Implementare

1) Clasa Joc se refera o configuratie de joc (un nod din graful jocului).

→ ***Atributele clasei Joc*** sunt cele care definesc tabla de joc si care NU se modifica deloc pe durata intregului joc (dar care pot fi modificate de la un joc la altul), de exemplu:

NR_LINII, NR_COLOANE (dimensiunea tablei de joc)

SIMBOLURI_JUC (lista cu cele 2 simboluri care vor fi folosite de jucatori)

JMIN, JMAX, GOL (simbolurile pentru cei 2 jucatori si simbolul pentru casuta necompletata)

→ ***Atributele unui obiect din clasa Joc*** sunt cele care se pot modifica de la o configuratie de joc la urmatoarea (in urma executarii unei mutari *valide* a jocului): „matr”

a) daca reprezentam tabla de joc ca *lista simpla*:

```
def __init__(self, tabla=None):
    self.matr = tabla or [Joc.GOL]*(Joc.NR_COLOANE * Joc.NR_LINII)
```

b) daca reprezentam tabla de joc ca matrice (*lista de liste*):

```
def __init__(self, tabla=None):
    self.matr = tabla or [[Joc.GOL for j in Joc.NR_COLOANE] for i in Joc.NR_LINII]
```

→ ***Metodele unui obiect din clasa Joc***:

- `final(self)` → Returneaza fie simbolul jucatorului care a castigat jocul, fie „remiza”, fie False daca jocul nu s-a terminat inca.

- `mutari_joc(self, jucator)` → Returneaza o lista de obiecte de tip Joc, reprezentand toate configuratiile de joc valide ca succesori, obtinute plecand din configuratia curenta „self” si adaugand o mutare valida a lui „jucator”.

- alte metode ajutatoare pentru metoda „estimeaza_scor”…
- `estimeaza_scor(self, adancime)` → Apeleaza `self.final()`, apoi:

  → Daca „self” este configuratie finala de joc, atunci returneaza:
  - 0, daca jocul s-a terminat cu „remiza”
  - un numar foarte mare (> 0), daca a castigat JMAX
  - un numar foarte mic (< 0), daca a castigat JMIN

  Obs: Putem folosi „adancime” pentru a considera **mai buna** o configuratie finala de joc in care s-a castigat din mai putine mutari.

  → Daca „self” NU este configuratie finala de joc (jocul este inca in derulare), atunci se face diferenta:
  - (sansele de castig pentru JMAX) – (sansele de castig pentru JMIN)

- `__str__(self)` → Suprascriem aceasta metoda pentru a *returna un obiect de tip* ***str***, cu care vom face afisarea frumoasa (a listei simple sau a listei de liste) „self.matr” sub forma de matrice, cu spatiu intre elementele de pe aceeasi linie.

2) Clasa Stare se refera la un nod din arborele de cautare al algoritmului.

→ ***Atributele clasei Stare*** sunt cele care NU se modifica pe durata aplicarii algoritmului.

`ADANCIME_MAX` → Un numar natural nenul care reprezinta maxim pe cate nivele are voie algoritmul sa extinda arborele de cautare. Numarul se stabileste la inceput, apoi se aplica algoritmul.

→ ***Atributele unui obiect din clasa Stare*** sunt cele care se pot modifica de la o configuratie de joc la urmatoarea (in urma executarii unei mutari *valide* a jocului):

```
def __init__(self, tabla_joc, j_curent, adancime, parinte=None, scor=None):

    self.tabla_joc = tabla_joc # un obiect de tip Joc => „tabla_joc.matr”
    self.j_curent = j_curent # simbolul jucatorului curent

    # adancimea in arborele de stari
    #     (scade cu cate o unitate din „tata” in „fiu”)
    self.adancime = adancime

    # scorul starii (daca e finala, adica frunza a arborelui)
    # sau scorul celei mai bune stari-fiice (pentru jucatorul curent)
    self.scor = scor

    # lista de mutari posibile din starea curenta
    self.mutari_posibile = [] # lista va contine obiecte de tip Stare

    # cea mai buna mutare din lista de mutari posibile pentru jucatorul curent
    self.stare_aleasa = None
```

→ ***Metodele unui obiect din clasa Stare***:

- `jucator_opus(self)` → Returneaza simbolul celuilalt jucator fata de „`self.j_curent`”, folosind simbolurile din `Joc.JMAX` si `Joc.JMIN`.

- `mutari_stare(self)` → Returneaza o lista de obiecte de tip Stare, reprezentand toti fiii posibili ai nodului curent „self” in arborele de cautare.

- `__str__(self)` → Suprascriem aceasta metoda pentru a *returna un obiect de tip* ***str***, cu care vom face afisarea configuratiei curente de joc (`str(self.tabla_joc)`), precum si simbolul jucatorului curent (`self.j_curent`). (Daca doriti, puteti include in afisare si alte atribute, precum „`self.scor`”, dar nu aglomerati prea tare afisarea, pentru a usor de umarit.)

3) Algoritmii Minimax si Alpha-Beta

→ *Pseudocod Minimax:*

Pseudocod de pe Wikipedia
(https://en.wikipedia.org/wiki/Minimax#Pseudocode)

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth − 1, TRUE))
        return value
```

```
(* Initial call *)
minimax(origin, depth, TRUE)
```

→ *Implementare Minimax:*

- Functia „`min_max(stare)`” → Primeste ca parametru un obiect de tip Stare, caruia ii actualizeaza valorile atributelor „scor” si eventual (daca nu e nod frunza) „mutari_posibile”, „stare_aleasa”, apoi returneaza tot acest obiect.

```
def min_max(stare):

    # Daca am ajuns la o frunza a arborelui, adica:
    # - daca am expandat arborele pana la adancimea maxima permisa
    # - sau daca am ajuns intr-o configuratie finala de joc
    if stare.adancime == 0 or stare.tabla_joc.final() :
        # calculam scorul frunzei apeland "estimeaza_scor"
        stare.scor = stare.tabla_joc.estimeaza_scor(stare.adancime)
        return stare


    # Altfel, calculez toate mutarile posibile din starea curenta
    stare.mutari_posibile = stare.mutari_stare()

    #aplic algoritmul minimax pe toate mutarile posibile
    #    (calculand astfel subarborii lor)
    mutari_scor = [min_max(mutare) for mutare in stare.mutari_posibile]

    if stare.j_curent == Joc.JMAX :
        #daca jucatorul e JMAX aleg starea-fiica cu scorul maxim
        stare.stare_aleasa = max(mutari_scor, key=lambda x: x.scor)
    else:
        #daca jucatorul e JMIN aleg starea-fiica cu scorul minim
        stare.stare_aleasa = min(mutari_scor, key=lambda x: x.scor)

    # actualizez scorul „tatalui” = scorul „fiului” ales
    stare.scor = stare.stare_aleasa.scor

    return stare
```

→ *Pseudocod Alpha-Beta:*

Pseudocod de pe Wikipedia
(https://en.wikipedia.org/wiki/Alpha-beta_pruning#Pseudocode)

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            α := max(α, value)
            if α ≥ β then
                break (* β cut-off *)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            β := min(β, value)
            if α ≥ β then
                break (* α cut-off *)
        return value
```

```
(* Initial call *)
alphabeta(origin, depth, −∞, +∞, TRUE)
```

→ *Implementare Alpha-Beta:*

- Functia „`alpha_beta(alpha, beta, stare)`" → Primeste ca parametri capetele intervalului, „alpha" si „beta", si un obiect de tip Stare, caruia ii actualizeaza valorile atributelor „scor" si eventual (daca nu e nod frunza) „mutari_posibile", „stare_aleasa", apoi returneaza tot acest obiect.

*!!! **Observatii** pentru a intelege mai bine ideea algoritmului:*

a) Intervalul [alpha, beta] se transmite *doar de sus in jos (de la tata catre fiul curent)*.
- La inceputul algoritmului, intervalul este egal cu **(- infinit, + infinit)**.
- Pe durata algoritmului, acest interval doar *se va restrange* (niciodata NU se va largi), pentru ca nodurile MAX „trag" de capatul alpha in sus, iar nodurile MIN „trag" de capatul beta in jos.
- In orice moment, se stie ca scorul final pe care il va avea radacina arborelui va fi *in interiorul acestui interval*. De aceea, cand un nod ajunge sa aiba alpha>=beta, restul fiilor sai pot fi ***retezati*** *fara ca asta sa afecteze* raspunsul final al algoritmului (scorul radacinii arborelui).

b) La intrarea in fiecare nod, se initializeaza **scorul nodului**:
- Daca nodul este *frunza*, se apeleaza „estimeaza_scor"
- Altfel, scorul este cea mai *rea* valoare pentru acel tip de nod:
  - (- infinit) pentru nodurile de tip MAX
  - (+ infinit) pentru nodurile de tip MIN

Scorul se transmite *doar de jos in sus (de la fiul curent catre tatal sau)*.

c) In nodul curent, pentru fiecare dintre fiii sai:
- Se apeleaza functia „alpha_beta" pentru fiul curent → adica se aplica algoritmul pentru intreg subarborele (cu radacina = fiul curent) → fiul curent isi afla scorul, pe care i-l trimite tatalui sau.
- Tatal, cu ajutorul scorului primit de la fiul curent, incearca sa-si imbunatateasca scorul sau, precum si unul din capetele intervalului [alpha, beta]:
  - Daca tatal este de tip MAX:
    scor_tata = max (scor_tata, scor_fiu)
    alpha = max (alpha, scor_fiu) # MAX nu are voie sa-l modifice pe beta
  - Daca tatal este de tip MIN:
    scor_tata = min (scor_tata, scor_fiu)
    beta = min (beta, scor_fiu) # MIN nu are voie sa-l modifice pe alpha

```
def alpha_beta(alpha, beta, stare):

    # Daca am ajuns la o frunza a arborelui, adica:
    # - daca am expandat arborele pana la adancimea maxima permisa
    # - sau daca am ajuns intr-o configuratie finala de joc
    if stare.adancime == 0 or stare.tabla_joc.final() :
        # calculam scorul frunzei apeland "estimeaza_scor"
        stare.scor = stare.tabla_joc.estimeaza_scor(stare.adancime)
        return stare

    # Conditia de retezare:
    if alpha >= beta:
        return stare # este intr-un interval invalid, deci nu o mai procesez


    # Calculez toate mutarile posibile din starea curenta (toti „fiii”)
    stare.mutari_posibile = stare.mutari_stare()

    if stare.j_curent == Joc.JMAX :
        scor_curent = float('-inf') # scorul „tatalui” de tip MAX

        # pentru fiecare „fiu” de tip MIN:
        for mutare in stare.mutari_posibile:
            # calculeaza scorul fiului curent
            stare_noua = alpha_beta(alpha, beta, mutare)

            # incerc sa imbunatatesc (cresc) scorul si alfa
            # „tatalui” de tip MAX, folosind scorul fiului curent
            if scor_curent < stare_noua.scor:
                stare.stare_aleasa = stare_noua
                scor_curent = stare_noua.scor

            if alpha < stare_noua.scor:
                alpha = stare_noua.scor
                if alpha >= beta: # verific conditia de retezare
                    break   # NU se mai extind ceilalti fii de tip MIN


    elif stare.j_curent == Joc.JMIN :
        scor_curent = float('inf') # scorul „tatalui” de tip MIN

        # pentru fiecare „fiu” de tip MAX:
        for mutare in stare.mutari_posibile:
            stare_noua = alpha_beta(alpha, beta, mutare)

            # incerc sa imbunatatesc (scad) scorul si beta
            # „tatalui” de tip MIN, folosind scorul fiului curent
            if scor_curent > stare_noua.scor:
                stare.stare_aleasa = stare_noua
                scor_curent = stare_noua.scor

            if beta > stare_noua.scor:
                beta = stare_noua.scor
                if alpha >= beta: # verific conditia de retezare
                    break   # NU se mai extind ceilalti fii de tip MAX


    # actualizez scorul „tatalui” = scorul „fiului” ales
    stare.scor = stare.stare_aleasa.scor

    return stare
```