P) ≤ beta
V(P, alpha, beta) > beta, dacă V (P) > beta,
unde prin V(P) am notat valoarea de tip minimax corespunzătoare unui nod intern.
Valoarea exactă a unui nod-rădăcină P poate fi întotdeauna calculată prin setarea limitelor după cum urmează:
V(P, -∞, +∞) = V(P).”

Exemplu de arbore de cautare pt alg Alpha-Beta:

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



„Procesul de căutare decurge după cum urmează:

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

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

„Eficienţa Algoritmului Alpha-Beta depinde de *ordinea în care sunt examinaţi succesorii.* Este preferabil să fie examinaţi mai întâi succesorii despre care se crede că ar putea fi cei mai buni.”

Exerciții

Porniți de la clasele din fișierul main.py din folderul laboratorului 6.

1) **Familiarizarea cu codul și implementarea regulilor jocului x și 0.** Inspectați codul (incomplet) oferit pentru acest laborator și citiți comentariile care descriu clasele, proprietățile, metodele.
Ne vom ocupa de clasa **InfoJoc**. Vom nota cu GOL simbolul pentru locul gol din tabla de joc, iar cu JMAX și JMIN simbolul cu care joacă calculatorul, respectiv utilizatorul (acestea pot fi 'x' sau '0'). Proprietatea NR_COLOANE indică dimensiunea tabeli de joc.
Realizați următoarele sarcini:

a. Instalați modulul **pygame**.
b. Implementați **constructorul clasei InfoJoc**. Acesta primește ca parametru tabla de joc (cu valoare implicită None) și setează proprietatea *matr* (care va contine tabla de joc) a instanței fie la valoarea parametrului, dacă este dat, fie va crea o matrice(listă de liste) 3*3 de simboluri GOL.
c. Implementați metoda de clasă **jucator_opus(),** care primește ca parametru simbolul unui jucător și returnează simbolul adversarului (de exemplu, dacă primește 'x', returnează '0').
d. Creați o metodă **mutari**(), care primește simbolul jucătorului curent și generează toate mutările posibile pentru acesta. Mutările trebuie să fie tot obiecte de tip InfoJoc. Va returna lista de mutari (obiecte InfoJoc).

2) **Finalizarea jocului.**

a. Creați o funcție (în afara clasei) **elem_identice**(lista) care verifică dacă toate elementele dintr-o listă sunt identice și diferite de simbolul GOL si returneaza simbolul daca e diferit de GOL, altfel False.
b. Creați o metodă **final()** care verifică dacă jocul x și 0 a ajuns în stare finală: dacă a câștigat un jucător, returnează simbolul jucătorului. Dacă s-a terminat cu remiză, returnează "remiza". Dacă nu e stare finală, returnează False.

3) **Estimarea scorului**

a. Definiți metoda **linie_deschisa**() care primește simbolurile de pe o linie și simbolul jucătorului pentru care vrem să verificăm dacă e linie deschisă. O linie deschisa este o linie(rând, coloană, diagonală) pe care jucătorul mai poate încă să facă o configurație câștigătoare (practic e o linie fara simboluri ale jucatorului opus). Returneaza True (sau 1) dacă e linie deschisă și False dacă nu este.

b. Definiți metoda **linii_deschise**() care primește simbolul unui jucător și calculează numărul de linii deschise pentru acesta.
c. Definiți funcția **estimeaza_scor**() care să primească un indicator al distanței față de rădăcină(în arborele minimax) care e cu atât mai mare cu cat nodul e mai departe de rădăcină (restAdancime = cat mai are până la adancimea maximă), astfel încât:
    - dacă e o stare finală în care a câștigat MAX, să returneze un număr foarte mare (totuși o stare finală mai apropiată de rădăcină în care MAX a câștigat trebuie să aibă scor mai mare decât o stare în care MAX a câștigat dar e mai depărtată de rădăcină în arborele
    - dacă e o stare finală în care a câstigat MIN, să returneze un număr foarte mic (totuși o stare finală mai apropiată de rădăcină în care MIN a câștigat trebuie să aibă scor mai mic decât o stare în care MIN a câștigat dar e mai depărtată de rădăcină în arborele
    - dacă e o stare finală de tip remiză, returnează 0
    - dacă nu e stare finală estimează scorul la numărul de linii deschise ale lui MAX din care scădem numărul de linii deschise ale lui MIN

4) Implementați funcția **minimax()** care primește un obiect de tip nod (stare) al arborelui minimax și, dacă e stare finală sau a ajuns la adâncimea maximă, îi calculează estimarea. Iar dacă e stare intermediară, îi calculează succesorii dintre care îl alege pe cel mai bun (și setează proprietatea stare_aleasa egală cu acesta) în funcție de tipul de nod (MIN sau MAX).

5) Implementați funcția **alphabeta()**, care primește un obiect de tip nod (stare) al arborelui minimax și valorile alpha și beta pentru acesta, și, dacă nodul-argument e stare finală sau a ajuns la adâncimea maximă, îi calculează estimarea. Iar dacă e stare intermediară, îi calculează succesorii dintre care îl alege pe cel mai bun (și setează proprietatea stare_aleasa egală cu acesta) în funcție de tipul de nod (MIN sau MAX). în cadrul calculării succesorilor, se vor calcula și valorile alpha și beta pentru fiecare nod.

Exercitii de recuperare punctaj (fiecare e de 0.5p din nota de laborator KR):
Atenție, aceste exercitii trebuie prezentate

6) Imbunătățiți funcția de estimare a valorii minimax pentru jocul **x și 0** aplicând următoarele 3 principii:
a) O linie deschisă fără simboluri (nici x și nici 0) e teren neutru deci nu se va aduna nici pentru jucătorul MIN, nici pentru MAX. O linie deschisă pentru un jucător care are deja 2 simboluri, ar trebui să aibă o pondere mai mare (de exemplu, 2) decât o linie deschisă cu un singur simbol al jucătorului (care de exemplu, să aibă pondere 1)
b) chiar dacă jocul nu e în stare finală, dar starea evaluată are proprietatea că prin mutarea următoare a jucătorului curent acesta poate câștiga, atunci, să se returneze scorul pentru cazul de câștig al jucătorului curent: un număr foarte mare pentru MAX (de exemplu, 99) și un număr foarte mic (de exemplu, -99) pentru MIN.
Exemplu:

jucatorul curent e X, dar starea evaluată e frunză în arborele minimax, deoarece s-a ajuns la adâncimea maximă.
dacă starea e cea de mai jos:
x#0
#0x
x0#
fiind rândul lui x, el poate să plaseze un simbol pe linia 1, coloana 0 și să câștige.
Deci în funcția de evaluare_scor veți verifica dacă există vreo linie cu 2 simboluri ale jucătorului curent și un loc liber.
c) Chiar dacă jocul nu e în stare finală, dar starea evaluată are proprietatea că un jucător are 2 variante de câștig (două poziții distincte în care poata plasa un simbol astfel încât să câștige), atunci, să se returneze scorul pentru cazul de câștig al jucătorului curent: un număr foarte mare pentru MAX (de exemplu, 99) și un număr foarte mic (de exemplu, -99) pentru MIN.
dacă starea e cea de mai jos:
x#0
#x#
x0#
x poate să câștige chiar dacă nu e rândul lui pentru că 0 nu poate bloca decât una dintre variantele lui de câștig.

7)Se citeste un numar N de la tastatura (intre 3 si 5 inclusiv - daca se citeste alt numar, se da un mesaj de eroare si se opreste programul) care va reprezenta dimensiunea tablei de joc (tabla va fi NxN). Schimbati functia de generare a mutarilor, de verificare a starii scop si de estimare a scorului in urmtoarele situatii:
a) Regulile de la x si 0 se pastreaza, insa configuratia castigatoare trebuie sa aiba dimensiunea egala cu N (o linie, o coloana sau o diagonala cu N simboluri identice)
b) Regulile de la x si 0 se pastreaza, insa configuratia castigatoare trebuie sa aiba dimensiunea egala cu K citit de la tastatura si 3<=K<=N (o linie, o coloana sau o diagonala cu K simboluri identice consecutive, deci vecine intre ele pe linie/coloana/diagonala; de exemplu pentru 0, si K=3, o linie poate fi x000#)
c) Regulile de la x si 0 se pastreaza, insa configuratia castigatoare trebuie sa aiba dimensiunea egala cu K citit de la tastatura si 3<=K<=N (o linie, o coloana sau o diagonala cu K simboluri identice dar nu neaparat consecutive - deci pe o linie/coloana/diagonala se gasesc K simboluri identice oricum dispuse, de exemplu pentru x, si K=3, o linie poate fi x0#xx).
Utilizatorul va fi întrebat la începutul jocului care dintre variantele a, b și c dorește să joace.

8) Implementați, folosind cadrul dat la laborator Connect four cu următoarea modificare: când se face click pe o coloană pe care vrem să plasăm simbolul, acesta urcă în loc să coboare până ajunge pe primul rând sau pe o poziție care are imediat deasupra un alt simbol.

9) Implementați, folosind cadrul dat la laborator x si 0 Ultimate în varianta Misere.

10) Pornind de la implementarea data pentru x si 0, se va implementa jocul Achi fie in consola fie pe interfata grafica (insa nu mai desenati liniile de pe gridul de joc si considerati punctele pe care se pun piesele ca fiind celulele gridului). Se vor schimba functiile de generare a mutarilor, evaluare a scorului (atât pestru stările finale cât și pentru cele nefinale dar care sunt frunze din cauza adâncimii maxime) dar și zona de cod dedicată mutării utilizatorului.

11) Modificați programul dat pentru x si 0 pentru a implementa reversi (veți folosi x pentru negru și 0 pentru alb). Nu se acceptă implementări care nu pornesc de la modelul dat la laborator (pentru a evita copierea programului de pe net).

Anexe

```
    def final(self):
        rez = elem_identice(self.matr[0]) \
              or elem_identice(self.matr[1]) \
              or elem_identice(self.matr[2]) \
              or elem_identice([self.matr[0][0], self.matr[1][0], self.matr[2][0]]) \
              or elem_identice([self.matr[0][1], self.matr[1][1], self.matr[2][1]]) \
              or elem_identice([self.matr[0][2], self.matr[1][2], self.matr[2][2]]) \
              or elem_identice([self.matr[0][0], self.matr[1][1], self.matr[2][2]]) \
              or elem_identice([self.matr[0][2], self.matr[1][1], self.matr[2][0]])
        if rez:
            return rez
        remiza = True
        for linie in self.matr:
            for elem in linie:
                if elem == InfoJoc.GOL:
                    remiza = False
        if remiza:
            return "remiza"
        return False
```

```
    def linie_deschisa(self, lista, jucator):
        return not InfoJoc.jucator_opus(jucator) in lista

    def linii_deschise(self, jucator):
        return self.linie_deschisa(self.matr[0], jucator) \
               + self.linie_deschisa(self.matr[1], jucator) \
               + self.linie_deschisa(self.matr[2], jucator) \
               + self.linie_deschisa([self.matr[0][0], self.matr[1][0], self.matr[2][0]], jucator) \
               + self.linie_deschisa([self.matr[0][1], self.matr[1][1], self.matr[2][1]], jucator) \
               + self.linie_deschisa([self.matr[0][2], self.matr[1][2], self.matr[2][2]], jucator) \
               + self.linie_deschisa([self.matr[0][0], self.matr[1][1], self.matr[2][2]], jucator) \
               + self.linie_deschisa([self.matr[0][2], self.matr[1][1], self.matr[2][0]], jucator)
```