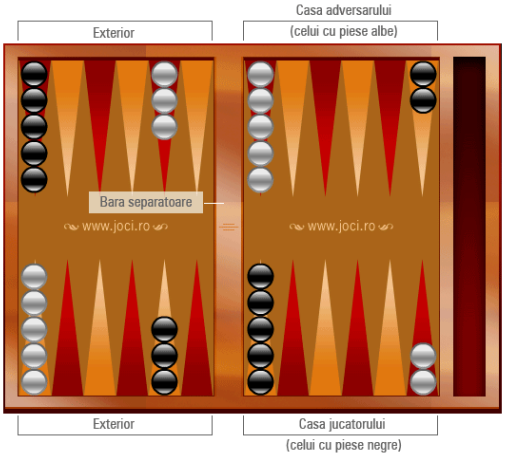
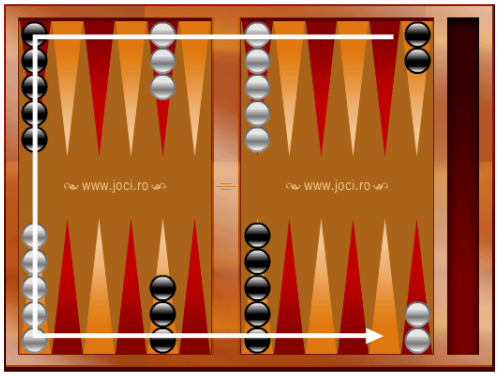
Algoritm MCTS pentru Backgammon (Table)

1. Descrierea problemei considerate

Backgammon (Table) este un joc popular cu 2 jucători care combină norocul și strategia. Jocul se joacă pe o tablă formată din 24 de triunghiuri înguste (locuri unde sunt așezate piesele). Triunghiurile au culori alternante și sunt grupate în 4 zone a cate 6 triunghiuri fiecare. Zonele sunt denumite după cum urmează: casă și exterior (în deplasare) ale jucătorului și casă și exterior (în deplasare) ale adversarului. Casele și câmpurile exterioare sunt separate printr-o bară în centrul tablei.



Scopul ambilor jucători este să-și mute piesele proprii în casă și să le scoată din aceasta înaintea adversarului. În imaginea următoare se poate observa direcția în care trebuiesc mutate piesele negre, piesele albe trebuiesc mutate în direcția opusă.



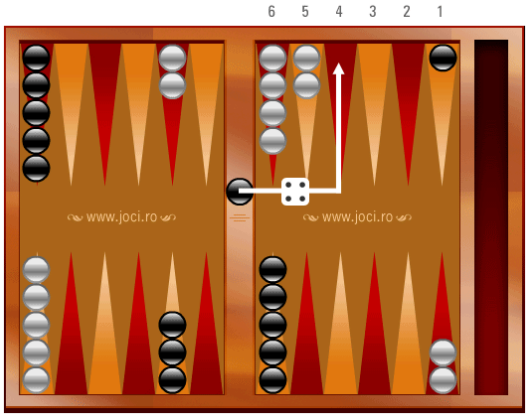
Jocul se desfășoară în felul următor:

a. Se arunca cele doua zaruri pentru a determina numărul de triunghiuri (locuri) după care se vor muta piesele, fiecare zar reprezentând o mutare diferită.
b. O piesă poate fi mutată într-un loc doar dacă acesta este liber, adică nu este ocupat de 2 sau mai multe piese ale adversarului.

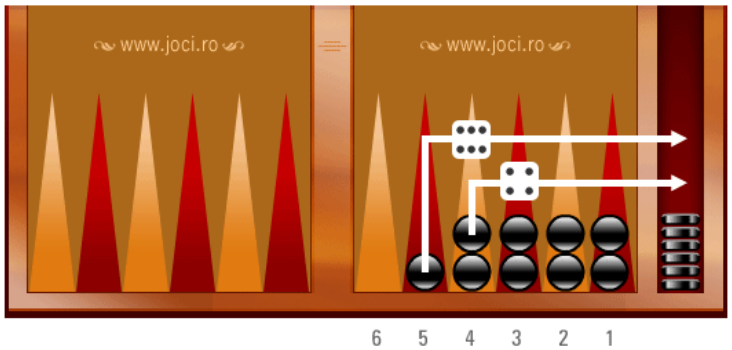
c. Dacă în urma aruncarii zarurilor, un jucător nimereste o dubla (ambele zaruri indica același număr), atunci acesta va putea face două mutări pentru fiecare zar.
d. Un jucător trebuie sa folosească toate mutarile dacă este posibil (două sau patru, dacă a dat dublă).
e. Dacă poate fi făcută doar o mutare, se va face mutarea cu numărul cel mai mare.
f. Dacă nu se poate face nici o mutare jucătorul își va pierde tura.

Alte aspecte ale jocului sunt:

- Capturarea pieselor:
    - Dacă un loc este ocupat de o singura piesa, aceasta poate fi capturată de o piesa a inamicului, în cazul în care piesa adversă ajunge pe același triunghi (loc). Piesa capturată trebuie așezată pe bara (axa tablei de joc).
    - Cand un jucator are una sau mai multe piesa capturate, acesta este obligat sa le repuna în joc, în casa adversarului, pe un loc liber în concordanță cu numerele indicate de zaruri.
    - Daca nu exista locuri libere in casa adversarului, jucatorul își pierde tura.



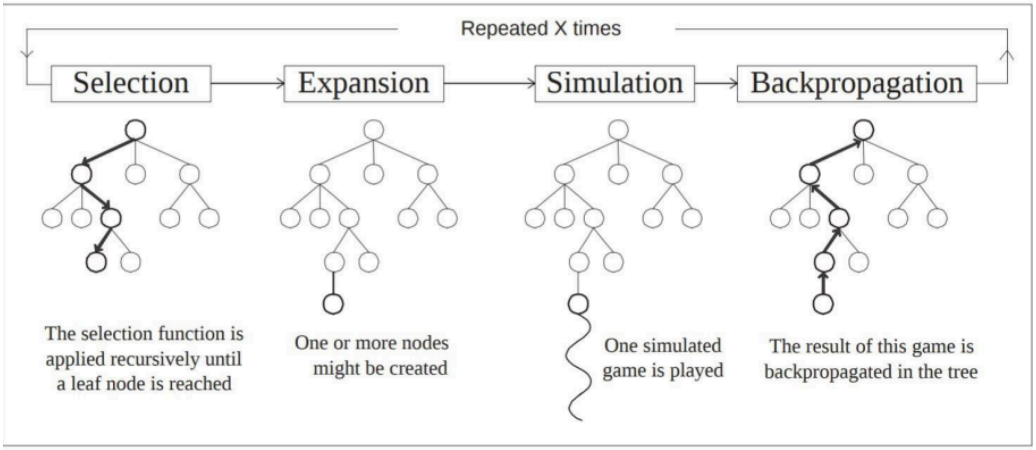
- Scoaterea pieselor:
    - După ce un jucător și-a adus toate cele 15 piese în casă, acesta poate începe scoaterea lor din joc.
    - Jucatorul da cu zarurile, dupa care scoate piesele indicate de acestea. Daca nu exista piese pe locurile respective, jucatorul trebuie sa mute (conform regulilor) o piesa de pe un loc al carui numar este mai mare.



2. Aspecte teoretice privind algoritmul MCTS

Algoritmul MCTS (Monte Carlo Tree Search) reprezintă o metodă de căutare stochastică de căutare a soluțiilor, utilizat adesea in jocuri de tip board game (ex. Șah, Backgammon, go etc.). Algoritmul MCTS excelează în gestionarea problemelor unde spațiul de căutare este prea mare pentru a fi explorat exhaustiv.

MCTS utilizează simulări aleatoare pentru a construi un arbore de cautare, prin care sa estimeze valoarea potențială a mutarilor. Algoritmul constă în patru pași care se repeta. (1) Se parcurge arborele de la rădăcina pana la un nod frunza, utilizand o strategie de selecție. (2) Se apelează o strategie de expandare (se poate realiza și aleatoriu) pentru a selecta noduri frunza încă nevizitate și a adăuga cate o nouă înregistrare pentru fii acestuia. (3) Se aleg aleatoriu, sau folosind o euristică, mutări pană se ajunge într-o stare terminală. (4) Rezultatul simularii este retro-propagat in arbore.



Explicație amănunțită a pașilor algoritmului:

Selecția este o etapa strategică care alege unul dintre copiii unui nod dat. Într-un nod aleator, următoarea mutare va fi aleasă la întâmplare, în timp ce într-un nod de alegere ține cont de echilibrul dintre exploatare și explorare. Pe de o parte, sarcina este adesea de a alege mișcarea care va aduce cel mai bun rezultate (exploatare). Pe de altă parte, mutările mai puțin promițătoare trebuie să fie explorate și ele, din cauza incertitudinii de evaluare (explorare).

Expandarea decide dacă pentru un nod frunză, copiii acestuia vor fi sau nu stocați în memorie. Cea mai simplă regulă este expandarea unui singur nod per simulare, nodul expandat reprezentând prima poziție întâlnită care nu a fost stocată încă.

Simularea (playout) este responsabilă de alegerea mutărilor în sine. Aceasta poate fi realizată la întâmplare, însă s-a dovedit că utilizarea unei euristici adecvate rezultă într-o îmbunătățire semnificativă a rezultatelor.

Actualizarea este procedeul prin care rezultatul unui joc simulat (victorie/înfrângere) este retro-propagat prin toate nodurile traversate pentru a ajunge la frunză.

În cele din urmă, mișcarea jucată de program este fiul rădăcinii cu cel mai mare număr de vizite.

3. Modalitatea de rezolvare

Implementarea AI-ului pentru Backgammon cu MCTS s-a realizat prin următoarele etape:

1. **Modelarea jocului Backgammon**:

   Jocul este reprezentat prin clasa BackgammonState din fișierul reguli.py. Aceasta include:

   - Structura tablei de joc:
     - Tabla este o listă de 24 de poziții, unde fiecare element reprezintă numărul de piese și jucătorul căruia îi aparțin. Valorile pozitive indică piesele jucătorului 1, iar cele negative piesele jucătorului -1.
   - Gestionarea regulilor:
     - Mutări legale: Funcția get_legal_moves returnează toate mutările posibile pentru un jucător, incluzând mutările din bară sau scoaterea pieselor din joc.
   - Zaruri duble: roll_dice generează de 4 ori același zar dacă cele două zaruri aruncate sunt egale.
   - Scoaterea pieselor: get_bearing_off_moves permite scoaterea pieselor doar dacă toate piesele unui jucător sunt în zona "home".
   - Aplicația mutării: apply_move modifică starea tablei în funcție de mutarea efectuată.
   - Verificarea stării terminale: is_terminal determină dacă un jucător a scos toate cele 15 piese.
   - Clonare: Funcția clone creează o copie a stării curente pentru a permite simulări independente fără a afecta jocul real.
   - Această modelare permite AI-ului să interacționeze cu tabla și să simuleze diferite scenarii.

2. **Implementarea algoritmului MCTS**:

   Implementarea MCTS se află în fișierul mcts.py și include:

   - Structura arborelui MCTS: Nodurile arborelui: Clasa Node reprezintă fiecare stare a jocului, păstrând informații despre mutarea care a dus la această stare, scorurile euristice și nodurile copil.
   - Expansiune și selecție:
     - Nodurile sunt expandate folosind mutări neexplorate din starea curentă (expand).

    - În procesul de selecție, nodul copil este ales pe baza unui echilibru între scorul mediu și explorare (UCT).
- Simulare: Funcția simulate_n_moves simulează un număr fix de mutări înainte de a aplica o funcție euristică (evaluate_state) pentru a evalua starea intermediară.
- Euristici: evaluate_state calculează scoruri bazate pe factori precum:
    - Blocarea pozițiilor strategice (ex. zonele 17-23).
    - Avansarea pieselor spre zona "home".
    - Siguranța pieselor (mai mult de o piesă pe poziție).
    - Penalizarea pieselor vulnerabile.
    - Scoaterea pieselor proprii din joc.
- Backpropagation: Rezultatele simulărilor sunt propagate înapoi pentru a actualiza statistici precum scorul mediu și numărul de vizite.
- La final, MCTS alege mutarea cu cel mai mare scor mediu și număr de vizite.

3. **Integrarea AI-ului în joc**:

   Integrarea AI-ului în joc se află în fișierul main.py. Jocul permite interacțiunea între un jucător uman și AI-ul bazat pe MCTS:

   - Interacțiunea cu utilizatorul:
       - Jucătorul uman primește opțiuni de mutări legale pe baza zarurilor aruncate.
       - AI-ul utilizează MCTS pentru a calcula cea mai bună mutare în funcție de numărul de simulări specificat de utilizator la începutul jocului.
   - Alternarea turelor: Jocul alternează între jucătorul uman și AI, iar tabla este actualizată și afișată după fiecare tură.
   - Determinarea câștigătorului: Jocul se termină când unul dintre jucători își scoate toate piesele, iar câștigătorul este anunțat.
   - Funcționarea generală a jocului: Jocul începe prin inițializarea tablei (BackgammonState) și setarea numărului de simulări pentru MCTS.
   - În fiecare tură:
       - Zarurile sunt aruncate, iar mutările legale sunt calculate.
       - Jucătorul uman sau AI-ul aplică o mutare bazată pe regulile jocului.
       - AI-ul folosește MCTS pentru a explora mutările și pentru a calcula cea mai bună decizie bazată pe simulări.
       - Jocul continuă până la atingerea unei stări terminale, determinând câștigătorul

4. Părți semnificative din codul sursă

Cele mai importante secvențe de cod se regăsesc în fișierul mcts.py, și anume:

a. mcts(stareInitiala, zar, simulari, nrMutari=10)

```python
def mcts(stareInitiala, zar, simulari, nrMutari=10):
    # Implementarea algoritmului Monte Carlo Tree Search
    radacina = Nod(stareInitiala, zar=zar)  # Creăm nodul rădăcină

    # Alocăm simulările pe baza scorurilor mutărilor
    while radacina.mutareNetestata:
        radacina.expandare(zar)  # Expandează toate mutările posibile

    scorTotal = sum(child.scorMediu() for child in radacina.copil)
    if scorTotal > 0:
        alocare = [int(simulari * (child.scorMediu() / scorTotal)) for child in radacina.copil]
    else:
        alocare = [simulari // len(radacina.copil) for _ in radacina.copil]

    for copil, alloc in zip(radacina.copil, alocare):
        for _ in range(alloc):
            nod = copil
            stare = nod.stare.clone()

            # Pasul 1: Selectie
            while not nod.mutareNetestata and nod.copil:
                nod = nod.celMaiBunCopil(weight=0.1)
                mutariLegale = stare.getMutariLegale(zar)
                if not mutariLegale:
                    break
                mutare = random.choice(mutariLegale)
                stare.aplicaMutare(*mutare)

            # Pasul 2: Expansiune
            if not stare.verificareFinal() and nod.mutareNetestata:
                nod = nod.expandare(zar)

            # Pasul 3: Evaluare euristică sau simulare pe mai multe mutări
            result = simulareNMutari(stare, zar, nrMutari)
```

```python
            # Pasul 4: Backpropagation
            while nod is not None:
                nod.update(result)
                nod = nod.parinte

    # Generăm statistici pentru mutările posibile
    detaliMutare = []
    mutariLegale = stareInitiala.getMutariLegale(zar)
    for i, copil in enumerate(radacina.copil):  # Adăugăm enumerate pentru indexare
        detaliMutare.append({
            'mutare': copil.mutare,
            'vizite': copil.vizite,
            'scorMediu': copil.scorMediu(),
            'index': i  # Indexul copilului în lista root.copil
        })

    if not detaliMutare:
        return None, 0

    # Sortăm mutările după scorul mediu și vizite pentru afișare
    detaliMutare.sort(key=lambda x: (x['scorMediu'], x['vizite']), reverse=True)

    # Afișăm statisticile pentru utilizator
    print("\nStatistici mutări posibile:")
    for stat in detaliMutare:
        print(f"Mutare: {stat['mutare']}, Vizite: {stat['vizite']}, Scor mediu: {stat['scorMediu']:.2f}")

    # Returnăm cea mai bună mutare și numărul de vizite
    ceaMaiBunaMiscare = detaliMutare[0]['mutare']
    nrViziteCeaMaiBunaMiscare = detaliMutare[0]['vizite']
    print(f"Mutarea selectată: {ceaMaiBunaMiscare} (Vizite: {nrViziteCeaMaiBunaMiscare}, Scor mediu: {detaliMutare[0]['scorMediu']:.2f})")
    return ceaMaiBunaMiscare, nrViziteCeaMaiBunaMiscare
```

Rol: Algoritmul Monte Carlo Tree Search pentru calcularea celei mai bune mutări.

Funcționalitate:

- Creează nodul rădăcină al arborelui.
- Execută procesul complet MCTS: selecție, expansiune, simulare și backpropagation.
- Alocă simulări mutărilor posibile și returnează cea mai bună mutare.

De ce e semnificativă? Este inima deciziilor AI-ului, utilizând simulări pentru a prezice cele mai bune mutări.

b. evaluareStare(stare)

```python
def evaluareStare(stare):
    """
    Evaluează starea curentă folosind reguli euristice și afișează contribuțiile la scor.
    """
    scorNod = 0

    for pozitie in range(24):
        # Blocarea unei zone între pozițiile 17 și 23
        if 17 <= pozitie <= 23 and stare.tabla[pozitie] * stare.playerCurent > 1:
            scorNod += 7  # Creștem importanța blocării zonelor

        # Scoaterea unei piese a adversarului
        if stare.tabla[pozitie] * stare.playerCurent == -1:
            scorNod += 6

        # Avansarea pieselor pentru jucătorul curent în zonele strategice
        if stare.playerCurent == -1 and 0 <= pozitie <= 6:
            scorNod += abs(stare.tabla[pozitie]) * 1.5
        elif stare.playerCurent == 1 and 18 <= pozitie <= 23:
            scorNod += abs(stare.tabla[pozitie]) * 1.5

        # Reducerea riscurilor prin păstrarea pieselor în siguranță (mai mult de 1 piesă pe poziție)
        if stare.tabla[pozitie] * stare.playerCurent > 1:
            scorNod += 3

        # Penalizare pentru piesele rămase singure pe tablă
        if stare.tabla[pozitie] * stare.playerCurent == 1:
            scorNod -= 4

        # Penalizare suplimentară pentru piesele care pot fi luate de adversar
        if stare.tabla[pozitie] * stare.playerCurent == 1 and stare.tabla[pozitie] * -stare.playerCurent == -1:
            scorNod -= 6

    # Bonus pentru scoaterea pieselor proprii
    pieseScoase = stare.scoase[stare.playerToIndex(stare.playerCurent)]
    scorNod += 12 * pieseScoase  # Creștem bonusul pentru piesele scoase

    return scorNod
```

Rol: Euristica pentru evaluarea stării jocului.

Funcționalitate:

- Atribuie punctaje pe baza poziției pieselor, blocajelor, scoaterii pieselor și riscurilor.
- Folosește reguli strategice pentru a favoriza mutările avantajoase.

De ce e semnificativă? Conduce deciziile AI-ului printr-un sistem bine definit de priorități.

5. Rezultatele obținute

a. Jucătorul umal câștigă:

În poza de mai jos se poate observa cum playerul (jucătorul uman, cu piese de culoare albă) a câștigat partida, acesta fiind primul care reușește să-și scoată cele 15 piese de pe tabla de joc, în timp ce AI-ul (reprezentat de piesele negre) a reușit să scoată doar 6.

Context înainte de ultima tură:

Jucătorul curent: Alb (1).

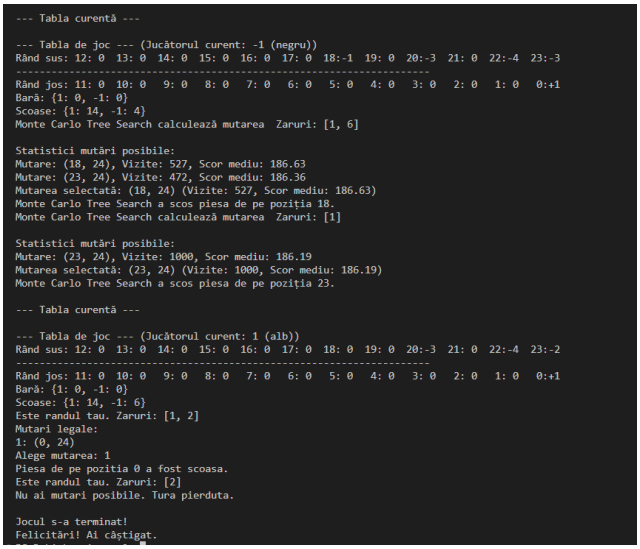
Stare tablă: Mai există o piesă albă pe poziția 0 (0: +1). Jucătorul alb are deja scoase 14 piese (Scoase: {1: 14, -1: 6}).

Zarurile disponibile sunt [1, 2].

Mutările jucătorului care au dus la victorie: Prima mutare: (0, 24) cu zarul 1 Acțiune: Piesa albă de pe poziția 0 este scoasă folosind zarul 1.

Actualizare tablă: Poziția 0 devine goală (0: 0). Numărul pieselor scoase de jucătorul alb ajunge la 15 (Scoase: {1: 15, -1: 6}).

A doua mutare: Zarul 2 Acțiune: După scoaterea piesei, nu mai există alte piese albe pe tablă. Deoarece nu poate efectua alte mutări, tura este considerată pierdută. Actualizare: Jocul se încheie automat, deoarece toate piesele jucătorului au fost scoase. Condiția de câștig: Conform regulilor jocului de table, un jucător câștigă dacă scoate toate cele 15 piese de pe tablă. După scoaterea ultimei tale piese, condiția a fost îndeplinită. Rezultatul final: Mesajul jocului: „Jocul s-a terminat! Felicitări! Ai câștigat.”

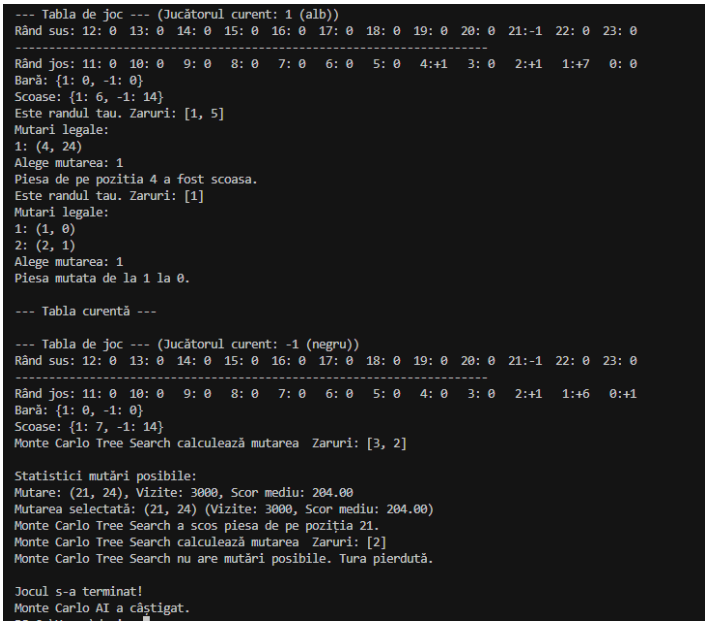


b. Jocul este câștigat de AI:

În acest caz se poate observa cum AI-ul câștigă partida cu un punctaj de 15 la 7.

Contextul jocului înainte de mutarea AI:
Jucătorul curent: Negru (-1).
Stare tablă: Pe poziția 21 există o piesă neagră (21: -1). Pe bara jucătorului alb nu există piese (Bară: {1: 0, -1: 0}).
AI-ul are deja 14 piese scoase (Scoase: {1: 7, -1: 14}).
Zarurile disponibile: [3, 2].

Prima mutare AI: (21, 24) Acțiune: AI-ul a scos piesa de pe poziția 21 folosind zarurile [3, 2].
Zarurile disponibile permit scoaterea piesei: Zarul 3 aduce piesa la poziția 24 (ieșire).
Actualizare tablă: Poziția 21 devine goală (21: 0). Numărul pieselor scoase de AI crește la 15 (Scoase: {1: 7, -1: 15}).
De ce AI-ul nu a avut mutări posibile după această mutare? După ce toate piesele unui jucător sunt scoase, jocul se termină automat. În această situație: AI-ul nu mai avea piese pe tablă pentru a efectua alte mutări. Astfel, tura sa s-a încheiat fără alte acțiuni.
Rezultatul final: Mesajul jocului: „Jocul s-a terminat! Monte Carlo AI a câștigat.”



6. Concluzii

- În concluzie acest AI nu este imbatabil, putând fi învins de către jucători umani, dar este o soluție pentru a implementa un joc de Backgammon (Table).
- Stilul de joc al lui MCTS poate fi configurata dupa preferințe prin modificarea numărului de puncte asignat fiecărui tip de mutare (adus piese în casă, scos piese, blocat poziții etc.).
- În funcție de dificultatea dorită numărul de ture și de simulări privite în avans de AI pot fi modificate.

7. Bibliografie

http://joci.ro/reguli/table

https://www.researchgate.net/publication/228378473_Monte-Carlo_tree_search_in_backgammon

https://en.wikipedia.org/wiki/Monte_Carlo_tree_search

https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/

8. Contribuția membrilor echipei

Ilioi Daniel: implementare MCTS, integrarea algoritmului cu regulile testare și colectare rezultate

Maxim Rareș-Constantin: implementare reguli, testare și colectare rezultate, documentație