UNIVERSITATEA TEHNICĂ „Gheorghe Asachi” din IAȘI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DOMENIUL: Calculatoare și Tehnologia Informației
SPECIALIZAREA: Tehnologia Informației

METODE DE INTELIGENŢĂ ARTIFICIALĂ PENTRU JOCUL DE ŞAH

LUCRARE DE DIPLOMĂ

Coordonator științific:
prof. dr. ing. Florin Leon

Absolvent:
Tudor-Matei Chiteală

Iași, 2022

DECLARAȚIE DE ASUMARE A AUTENTICITĂȚII LUCRĂRII DE DIPLOMĂ

Subsemnatul(a) CHITEALĂ TUDOR MATEI,
legitimat(ă) cu CI seria MZ nr. 963576, CNP 1990514226774
autorul lucrării METODE DE INTELIGENȚĂ ARTIFICIALĂ PENTRU JOCUL DE ȘAH

elaborată în vederea susținerii examenului de finalizare a studiilor de licență organizat de către Facultatea de Automatică și Calculatoare din cadrul Universității Tehnice „Gheorghe Asachi" din Iași, sesiunea IUNIE-IULIE a anului universitar 2022, luând în considerare conținutul Art. 34 din Codul de etică universitară al Universității Tehnice „Gheorghe Asachi" din Iași (Manualul Procedurilor, UTI.POM.02 – Funcționarea Comisiei de etică universitară), declar pe proprie răspundere, că această lucrare este rezultatul propriei activități intelectuale, nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislației române (legea 8/1996) și a convențiilor internaționale privind drepturile de autor.

Data
08.07.2022

Semnătura

Cuprins

METODE DE INTELIGENŢĂ ARTIFICIALĂ PENTRU JOCUL DE ŞAH

Tudor-Matei Chiteală

Rezumat

Lucrarea implică dezvoltarea unei aplicaţii software ce foloseşte metode de inteligenţă artificială pentru jocul de şah. Motoarele de şah sunt în continuă dezvoltare, acestea abordând diferite căi pentru rezolvarea jocului.

Obiectivul lucării este de a compara algoritmii Monte Carlo şi retezarea alfa-beta în diferite situaţii. Pentru a realiza acest aspect, este necesără implementarea logicii jocului de şah si a algoritmilor împreună cu tabela de transpoziţie.

Prin intermediul limbajului Python este dezvoltată componenta logică. Stocarea tabelei de transpoziţie a necesitat folosirea sistemului de management al bazelor de date MongoDB. Pentru generarea cheii unice în cadrul tabelei este utilizată operaţia de hashing Zobrist.

Interfaţa cu utilizatorul este dezvoltată în Python, folosind modulul PyGame. Rolul este de a expune funcţionalităţile într-un mod simplu şi uşor de folosit. Prin intermediul interfeţei se pot realiza mutările, selectarea dificultăţii, modul de joc şi algoritmul folosit.

Pentru realizarea aplicaţiei a fost necesară documentarea asupra strategiilor abordării jocului de şah, astfel încăt rezultatul returnat de algoritm să fie mai precis.

Introducere

Jocul de şah este un joc abstract de strategie, unde mulţi istorici consideră că-şi are originea in India. Starea curentă a jocului de şah apare în a doua jumătate a secolului 15 [1]. Acesta este jucat pe o tabla de 8x8 intre 2 jucatori în care fiecare controleaza 16 piese: un rege, o regină, două ture, doi nebuni, doi cai si opt pioni. Scopul jocului este de a da şah mat adversarului, unde regele este atacat si acesta nu are nici o cale de scăpare.

Fiecare piesa are un mod unic de a fi mutat ce conduce ca fiecare să aibă un anumit punctaj. Deoarece şahul este un joc dinamic, anumite piese pot fi mai utile în anumite contexte.

Unul dintre obiectivele programatorilor a fost construirea unei maşini care poate juca şah la un nivel înalt. În 1951, inventatorul Dietrich Prinz a creat primul program de şah, folosind Ferranti Mark 1. Calculatorul nu avea puterea necesară pentru a juca un meci complet, astfel inventatorul a limitat programul doar dacă sah mat putea fi găsit în doua mutări [2].

În anul 1957, apare primul program de şah, implementat de cercetătorul de la IBM Alex Bernstein, ce putea rula un joc complet de şah. Mutarea unei piese dura aproximativ opt minute[3].

Acest ţel a fost atins în 1997, atunci când Deep Blue, sistemul construit de IBM, l-a învins pe Garry Kasparov, care era în acel moment campionul mondial.

Deep Blue este succesorul motoarelor de şah Chiptest şi Deep Thought. Acesta a făcut istorie în 1996 atunci când a câştigat un meci împotriva lui Garry Kasparov din şase. În 1997, în meciul de revanşă, Deep Blue a câştigat decisiv cu scorul de 3.5-2.5, având 2 meciuri câştigate şi 3 egaluri[4].

Deep Blue a folosit cautarea alpha-beta folosind paralelismul ce conduce la o căutare mai rapidă. Acesta a folosit o carte pentru deschideri de aproximativ 4000 de pozitii. Faţă de motoarele de şah actuale, Deep Blue s-a bazat pe forţă brută computaţională, putând evalua 200 milioane de poziţii pe secundă.

Pe parcusul istoriei, au apărut diferite motoare de şah precum Stockfish şi AlphaZero.

Stockfish este un motor de șah gratuit cu sursa deschisă. Acesta a câştigat campionatul de motoare de șah de top de 12 ori[5]. Acesta foloseşte o variantă a algoritmului retezarea alpha-beta.

AlphaZero a fost dezvoltat de compania de cercetare DeepMind pentru a rezolva jocuri precum şah, shogi sau go. În 2017, echipa DeepMind anunţă că AlphaZero doar în 24 de ore acesta a ajuns la un nivel de mare maestru, învingându-l pe campionul mondial Stockfish[6].

AlphaZero foloseşte o variantă a algoritmului Monte Carlo împreună cu o metodă de întărire profundă.

La momentul actual, cel mai bun motor de şah este Stockfish cu 3529 ELO dar nu la o diferenţă foarte mare de AlphaZero cu 3460 ELO.Aceste motoare de şah folosesc în medie 512 procesoare si cu o tabelă de transpoziţie de aproximativ 32 TB.

Tabela de transpoziţie este un cache pentru poziţiile anterior văzute, în care este stocat rezultatul, intr-un joc precum GOul. Aceasta este folosită prin analizarea a milioanelor de poziţii diferite, prin generarea de hash-uri ce codifică poziţia pieselor pe tabla de joc[7].

O metodă pentru operaţia de hashing este Zobrist[8]. Dezavantajul este că se poate produce o coliziune ce ar putea rezulta într-un raspuns greşit. Totuşi aceasta coliziune scade cu cât numarul de biţi creşte, la 64 biţi coliziunea se poate produce dupa 4 miliarde de poziţii.

Alt dezavantaj este numarul mare de spaţiu alocat, unde programe precum Stockfish folosesc aproximativ 32 TB.

Majoritatea motoarelor de şah folosesc deschideri şi matrici pentru calculul poziţional pentru a obţine un rezultat corect.

Obiectivul lucrării este realizarea unei aplicaţii în care utilizatorul poate alege cu ce motor de şah doreşte să joace. Aplicaţia conţine un motor de şah care foloseşte căutarea Monte Carlo şi alt

motor de şah ce implementează retezarea alpha-beta. Utilizatorul poate să selecteze dificultatea si motorul. Dificultatea se setează prin adăugarea adâncimii maxime la care poate ajunge programul şi pentru Monte Carlo numărul de iterări. În plus se setează si numarul maxim de minute pentru procesarea mişcării calculatorului dacă utilizatorul doreşte.

În urma dezvoltării acestui proiect se realizează o statistică. Aceasta constă in analizarea celori două motoare de şah. Motoarele de şah vor fi puse sa joace unul contra celuilalt, în care anumiţi parametrii vor fi schimbaţi precum adăncimea, numarul de iteratii şi daca motorul de căutare poate folosi tabela de transpoziţie.

Capitolul 1. Fundamentarea teoretică și documentarea bibliografică

1.1. Domeniul şi contextul abordării temei

Folosirea inteligenţei artificiale în rezolvarea jocurilor complexe precum şahul este în continuă dezvoltare. Jocul de şah având o multitudine de posibile mutări, face imposibilă parcurgerea jocului complet şi returnarea unei mutări "corecte". Această problemă este rezolvată de algoritmi de căutare precum retezarea alpha-beta şi Monte Carlo.

S-au efectuat numeroase jocuri pentru antrenarea motoarelor de şah, în anul 1997 Deep Blue devine primul motor ce reuşeşte să învingă un campion mondial la şah.

Evoluţia motoarelor de şah poate fi identificată cu usurinţă prin compararea ELOului Deep Blue de 2853 şi cel mai bun motor de şah la ora actuală Stockfish cu 3529.

În cadrul lucrării, accentul este pus pe implementarea algoritmilor retezarea alpha-beta şi monte carlo, fiind îmbunătăţite prin folosirea unor deschideri, tabelă de transpoziţie şi utilizarea teoriei de şah pentru a efectua un calcul corect asupra poziţiei.

1.1.1. Jocul de şah

Şahul este un joc complex de strategie folosind o tabla de joc 8x8 cu 32 de piese. Fiecare jucător are 16 piese, acestea fiind: un rege, o regină, două ture, doi cai şi opt pioni. Piesele sunt aşezate ca în figura (Figura 1.1). Scopul final este ca un jucător să dea şah mat, regele este atacat şi nu poate fi evitat atacul printr-o mişcare validă.

Figura 1.1. Tabla de şah[1]

1.1.2. Piesele jocului de şah

- **Pionul**

  Se poate muta doar inainte cate o casuţă exceptând prima mutare când acesta poate parcurge doua celule.

  Piesa poate captura doar pe diagonală(doar în faţă) şi acesta poate executa mişcarea specială numită En passant. En passant se produce imediat după ce un pion a fost mutat 2 casuţe păna in dreptul pionului oponent.Poziţia finală este identică cu capturarea normală a pionului. Dacă mişcarea specială nu se produce imediat, se pierde dreptul de a o mai face.

[1] https://thechessworld.com/

Pionul ajuns la capătul tablei, acesta poate să fie schimbat cu o altă piesă, în cele mai multe cazuri acesta este schimbat cu regina.

Valoarea = 1

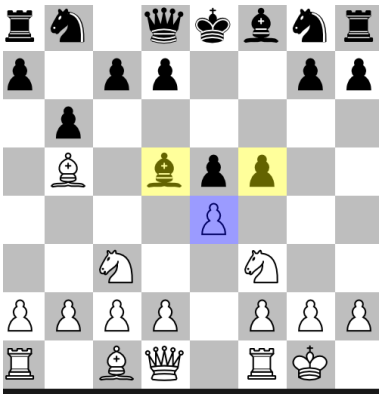

Figura 1.2. Mutări pion[2]

- **Calul**

Poate muta doar în "formă de L". Prin această mişcare specială, calul este singura piesă care poate sări peste alte piese.

Valoarea = 3

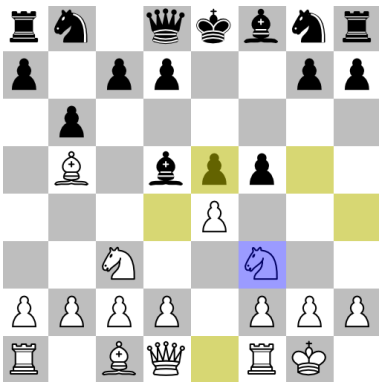

Figura 1.3. Mutări cal[3]

- **Nebunul**

Poate executa mişcări/captura doar pe diagonală si acesta nu-şi poate schimba culoarea casuţei.

Valoarea = 3

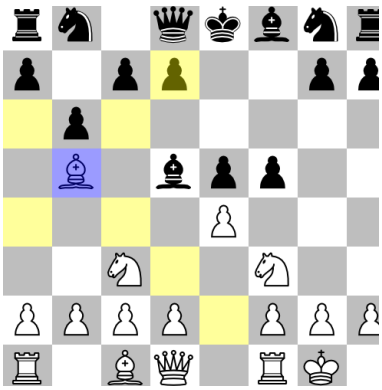

Figura 1.4. Mutări nebun[4]

- **Tura**

  Poate executa mişcări/captura doar pe orizontală sau verticală.

  Valoarea = 5

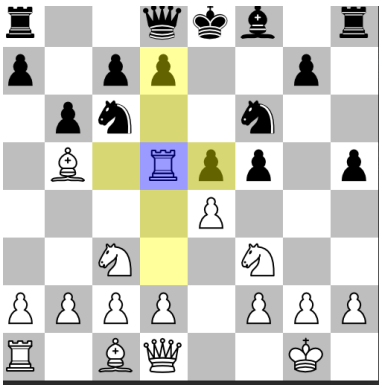

Figura 1.5. Mutări tură[5]

- **Regina**

  Considerată cea mai puternică piesă aceasta se poate mişca în toate cele 8 direcţii si pe orice distanţă.

  Valoarea = 9

Figura 1.6. Mutări regină[6]

- **Regele** Cea mai importantă piesă, poate muta/captura doar o casuţă în cele 8 direcţii.

  Regele poate efectua rocada, schimbarea locului cu tura doar dacă regele/tura nu au fost mutate până în acel moment, spaţiul dintre cele două piese este liber şi nu este atacat de o piesă a oponentului.

  Valoarea = ∞

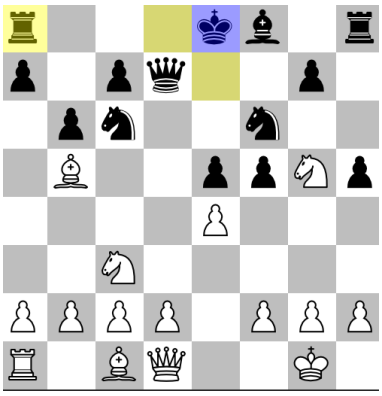

Figura 1.7. Mutări rege[7]

1.1.3. Arborele jocului

În teoria jocurilor arborele este o structură formată din noduri (Figura 1.8) ce reprezintă toate stările posibile ale jocului, unde se porneşte secvenţial de la nodul rădăcină, nodul 0, spre frunze, frunzele reprezentând starea finală a jocului[9]. Şahul fiind un joc complex se folosesc arbori parţiali, din cauza numarului mare de posibilităţi, care fac fezabil calculul pentru calculatoarele moderne.

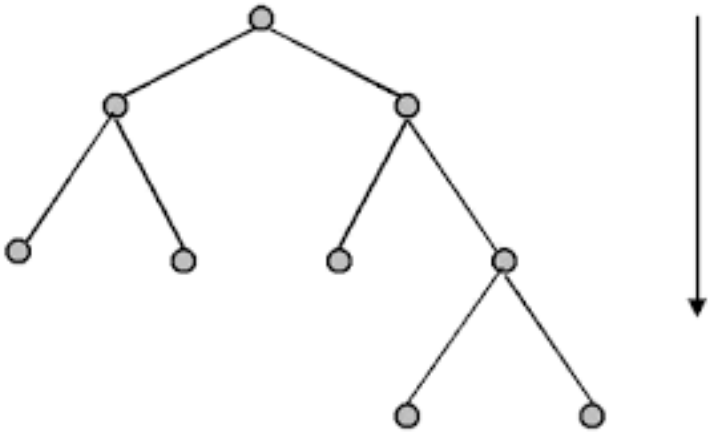

Figura 1.8. Arborele jocului[8]

1.1.4. Scorul

Scorul este un factor important în repezentarea corectă a stării jocului. Acesta specifică cine câştigă intr-o anumită stare şi cine pierde. În jocul de şah, calculul numarului de piese nu este suficient, acest tip de calcul conduce algoritmul în a considera ca anumite mutări au aceeaşi valoare. Un exemplu ar fi la deschidere, când în primele 3-5 mişcari nu se fac capturări, algoritmul considerând că jocul încă este la egalitate. Dar modul corect de abordare ar fi de a controla

[8] https://stanford.library.sydney.edu.au/archives/fall2003/entries/game-theory/

centrul tablei de joc. Din această cauză se folosesc matrici pentru calculul poziţional sau carţi de deschidere.

1.1.4.1. Matrice pentru calculul pozitional

Matricea pentru calculul poziţional, precum cea de mai jos, vine în ajutorul algoritmilor pentru o reprezentare mai bună a stării actuale. Matricea acordă puncte în plus pieselor dacă sunt într-o poziţie favorabilă. De exemplu calul este mai eficient si mai periculos în centrul tablei de joc. Acestă poziţie oferă piesei 8 direcţii de atac. Faţă de marginea tablei de joc, unde calul nu este la fel de eficient, acesta având doar 4 direcţii de atac disponibile.

$$\begin{bmatrix} -50 & -40 & -30 & -30 & -30 & -30 & -40 & -50 \\ -40 & -20 & 0 & 0 & 0 & 0 & -20 & -40 \\ -30 & 0 & 10 & 15 & 15 & 10 & 0 & -30 \\ -30 & 5 & 15 & 20 & 20 & 15 & 5 & -30 \\ -30 & 0 & 15 & 20 & 20 & 15 & 0 & -40 \\ -30 & 5 & 10 & 15 & 15 & 10 & 5 & -30 \\ -40 & -20 & 0 & 5 & 5 & 0 & -20 & -40 \\ -50 & -40 & -30 & -30 & -30 & -30 & -40 & -50 \end{bmatrix}$$

1.1.4.2. Deschideri

Motoarele de obicei folosesc la începutul jocului cartea de deschidere. Baza de date poate fi folosită cât timp mişcările au mai fost jucate. După ce o mutare noua a fost jucată, motorul de şah foloseşte algoritmul de căutare. În baza de date sunt stocate mutări în format PGN(Portable Game Notation), unde este specificat ce piesă a fost mutată si în ce locaţie pe tabla de joc. De exemplu deschiderea Ruy Lopez(Figura 1.9) poate fi transcrisă in formatul: 1. e4 e5 2. Nf3 Nc6 3. Bb6 a6

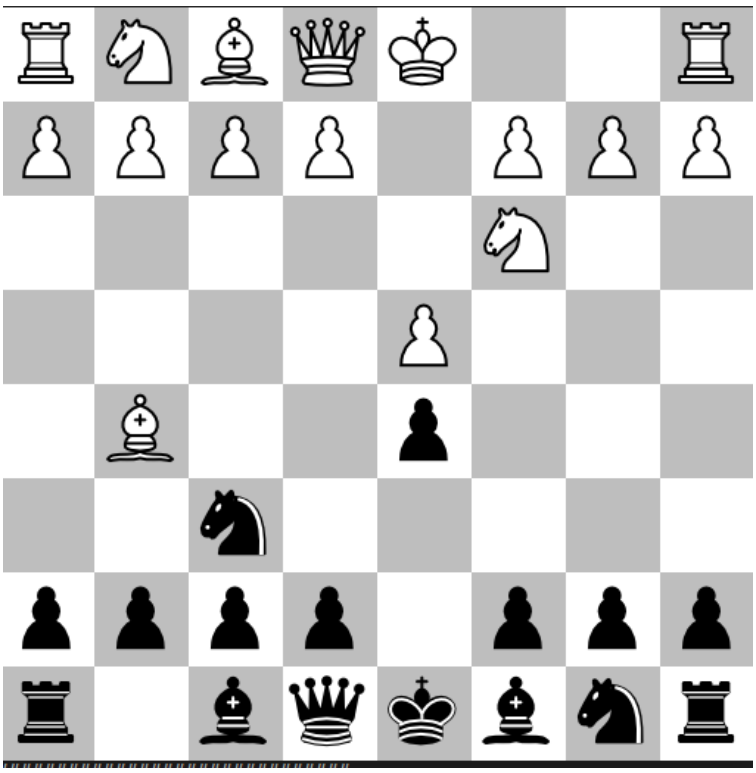

Figura 1.9. Deschiderea Ruy Lopez[9]

1.1.5. Algoritmul minimax

Algoritmul minimax este un algoritm de decizie recursiv folosit în inteligenţa artificială şi teoria jocurilor. Minimax pleacă cu premiza că oponentul încearcă să câştige(joaca optim). Dacă nu este respectată această condiţie, poate să afecteze eficacitatea algoritmului. Altă ipoteza ar fi că jocul trebuie să fie strategic şi să nu incorporeze componente pentru noroc[10].

Există două tipuri de jucători, cel care maximizează(MAX) şi cel care minimizează(MIN). Scopul jucătorului MAX este de a maximiza valoarea nodului şi a jucătorului MIN de a o minimiza (Figura 1.10). Deoarece numarul de mutări în şah este mare, frunzele sunt de obicei alese atunci când adâncimea maximă a fost atinsă. Paşii algoritmului ar fi:

1. Se construieşte arborele jocului până la o anumită adâncime
2. Evaluare scorului pentru fiecare frunză
3. Propagarea înapoi de la frunze spre rădacină:
   (a) Pentru MAX se selectează maximele
   (b) Pentru MIN se selectează minimile
4. La nodul rădăcină se alege scorul maxim, reprezentând cea mai bună mutare.

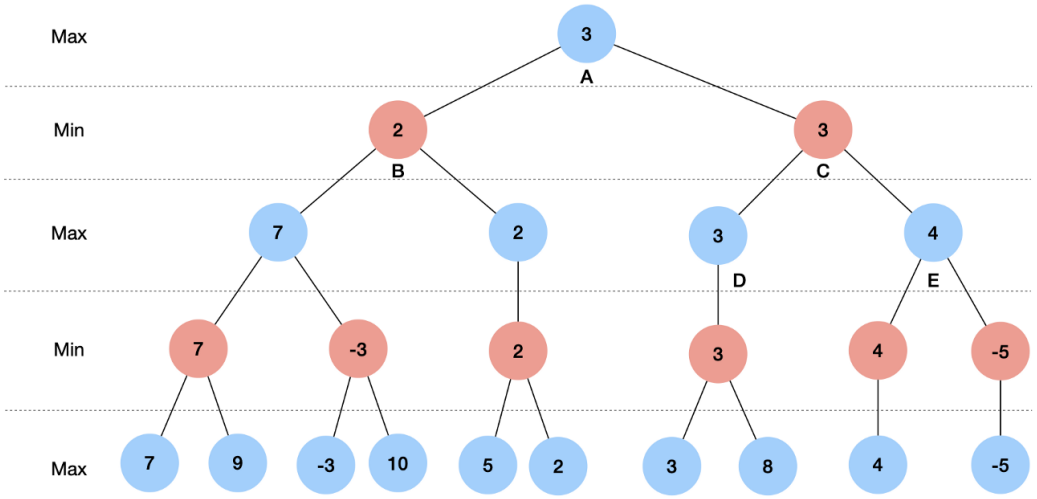

Figura 1.10. Algoritmul minimax[10]

1.1.5.1. Algoritmul retezarea alpha-beta

Algoritmul retezarea alpha-beta(Figura 1.11) vine în ajutorul algoritmului minimax pentru a minimiza numărul de noduri evaluate. Acesta opreşte evaluarea nodului în 2 cazuri[10]:

1. Dacă valoarea alfa este mai mare sau egală decât valoarea beta a unui nod descendent, atunci se opreşte generarea fiilor nodului descendent.
2. Dacă valoarea beta este mai mică sau egală decât valoarea alfa a unui nod descendent, atunci se oprește generarea fiilor nodului descendent.

[10] https://towardsdatascience.com/

Algoritmul returnează aceeaşi valoare cu algoritmul minimax doar că evită calculul anumior noduri.

Algoritmul foloseşte variabila alfa pentru a păstra cea mai bună valoare pe care o are MAX, iar beta pentru MIN.

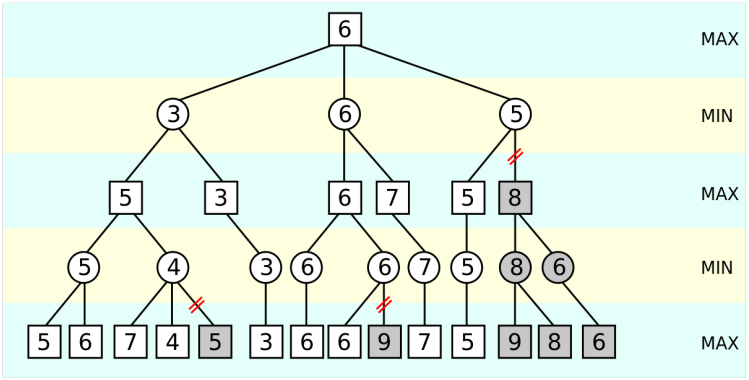

Figura 1.11. Algoritmul retezarea alpha-beta[11]

1.1.6. Algoritmul Monte Carlo

Algoritmul Monte Carlo este un algoritm de decizie recursiv folosit în jocuri de strategie. A fost folosit în 2016 cu o reţea neuronală pentru AlphaGo.

AlphaGp a folosit Monte Carlo pentru a aduna date de antrenare pentru reţeaua neuronală, evitand sa mai înveţe din jocurile oamenilor. Acesta foloseşte doar regulile jocului şi foloseşte o reţea cu două capete, care aproximează politicile[11]. pentru a invata.

MCTS pune accentul pe analiza celor mai promiţătoare mutări, extinzând arborele de căutare bazat pe eşantionarea aleatorie[10]. Rezultatul este folosit pentru a pondera nodurile astfel încat nodurile mai bune sunt folosite în rundele următoare.

Algorimul MCTS este format din 4 paşi: Selecţia, Expandarea, Simularea, Propagarea înapoi.

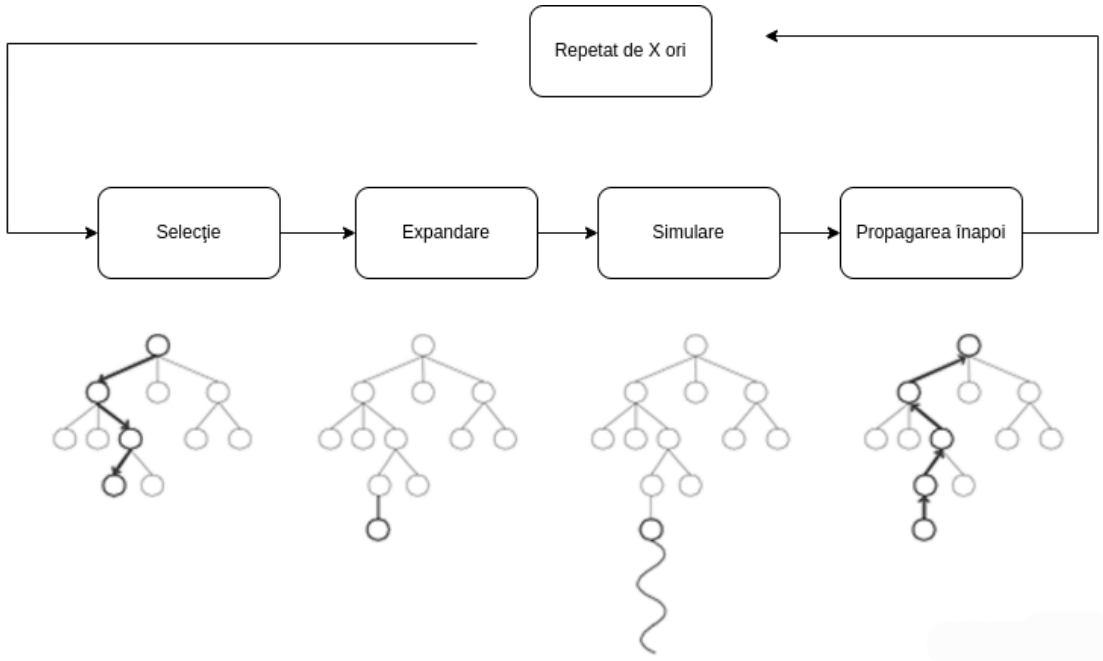

Figura 1.12. Algoritmul Monte Carlo[12]

[11] https://en.wikipedia.org/wiki/

1.1.6.1. Selecţia

Pornind de la nodul rădăcină, selectăm succesiv nodurile folosind o strategie specifică până ajungem la frunză. Strategia este folosită pentru a selecta optim valoarea cea mai bună. MCTS foloseşte Limita superioară de încredere(Upper Confidence Bound(UCB)). Fiecare nod conţine numărul de victorii şi numarul de selecţii.

$$UCB1 = \frac{w_i}{n_i} + C * \sqrt{\frac{\log N}{n_i}} \tag{1.1}$$

- w_i - numărul de victorii al nodului fiu i
- n_i - numărul de simulări în nodul fiu i
- N - numărul de simulări în nodul curent
- C - contantă folosită pentru ponderarea explorării şi exploatării

1.1.6.2. Expandarea

Expandarea se execută atunci când selecţia nu mai este posibilă.

Cât timp jocul nu se termină decisiv în frunză gasită, în etapa de selecţie, creăm un nod nou sau mai multe.

Sunt selectate în mod aleatoriu un nod frunză nevizitat, deoarece pentru nodurile nevizitate UCB1 este infinit.

1.1.6.3. Simularea

În acest proces, o simulare este executată până ajungem la un rezultat. Se aleg mutări aleatorii până se atinge o stare terminală, victorie sau înfrângere.

În locul căutării pur aleatorii, se pot folosi euristici care să aleagă mutări mai bune.[curs IA].

1.1.6.4. Propagarea înapoi

După aflarea rezultatului a nodului nou creat, actualizăm informaţiile de la frunză până la nodul rădăcină.

Nodurile parcurse în etapa de simulare nu se actualizează, doar cele de la nodul selectat în sus.

1.1.7. Alegerea mutării

După aplicarea algoritmului, se alege mutarea cu cel mai mare număr de selecţii, deoarece valoarea sa este cel mai bine estimată[curs IA]

Dupa ce se execută mutări de ambele părţi, se pot refolosi valorile subarborelui.

1.1.8. Tabela de transpozitie

Tabela de transpoziţie este un cache pentru poziţiile anterior calculate, folosit adesea în jocul de şah. O poziţie poate fi atinsă în mai multe secvenţe de mutări. Dacă poziţia a fost găsită în baza de date, calculul nu mai trebuie făcut, rezultând în obţinerea mutării "mai bune" într-un timp mai bun.

1.1.8.1. Operaţia de hashing Zobrist

Operaţia de hashing Zobrist este funcţia de hashing folosită în jocurile de strategie, pentru a implementa tabela de transpoziţie astfel încât să evităm analizarea aceleaşi poziţii de mai multe ori.

Hashuirea începe prin generarea aleatorie a numerelor de 64 bits pentru fiecare element posibil. Fiecare căsuţa are 12 numere reprezentând 6 piese diferite ale albului şi 6 ale negrului (Figura 1.13). După generarea numerelor pentru fiecare casuţa se efectuează funcţia XOR (Figura 1.14) pentru a forma cheia unică Zobrist.

Figura 1.13. Numere generate aleatoriu pentru fiecare celulă[13]

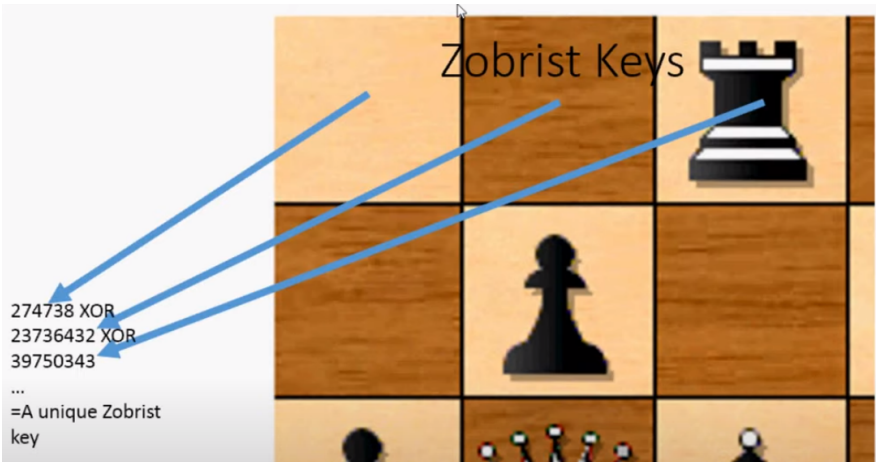

Figura 1.14. Cheia unică zobrist[14]

[14] https://youtu.be/QYNRvMolN20?t=353

1.1.8.2. Datele stocate în tabela de transpoziţie

În baza de date sunt stacate date despre adâncime, mutarea făcută şi scorul poziţiei identificate prin cheia unică. Adâncimea ne specifică la ce nivel am obţinut scorul. Dacă în timpul algoritmului avem aceeaşi stare dar suntem la un nivel mai mare ca cel din baza de date, nu vom folosi rezultatul din tabela de transpoziţie. Doar dacă nivelul din algoritm este mai mic atunci vom folosi scorul din baza de date. Nivelul mai ridicat ne arată că rezultatul este mai util faţă de cel cu adâncimea mai mică.

```
{
   8544513627688316925:{
   mutare: Rc7,
   adâncime: 6,
   scor: 230
},
}
```

1.2. Calcularea scorului

Pe parcursul istoriei jocului de şah au existat diferite teorii pentru a identifica dacă o poziţie este în avantajul jucătorului.

1.2.1. Structura pionilor

1.3. Tema propusă

Tema propusă în această lucrare constă în aplicarea algoritmilor monte carlo şi retezarea alpha-beta. În ajutorul acestor algoritmi vine tabela de transpoziţie şi deschideri care scad din numărul de calcule. Acesta vine în sprijinul jucătorilor de şah care doresc să-şi gasească un adversar care este de nivelul acestora.

Scopul acestei lucrări nu este doar pentru a juca împotriva calculatorului. În acelaşi timp putem verifica care algoritm este mai eficient în anumite cazuri de joc.

A fost propusă această lucrare deoarece inteligenţa artificială aplicată în jocul de şah este o temă de actualitate unde de la an la an se fac descoperiri şi moduri inedite pentru rezolvarea jocului.

1.4. Lucrări similare. Analiza de aplicaţii existente

Până în prezent, au existat numeroase programe, folosind diferiţi algoritmi pentru a rezolva jocul de şah.

Deep Blue a fost primul motor de şah care a câştigat împotriva unui campion mondial, acela fiind Garry Kasparov, considerat fiind cel mai bun jucător de şah din toate timpurile. Motorul de şah, construit de IBM, a folosit algoritmul alpha-beta, împreună cu partea hardware contruită de IBM, Deep Blue putea analiza 200 milioane de poziţii pe secundă.

Funcţia de evaluare a lui Deep Blue a fost iniţial scrisă într-o formă generalizată cu mulţi parametrii care trebuie determinaţi. Acesta a fost scris în limbajul C. Valoarea parametrilor a fost determinată prin analiza jocurilor de şah[4]. Motorul de şah a folosit o carte de deschidere de aproximativ 4000 de poziţii.

Pentru partea hardware, motorul a folosit cipuri VLSI (Figura 1.15) având în folosinţă 30 PowerPc 604e procesoare[12].

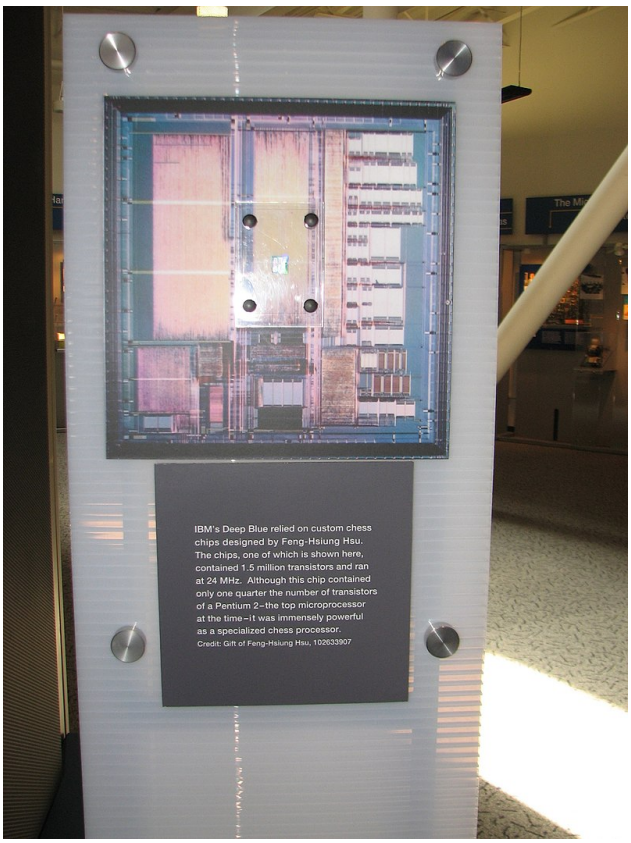

Figura 1.15. Chip Deep Blue[15]

Stockfish este un motor de şah gratis şi cu sursă deschisă. La momentul actual, Stockfish este cel mai puternic motor de şah, fiind estimat la un ELO de 3529.

Acesta utilizează 512 procesoare şi o tabelă de transpoziţie de 32 TB. La fel ca şi Deep Blue, Stockfish aplică o versiune de căutare alpha-beta. Comparativ cu alte motoare, acesta produce o retezare mai agresivă. Stockfish ajunge la adâncimi de 40, faţă de Deep Blue care ajunge în medie 6-8[13].

AlphaZero este un program dezvoltat de DeepMind pentru a rezolva jocuri precum şah, shogi sau go. Faţă de motoarele de şah Deep Blue şi Stockfish, AlphaZero foloseşte algoritmul de căutare Monte-Carlo. Pentru fiecare mişcare, programul caută doar o fracţiunile din poziţiile posibile. Stockfish are căutări de 60 milioane (Figura 1.16), comparativ cu AlphaZero 60 mii[6].

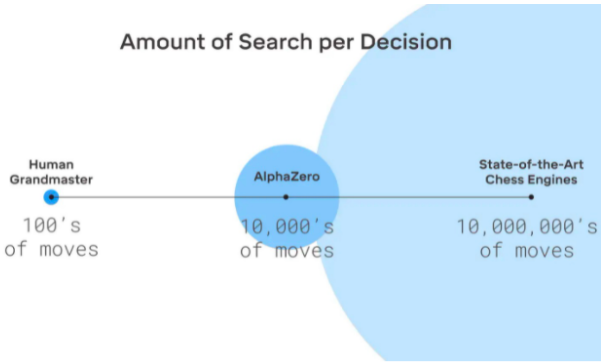

Figura 1.16. Comparaţie AlphaZero[16]

[15] https://www.flickr.com/
[16] https://www.deepmind.com/

Altă diferenţă dintre cele doua este că AlphaZero nu se foloseşte euristici şi reguli contruite de jucatorii de şah. A fost înlocuit acest lucru cu o reţeau neuronală şi algoritmi precum Monte-Carlo, folosind doar cunoştiţe despre regulile de bază ale jocului.

Capitolul 2. Proiectarea aplicației

2.1. Proiectarea software a aplicaţiei

Aplicaţia prezentată in acest document este oferită utilizatorilor sub forma unei aplicaţii desktop unde utilizatorul poate selecta să joace contra altui utilizator sau contra calculatorului.

Implementarea aplicaţiei a fost realizată folosind Visual Studio Code cu limbajul de programare Python 3.10 care pune la dispoziţie o varietate de module utile in lucrul cu matrici, procesarea imaginilor si dezvoltarea interfeţei cu utilizatorul.

2.1.1. Python

Python este un limbaj de programare de nivel înalt, orientat obiect şi multi-paradigmă. Proiectarea limbajului pune accent pe o înţelegere mai uşoara şi rapidă a codului.

Limbajul foloseşte garbage-collector, prin intermediul caruia acesta redobândeşte memoria care nu mai este folosită. O altă caracteristică importantă este declararea variabilelor, aceasta nu este fixată la începutul declarării ci este determinat de interpretor.

Programul Python este interpetat, acesta este executat linie cu linie şi nu tot în acelaşi timp. Nu trebuie compilat deoarece este procesat la runtime de interpretor.

Limbajul este unul portabil, astfel încât codul sursă poate fi folosit pe diferite maşini fără a efectua difertie modificări.

Un factor important în alegerea limbajul Python a fost suportul pentru interfaţa grafică cu utilizator. Python poate fi folosit uşor cu Tkinter, JPython şi un modul specific Pygame.

Python dispune de o multitudine de biblioteci, acesta fiind descris şi ca un limbaj cu "batteries included". Datorită bibliotecilor, limbajul este folosit deseori pentru proiecte de inteligenţă artificială, învăţare automată şi securitate cibernetică.

Am folosit acest limbaj deoarece dispune de o varietate mare de biblioteci, uşor de utilizat pentru implementarea unei interfeţe grafice prin bibilioteca PyGame şi accesarea rapidă la sistem de management al bazelor de date MongoDB, oferind totodată simplitatea scrierii codului.

Module utilizate în cadrul aplicației sunt:

- Pygame - un set de module utile pentru a construi un joc in Python[14]
- PygameMenu - set de module pentru implementarea meniului
- Numpy - pentru lucrul cu vectori si matrici[15]
- MongoEngine - folosit pentru a lucra cu sistem de management al bazelor de date MongoDB[16]
- json - folosit pentru a extrage sau încărca documente în baza de date

2.1.2. MongoDB

MongoDB este un sistem de management al bazelor de date, multi-platformă orientată pe documente care stochează datele sub formă de cheie-valoare. Acesta foloseşte documente de tip JSON, util în lucrul cu o multitudine de date distribuite şi aplicaţii web. Sistem de management suportă o varietate mare de tipuri de date: numere, vectori, hash.

Caracteristicile bazei de date MongoDB sunt[17]:

- Baza de date fară schemă: O colecţie poate stoca diferite tipuri de documente, fără ca acestea să fie la fel faţă de bazele de date SQL.
- Scalabilitatea: MongoDB oferă scalabilitate orizontală. Acest lucru este realizat prin partajarea datelor pe mai multe servere.

- Ad hoc: Suport pentru interogări ad hoc.
- Indexarea: Permite gasirea documentelor fară a scana toata baza de date.
- Replicarea: Un master poate efectua citiri și scrieri, iar un slave copiază datele de la master și poate fi folosit doar pentru citiri sau copii de rezervă.[mongodb feature]
- Eficienţă: Operaţiile sunt mai rapide ca intr-o baza de date relaţională
- Agregarea: Permite efectuarea de operații asupra datelor grupate și obținerea unui singur rezultat
- Conţine date eterogene

Dezavantajele MongoDB sunt:

- Foloseşte multă memorie pentru stocarea datelor.
- Există o limită de 16MB pentru documente
- Nu este suportată operaţia de tranzacţie.

Exemplu de document păstrat în baza de date pentru cheile Zobrist:

```
{
     "_id": {
        "$oid": "62b46989b23cdb603394bbc7"
    },
    "row": 0,
    "cols": [
        [
            {
                "$numberLong": "7968574837187944547",
                ...
            },
                ...
        ],
        ...
}
```

Am ales să folosesc sistemul de management MongoDB deoarece suportă tipuri de date vectori şi hash. Aceste tipuri de date sunt utile pentru implementarea tabelei de transpoziţie.

În acelaşi timp, modulul mongoengine face accesarea bazei de date simplă.

2.2. Descrierea modulelor principale ale aplicaţiei

În cadrul programului avem 7 module scrise în limbajul Python. Modulul Piece ce reţine informaţii despre piese, modulul Move ce conţine informaţii necesare despre modul în care a fost mutată piesa, modulul Score folosit pentru evaluarea la finalul algoritmului, modulele monteCarlo si alphaBeta conţinând algoritmii pentru inteligenţa artificială, modulul Zobrist ce este folosit pentru tabela de transpoziţie, alt modul este Board ce reţine referinţele pieselor şi diferite funcţii pentru a respecta regulile jocului şi ultimul modul principal este Gui ce ofera interfaţa cu utilizatorul prin intermediul PyGame.

2.2.1. Modulul Pieces

Piece conţine clasele pentru piese jocului de şah.

Acesta conţine toate piesele jocului de şah. Avem clasa abstractă Piece ce conţine funcţia getMoves, ce oferă informaţii despre toate mutările valide ale piesei respective. Alte informaţii importante ce reţine obiectul sunt echipa de care aparţine piesa, alb sau negru, valoarea piesei şi poziţia.

În modulul Pieces avem şi Move ce reţine informaţii despre mutare. Are ca scop reţinerea mutării făcute, pentru a fi trimisă la verficare în modulul Board.py. În acelaşi timp acesta este folosit pentru a face translarea din mutarea facută de utilizator în limbaj PGN, folosit pentru cartea de deschidere.

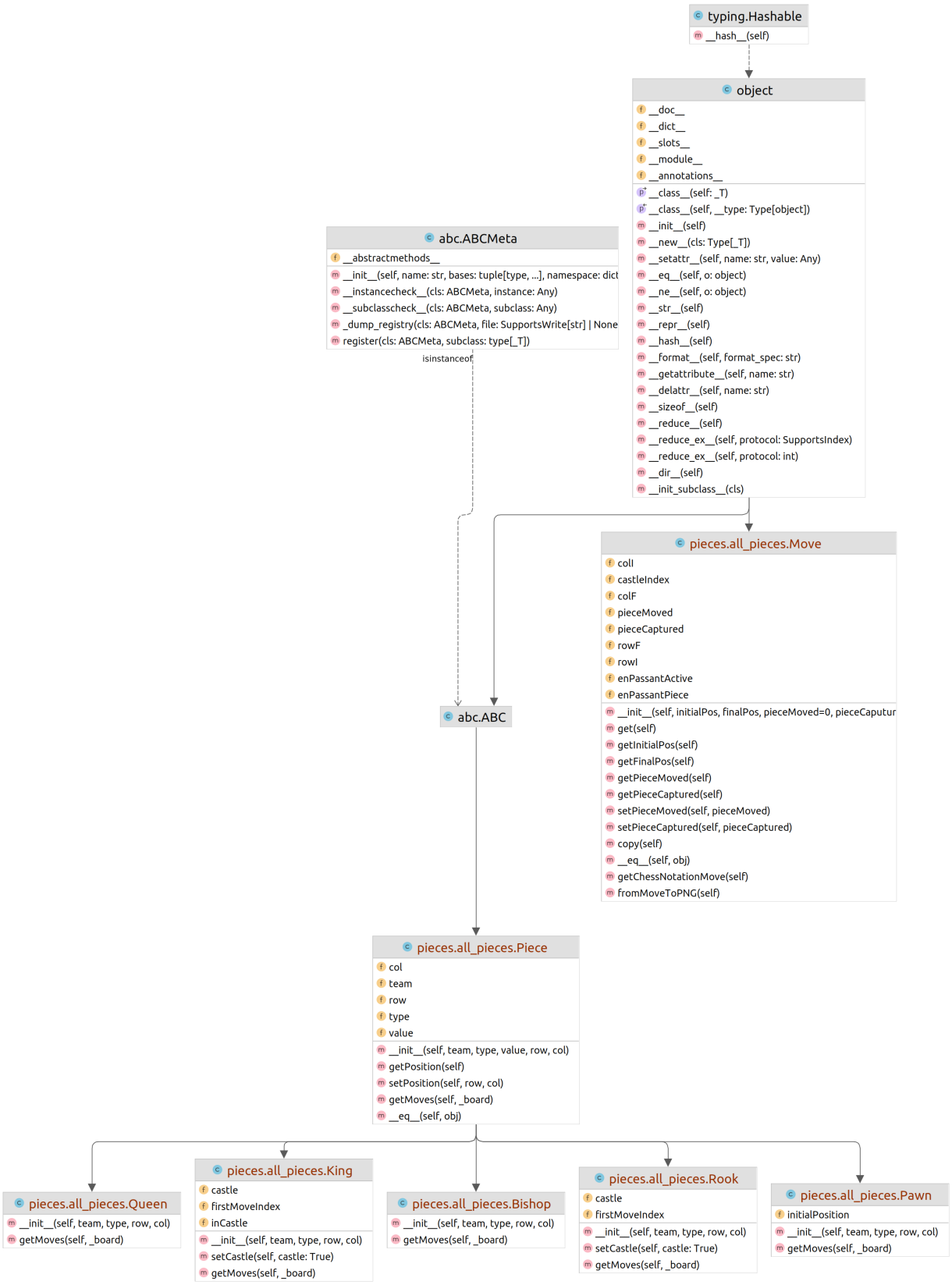

Figura 2.1. Diagrama pentru modulul Pieces[17]

Câmpurile şi metodelor clasei Piece:

- team - pentru specificarea echipei.
- type - pentru specificarea tipului de piesa (folosit în special pentru traducrea în PNG)

- value - valoarea piesei

- row, col - folosit pentru poziţia piesei pe tabla de joc

- getMoves() - funcţia folosită pentru a colecta mutările piesei

2.2.2. Modulul Score

Score.py folosit pentru calcularea scorului.

Modulul Score are ca cerinţă să evalueze poziţia jucătorului. Acest calcul se face în diferite moduri, unul dintre ele este adunarea numarul de piese existente pe tabla de joc. În ajutorul acestui calcul vin funcţii din module precum bishopScore sau pawnScore, ce folosesc teoria jocului de şah pentru a oferi un scor mai exact.

Funcţii folosite de modulul Score pentru rezultautul jocului:

- bishopPair - acordă puncte în plus dacă ambii nebuni sunt disponibili jucătorului
- doubledPawmScore - penalizează jucătorul care are doi sau mai mulţi pioni pe aceeaşi coloană
- isolatedPawn - penalizeză jucătorul care are pe tabla de joc pioni izolaţi
- rookOpenFile - acordă puncte dacă tura este pe linie fară alte piese

2.2.3. Modulul MonteCarlo

MonteCarlo.py conţine implementarea algoritmul MCTS.

Funcţia bestMove returnează mutarea cea mai bună dupa executarea algoritmului. Folosită de modulul Gui pentru executarea mutării.

Câmpuri şi funcţii relevante:

- untriedMoves - nodurile care nu au fost încă vizitate
- moveMade - reţine mutarea făcută pentru a reuşi propagarea înapoi
- bestMove - returnează mutarea cea mai bună

2.2.4. Modulul AlphaBeta

AlphaBeta.py conţine implementarea algoritmul retezarea alpha-beta.La fel ca modulul MonteCarlo acesta este accesat din modulul Gui.

2.2.5. Modulul tabelei de transpoziţie

Modulul folosit pentru tabela de transpoziţie.

Modulul Zobrist implementează crearea cheilor zobrist pentru a face o cheie unica reprezentativă stării jocului. În plus, modulul are ca rol calcularea cheilor şi inserarea acestora în baza de date Mongo împreună cu valori precum scor şi adâncimea.

Clasa ZobristTable moşteneşte clasa Document pentru trimiterea informaţiilor în MongoDB.

Câmpurile sunt:

- row
- cols - cheile generate aleator pentru fiecare celulă de pe rândul specificat de câmpul row

Clasa HashTable moşteneşte clasa Document, ce reţine cheile unice Zobrist.

Câmpurile pentru HashTable:

- hash - cheia unică Zobrist
- score - scorul tabelei
- depth - adâncimea la care a fost calculat scorul

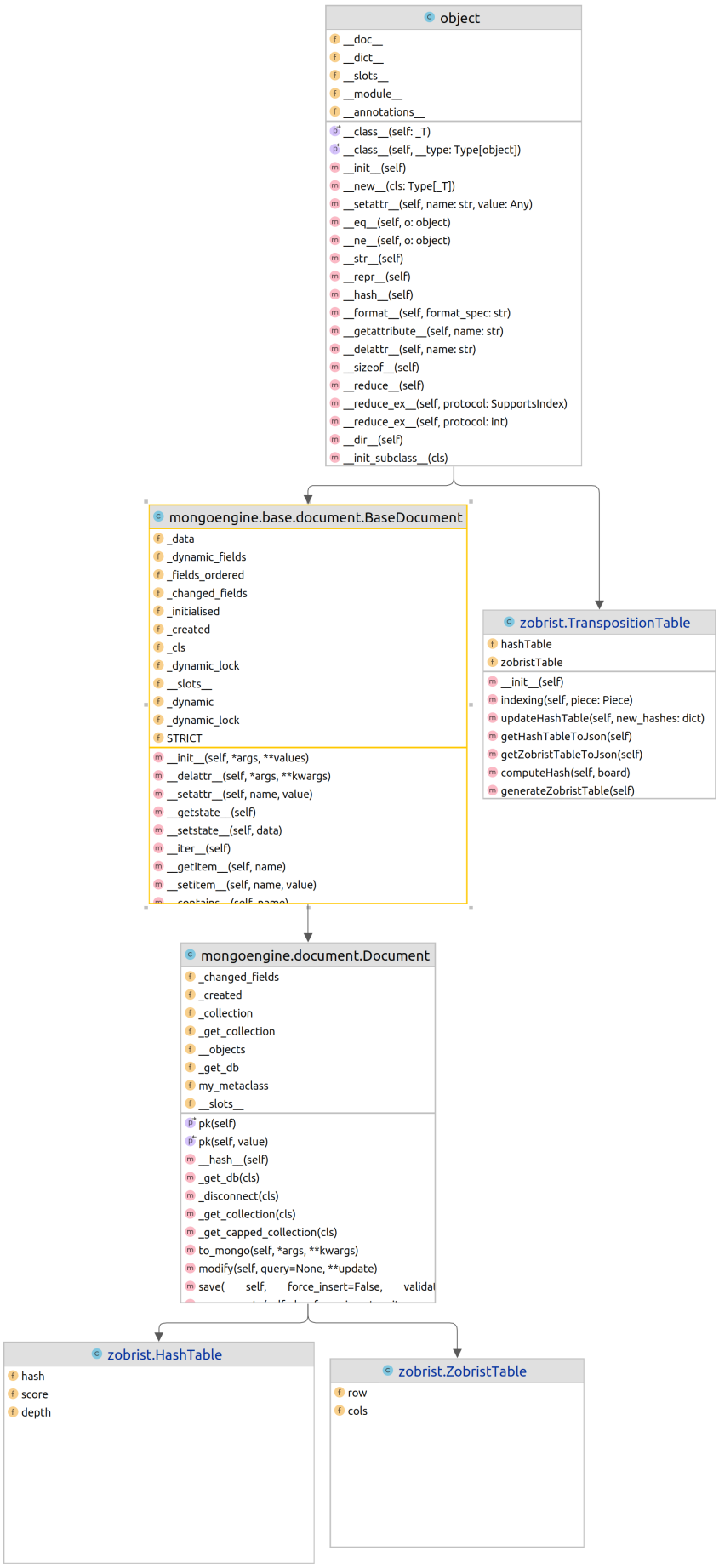

Figura 2.2. Diagrama modulului tabelei de transpoziţie[18]

2.2.6. Modulul Board

Modulul Board.py reţine logica tablei de joc.

Modulul are rolul de a reţine referinţă la piesele de joc, de a identifica daca este şah sau şah mat. Conexiunea dintre Gui şi Board se face prin funcţia guiToBoard, folosind ca variabilă un element de tip Move.

Câmpurile şi funcţiile relevante:

- pins - stochează piesele care sunt fixate
- attackPins - piesele care atacă piesele fixate
- guiToBoard - funcţia pentru a face tranziţia dintre interfaţă şi tabla de joc

2.2.7. Modulul pentru interfață

Moduluk Gui.py face conexiunea dintre utilizator si Board

Utilizatorul poate selecta dacă doreşte să joace cu AI-ul sau cu un alt utilizator. În modul AI acesta poate alege algoritmul, adâncimea şi în cât timp să returneze calculatorul mutarea considerată cea mai bună.

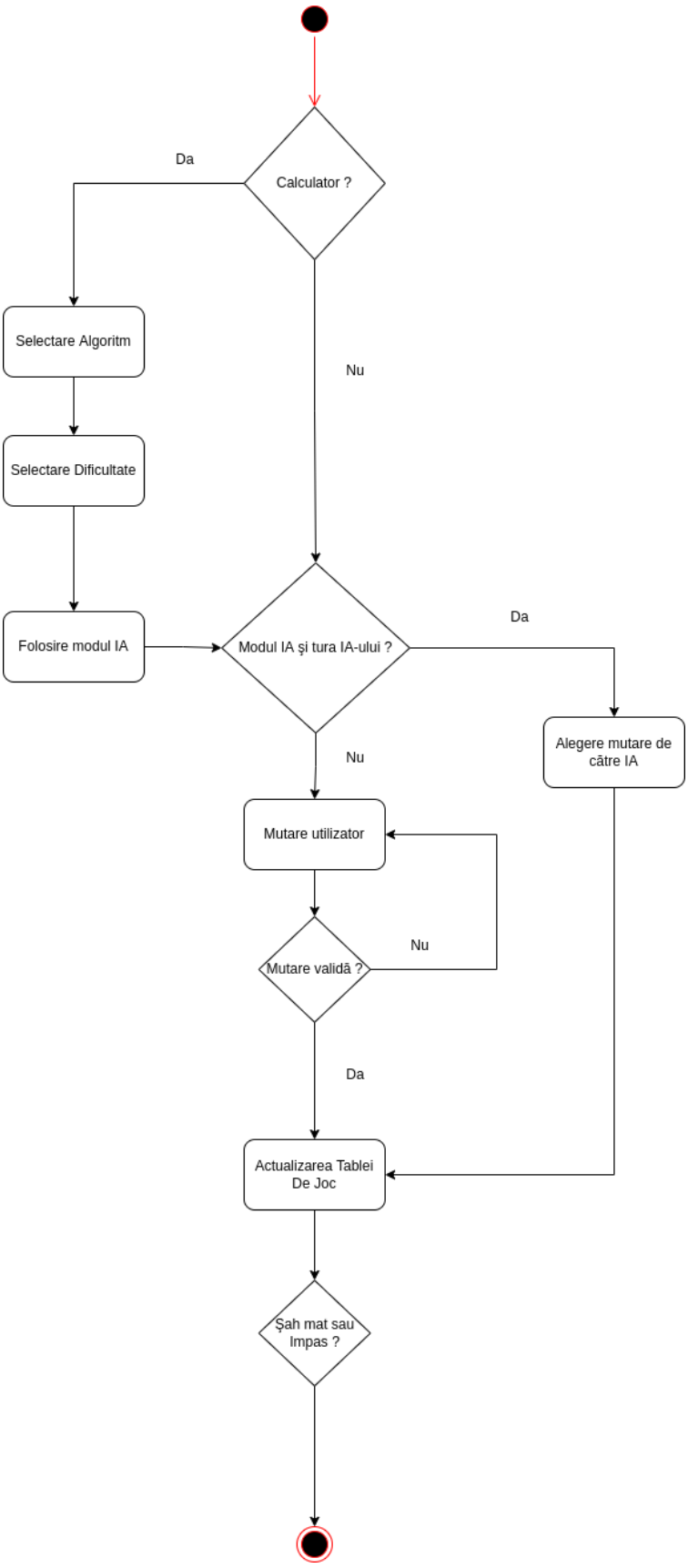

Figura 2.3. Schema logică de funcţionare a aplicaţiei[19]

2.3. Limitele în care va funcţiona aplicaţia

Aplicaţia funcţionează optim atunci când nu sunt selectate adâncimi foarte mari sau repetări în cazul Monte Carlo.

În acelaşi timp, răspunsul aplicaţiei poate să difere în funcţie de procesorul calculatorului.

Capitolul 3. Implementarea aplicației

Scopul acestei lucrări este de a aplica metode de înteligenţă artificială pentru jocul de şah. În cadrul proiectului sunt folosiţi doi algoritmi: retezarea alpha-beta şi algoritmul Monte Carlo. În acelaşi timp, algoritmii vor avea acces la o tabelă de tranpoziţie pentru salvarea stărilor.

Implementarea proiectului a fost folosit limbajul Python 3.10 şi pentru managementul bazelor de date a fost folosit MongoDB.

3.1. Implementarea logicii jocului de şah

Primul pas în aplicarea metodelor de inteligenţă artificială este de a genera tabla de joc împreună cu regulile sale. Implementarea pieselor de şah se gaseşte în folderul pieces şi generearea tablei de şah este în fişierul board.py.

3.1.1. Generarea Tabelei

Pentru generearea tabelei avem clasa Board din board.py. La crearea obiectului se contruiesc piesele, care sunt puse în poziţii specifice pe tabla de joc(matrice). Utilizatorul poate alege un anumit final de joc, în mod implicit se construieşte tabla de joc din Figura 1.1.

3.1.2. Piesele de şah

În cadrul lucrării a fost creat un folder pieces în care regasim câte un fisier Python corespunzător fiecărei piese de şah, exemplu king.py. Fiecare conţine o clasă respectivă numelui fişierului, care extinde clasa abstractă Piece din fişierul piece.py.

Clasa Piece suprascrie şi funcţia de comparare eq, pentru a verifica daca piesele comparate sunt de acelaşi tip şi dacă se află pe aceeasi poziţie pe tabla de joc.

3.1.3. Mutările posibile ale pieselor

Mutările sunt obiecte de tip Move din fişierul move.py, ce conţin informaţii despre poziţia iniţială, poziţia finală, referinţă la piesa mutată şi piesa capturată.

Toate clasele pieselor conţin funcţia getMoves ce returnează toate mutările piesei. Aceste mutări nu sunt verificate dacă pun în pericol regele, verificarea se face de funţia validMovesPieceWithChecking din board.py.

Pentru mutările valide ale turei, implementarea constă în parcurgerea matricei în direcţiile valide ale turei, acestea fiind în sus, jos, drepta, stânga. Se parcurg direcţiile şi se construiesc mutările cât sunt în interiorul tabelei. Daca se găsesc celule goale, egale cu zero, se adaugă în lista de mutări. Căutarea se opreşte pe o anumită direcţie atunci cand este întâlnită o piesă inamică sau din aceeaşi echipă.

3.1.4. Verificarea pentru şah, şah mat şi impas

Mutările returnate de getMoves pentru fiecare piesa, nu verifică daca pun în pericol regele, specificat şi în secţiunea anterioară. În clasa Board, avem funcţia getChecksAndPins pentru identificarea pieselor care nu pot fi mutate şi piesele adversarului care pun în şah regele. Tot o data, getEscapeCheckValidMoves returnează mutările valide pentru scăparea din şah dacă acestea există.

3.1.4.1. Şah şi piesele fixate

În funcţia getChecksAndPins se colectează informaţii despre pisele fixate şi piesele care dau şah. Pentru a optimiza calculul, se iau direcţiile din care poate fi atacat regele şi nu toate mutările posibile ale pieselor adversarului.

Pentru fiecare direcţie verificăm dacă piesa adversa poate ataca pe direcţia respectivă şi daca atacul este blocat de o piesă. Pe parcursul verificării direcţiei se colectează informaţii despre celulele dintre rege şi piesă adversarului. Acestea sunt folosite de funcţia getEscapeCheckValidMoves, pentu a bloca atacul inamicului sau de a captura.

Tot o data, lista pins conţine toate piesele ce nu pot fi mutate deoarece acesta blochează atacul şi prin mutarea lor, regele ar intra automat în şah. Implementarea se poate vedea în codul 3.1.

```
1   if (((piece.type == 'p' and i == 1 and
2        (direction in [(1, 1), (-1, -1), (-1, 1), (1, -1)])) and
3        ((player and direction[0] < 0)
4        or (player == False and direction[0] > 0))) or
5        (piece.type == 'R' and
6        (direction in [(0, 1), (0, -1), (-1, 0), (1, 0)])) or
7        (piece.type == 'B' and
8        (direction in [(1, 1), (-1, -1), (-1, 1), (1, -1)])) or
9        (piece.type == 'Q') or
10       (piece.type == 'K' and i == 1))
```

Listing 3.1. Şah şi piese fixate

3.1.4.2. Mutările pentru scăparea din şah

Functţia getEscapeCheckValidMoves, returnează mutările prin care şahul poate fi blocat. Aceasta foloseşte parametrii returnaţi de funcţia getChecksAndPins, descrisă la punctul anterior.

Dacă regele primeşte şah doar dintr-un punct, atacul poate fi oprit nu doar de rege. Pentru atacul din două direcţii regele este singura piesă care poate bloca atacul. La final, validMoves conţine toate mutările valide, dacă acesta este gol, rezultă faptul că regele este în şah mat.

3.1.5. Mutarea pe tabla de şah

Funcţia move din Board, face mutarea după ce aceasta a fost validată de funcţiile descrise anterior. Aceasta are rolul de a poziţiona corect piesele mutate dar şi pentru a face o salvare a stării.

3.1.6. Revenirea la starea anterioară

Pentru revenirea la starea anterioară, se foloseşte funcţia undoMove, aceasta extrage ultima mutare făcută din lista logMoves şi repoziţionează piesele şi stările precum dacă regele este în şah sau poate să facă rocada.

3.2. Metode de inteligenţă artificială

În cadrul proiectului, sunt folosiţi doi algoritmi, retezarea alpha-beta şi Monte Carlo. Implementările pot fi găsite în fişierele alphaBeta.py şi monteCarlo.py.

3.2.1. Retezarea alpha-beta

În fişierul alphaBeta.py avem funcţia bestMoveMinMax ce primeşte ca parametrii mutările valide şi o referinţă la un obiect Board. Aceasta apelează funcţia minimax ce are implementat algoritmul.

Recursivitatea algoritmului se termină atunci când este ajuns la adâncimea maximă sau jocul a ajuns într-o stare finală precum şah mat sau impas. Acest lucru este verificat prin variabila status din Board, fiind actualizată mereu, care specifică starea jocului. În pseudocodul din 3.2 este prezentat algoritmul retezarea alpha-beta.

Atunci când este apelat din nou funcţia minimax, se foloseşte funcţia aiToBoard(move) ce trimite mutarea şi actualizează starea. După ieşirea din funcţia minimax se apelează funcţia

undoMove pentru revenirea la starea anterioară.

```
if (stare finala sau adâncimea=0)
    return score
if playerTurn:
    maxScore = -INF
    for move in validMoves:
        score = minimax(board, nextMoves, depth-1, False, alpha, beta)
        maxScore = max(maxScore, score)
        alpha = max(alpha, maxScore)
        if beta <= alpha:
            break
    return maxScore
else:
    minScore = INF
    for move in validMoves:
        score = minimax(board, nextMoves, depth-1, True, alpha, beta)
        minScore = min(minScore, score)
        beta = min(beta, minScore)
        if beta <= alpha:
            break
    return minScore
```

Listing 3.2. Pseudocode retezarea alpha-beta

În cadrul algoritmului a fost introdusă şi verificarea cu tabela de transpoziţie. Se efectuează operaţia de hashurie prin funcţia computeHash şi se verifică daca există în baza de date. Dacă valoarea este găsită, se colectează scorul din tabelă şi se foloseşte fără a mai efectua calculul (cod 3.3).

```
hash = zob.computeHash(board.board)
if hash not in zob.hashTable:
    return scoreMaterial(board) + score
else:
    return zob.hashTable[hash]["score"]
```

Listing 3.3. Verificarea cu tabela de transpoziţie

3.2.2. Algoritmul Monte Carlo

Al doilea algoritm folosit, Monte Carlo, a fost implementat în clasa MonteCarloTreeSearchNode din fişierul monteCarlo.py.

Clasa are ca parametrii o referinţa la un obiect Board, o variabila pentru stocarea scorului, adâncimea pentru simulare şi untrieMoves o listă pentru nodurile care nu au fost expandate încă. Fiecare obiect are si o listă children pentru reţinerea informaţiilor despre copii nodului.

Pentru oprirea simulării se foloseşte adâncimea sau starea finală a jocului. Pentru calcularea scorului se foloseşte funcţia scoreMaterial din fişierul score.py.

Prin funcţia bestMove, cod 3.4, utilizatorul obţine mutarea considerată cea mai bună.

```
for _ in range(repeats):
     node = self.selection()
     result = node.simulation()
     node.backpropagation(result)
 best_child: MonteCarloTreeSearchNode = self.bestChildWithdScore()
 return best_child.moveMade
```

Listing 3.4. Obţinearea mutării optime prin algoritmul Monte Carlo

Funcţia selection, cod 3.5, parcurge arbore cât timp nu este o frunză. Dacă nodul găsit nu este complet expandat acesta apelează funcţia expansion pentru a obţine alt nod şi de al returna.

Pentru nodurile complet expandate se apelează funcţia bestChild, pentru a opţine cel mai favorabil nod, folosind formula UCB.

Se simulează nodul returnat de selection(), cât timp nu este stare finală sau adâncimea atinsă.

Prin funcţia backpropagation se returnează starea la nodul rădăcină, actualizând pe parcurs valorile nodurilor.

La final este folosită funcţia bestChildWithScore pentru a returna nodul copil cu rezultatul cel mai bun.

```
1   def selection(self):
2       current_node = self
3       while not current_node.is_terminal_node():
4           if not current_node.is_fully_expanded():
5               return current_node.expansion()
6           else:
7               current_node: MonteCarloTreeSearchNode = current_node.best_child()
8               self.board.aiToBoard(current_node.moveMade)
9       return current_node
```

Listing 3.5. Selecţia din Monte Carlo

3.3. Tabela de transpoziţie

Tabela de transpoziţie a fost implementată pentru a eficientiza obţinerea rezultatelor.

În fişierul zobrist.py avem doua clase ZobristTable şi HashTable folosite pentru a stoca informaţii in MongoDB.

1. ZobristTable:
   - row - tip int pentru a specifica linia
   - cols - tip matrice ce reţine toata celulele de pe rândul row
2. HashTable:
   - hash - cheia unica Zobrist pentru identificarea stării
   - score - scorul stării respective
   - depth - adâncimea la care a fost calculat scorul

Clasa TranspositionTable folosită pentru a genera tabelele Zobrist şi Hash. Pentru extragerea informaţiilor din MongoDB se foloseşte mongoengine. Un exemplu ar fi codul 3.6.

```
1   def getHashTableToJson(self):
2       zob_dict = {}
3       docs = list(HashTable.objects())
4       for obj in docs:
5           obj_json = json.loads(obj.to_json())
6           zob_dict[obj_json["_id"]] = {
7               "score": obj_json["score"],
8               "depth": obj_json["depth"]
9               }
10      disconnect()
11      return zob_dict
```

Listing 3.6. Extragerea informaţiilor din tabela de transpoziţie

Operaţia de hashing, este implementată în funcţia computeHash care primeşte ca referinţă la un obiect Board. Se execută funcţia XOR pe toate celulele diferite de zero(cod3.7).

```
1   def computeHash(self, board):
2       h = 0
3       for i in range(8):
4           for j in range(8):
5               if board[i][j] != 0:
6                   piece = self.indexing(board[i][j])
7                   h ^= self.zobristTable[i][j][piece]
8       return h
```

Listing 3.7. Operaţia de hashing

3.4. Interfaţa

Interfaţa a fost dezvoltată folosind modulul Pygame specifcă pentru implementarea jocurilor în limbajul Python.

La pornirea programului Figura 3.1, utilizatorul poate alege opţiunile.

Figura 3.1. Interfaţă de pornire a programului

3.4.1. Opţiunele pentru jucători

Utilizatorul poate alege ce jucător să fie gestionat de calculator şi ce algoritm foloseşte.

Din Figura 3.2, utilizatorul selectează algoritmul folosit dar şi dificultatea, setată de adâncime. Altă opţiune este alegerea tablei de joc, utilizator poate selecta diferite finaluri de joc pentru exersare.

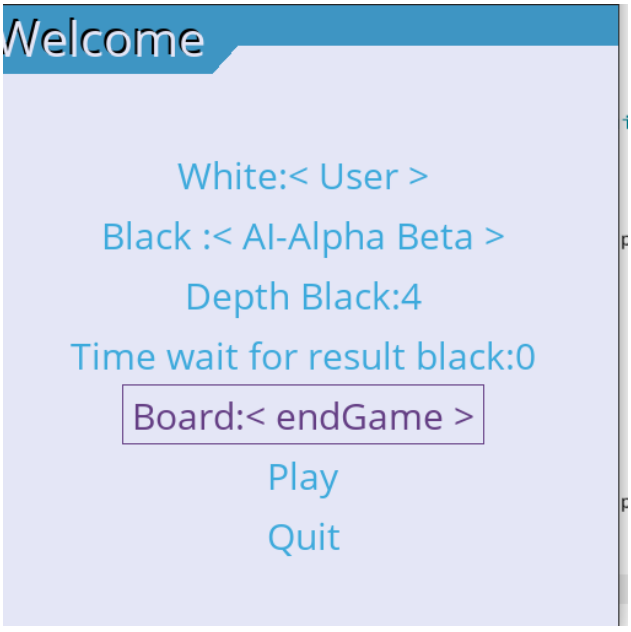

Figura 3.2. Opţiunile utilizatorului

3.4.2. Pornirea jocului

După ce utilizatorul selectează butonul Play, fereastra tablei de joc este activată.

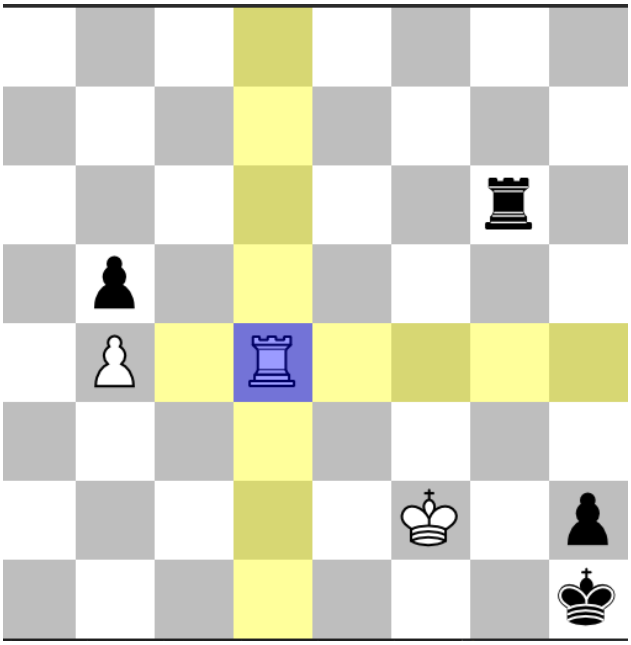

Figura 3.3. Tabla de joc din interfaţă

Capitolul 4. Testarea aplicației și rezultate experimentale

4.1. Lasarea aplicaţiei

Pentru lansarea în execuţie a aplicaţiei este necesără instalarea următoarelor module:

1. pip install mongoengine
2. pip install pygame
3. pip install numpy

Algoritmii au fost testaţi în cazul unui joc final dintre GM Judit Polgar (2686) - GM Veselin Topalov (2786) din 2010 unde poziţia albului este considerată favorabilă.

4.2. Algoritmul Monte Carlo

Algoritmul Monte Carlo pentru obţinerea unor rezultate favorabile acestă necesită un număr mare de repetări.

4.2.1. Numărul de mutări

Figura 4.1 pune în evidenţă relaţia dintre numărul de simulări şi numărul de mutări necesare pentru a obţine victoria.

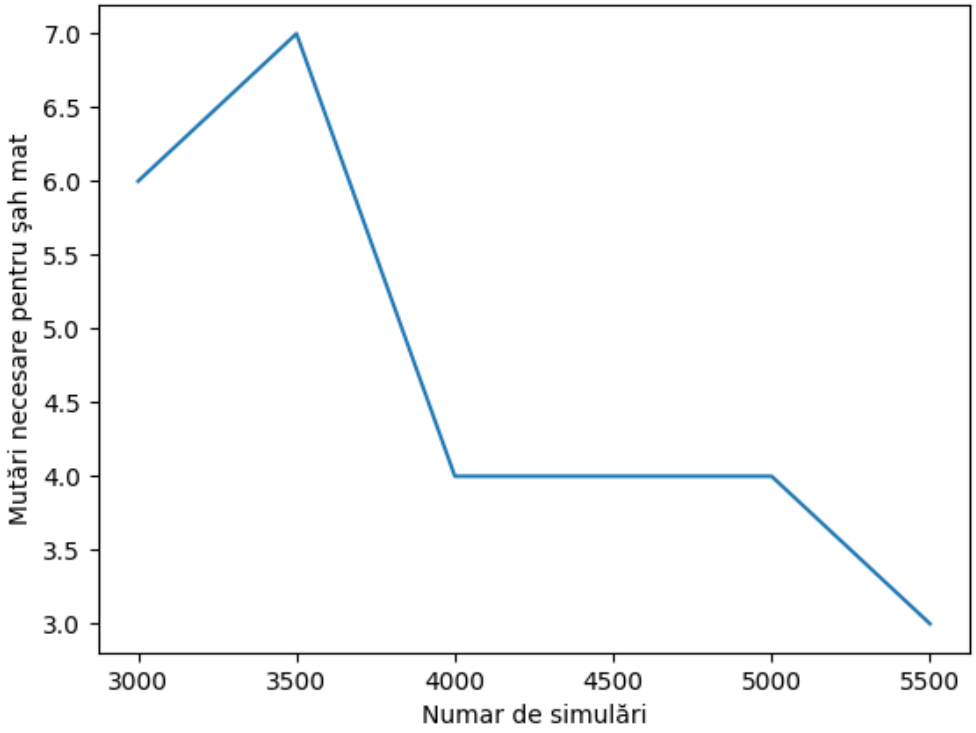

Figura 4.1. Numarul de simulări Monte Carlo[20]

4.2.2. Numărul de minute acordate

Figura 4.2 evidenţiază faptul că timpul acordat unei mutări influenţează procentajul de câştig(1: câştig, 0: egalitate, -1: înfrângere).

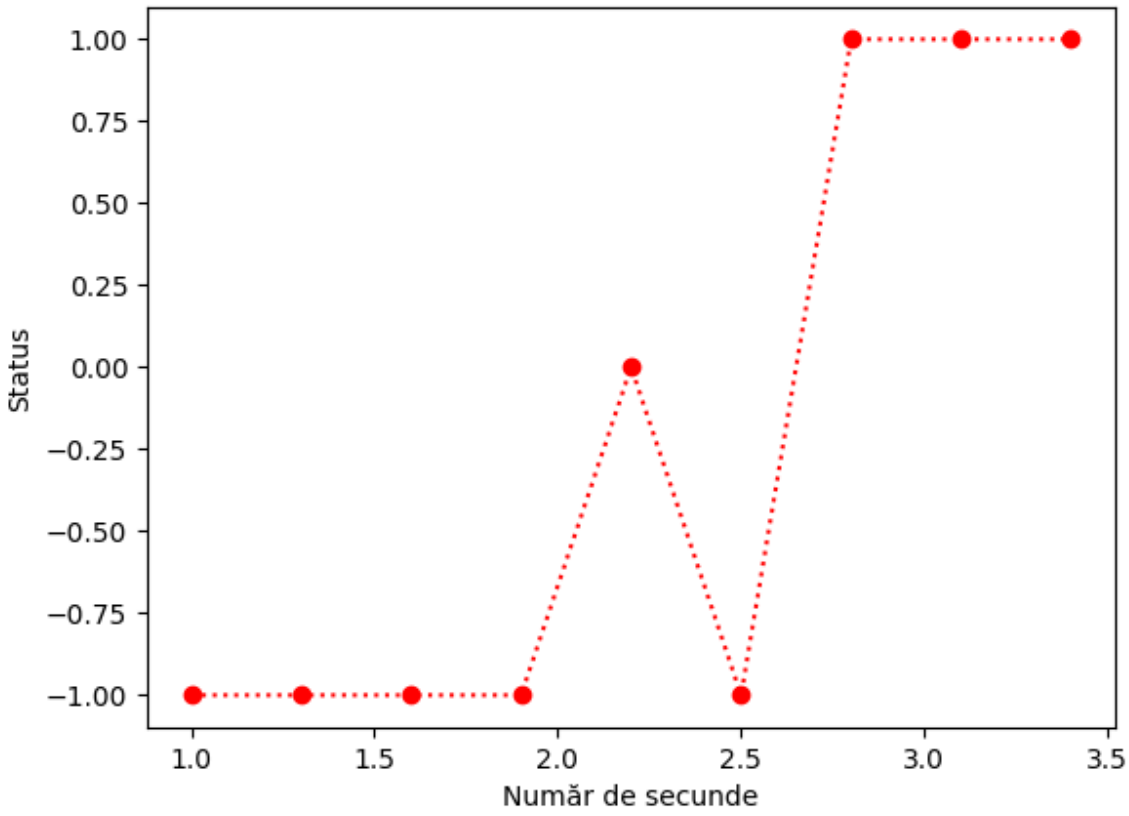

Figura 4.2. Influenţa timpului acordat asupra câştigului[21]

4.3. Algoritmul retezarea alpha-beta

Algoritmul retezarea alpha-beta necesită setarea unei adâncimi mari pentru obţinerea rezultatelor mai bune.

4.3.1. Adâncimea căutării

Figura 4.3 pune în evidenţă relaţia dintre adâncime şi precizia mutării.

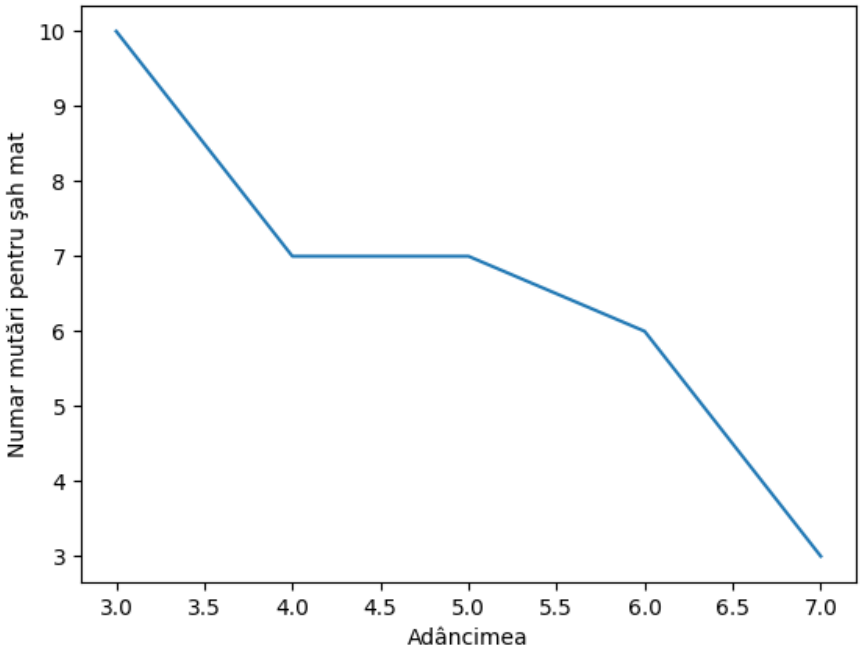

Figura 4.3. Adâncimea în relaţia cu numărul de mutări[22]

4.4. Tabela de transpoziţie

Tabela de transpoziţie este folosită pentru a stoca stările anterior calculate. Dacă starea curentă este găsită în baza de date, se utilizează scorul deja calculat pentru a evita calculele. Se poate observa în Figura 4.4 diferenţa folosirii unei tabele de transpoziţie.

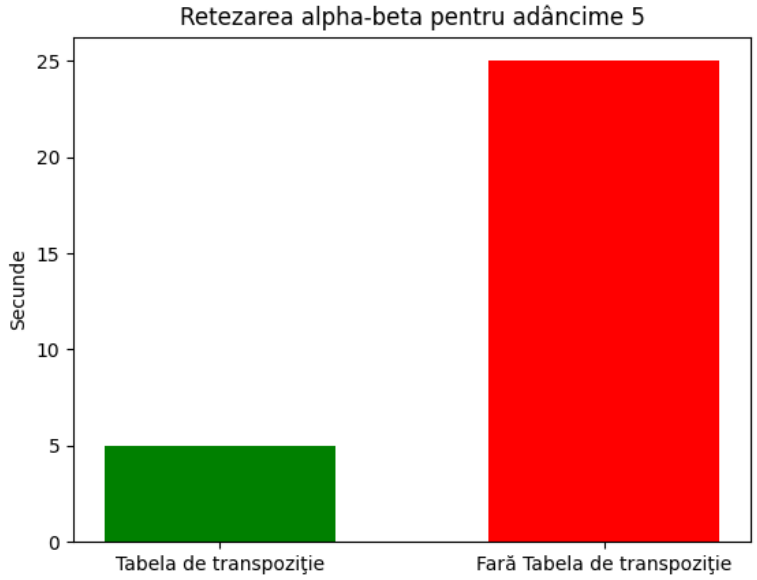

Figura 4.4. Tabela de transpoziţtie barchart[23]

Un dezavantaj pentru folosirea tabelei de tranpoziţie este încărcarea bazei de date. Se poate obeseva în Figura 4.5 numarul de documente în relaţie cu adâncimea.

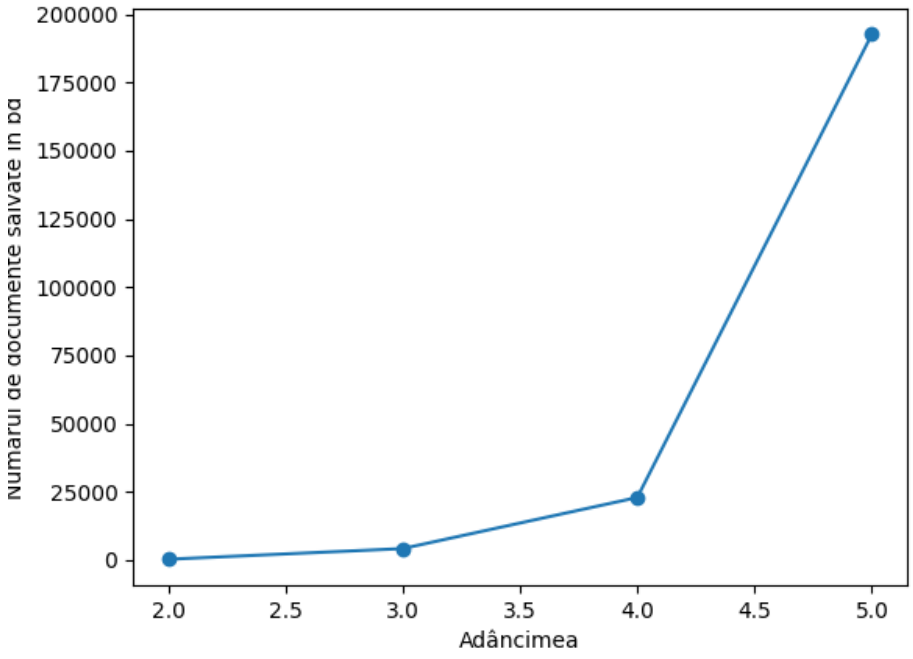

Figura 4.5. Adâncimea şi numărul de documente[24]

Figura 4.6 prezintă nivelul de utilizare a procesorului atunci când nu sunt folosiţi algoritmii ci doar logica jocului de şah.

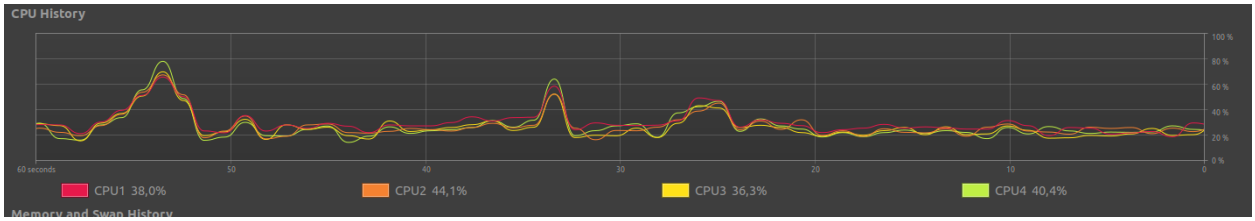

Figura 4.6. Nivelul de încărcare a procesorului pentru logica de şah[25]

Figura 4.7 prezintă nivelul de utilizare a procesorului atunci când sunt folosiţi ambii algoritmi.

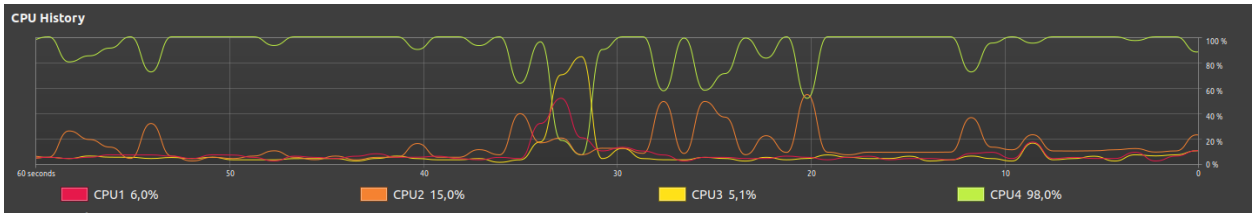

Figura 4.7. Nivelul de încărcare a procesorului[26]

Concluzii

Proiectul implementează logica jocului de şah, a algoritmilor Monte Carlo şi retezarea alfa-beta, tabela de transpoziţie pentru salvarea stărilor şi interfaţa cu utilizatorul.

Scopul aplicaţiei este de a compara algoritmii în diferite moduri şi de a identifica care oferă rezultate mai bune. Aplicaţia oferă în acelaşi timp şi diferite finaluri de joc pentru antrenare.

În comparaţie cu alte proiecte similare analizate, Stockfish şi AlphaZero, aplicaţia nu găseşte întotdeauna mutarea considerată cea mai bună. Acesta în schimb, reuşeşte să rezolve puzzle-de şah de nivel mediu şi poate fi o provocare pentru cei nou iniţiaţi în jocul de şah.

Direcţii de dezvoltare:

- Tabela de transpoziţie: Aceasta foloseşte multă memorie si încărcarea în sistem necesită timp. Pentru a diminua aceste probleme se poate împărţi baza de date în trei colecţii pentru jocul de început, jocul de mijloc şi jocul de final. Aplicaţia ar încărca în sistem doar colecţia specifică starii jocului.

- Paralelism: Algoritmii folosiţi, pentru a returna un rezultat corect necesită fortă brută computaţională, acesta conduce la un timp de răspuns lent. Paralelismul ar putea fi folosit pentru a împărţi calculul arborelui.

- Calcularea scorului: Pentru returnarea unui rezultat mai exact este necesară aplicarea teoriei de şah. Elemente de calcul care pot fi dezvoltate sunt: identificarea celulelor slabe/puternice şi dezvoltarea pieselor fiecărui jucător.

Bibliografie

[1] D. Shenk, *The immortal game : a history of chess*, 1st ed. New York : Doubleday, 2007. [Online]. Available: https://archive.org/details/isbn_9780385510103/page/n1/mode/2up

[2] B. Edwards, "A brief history of computer chess," https://www.pcworld.com/article/451599/a-brief-history-of-computer-chess, 2013, Ultima accesare: 5.07.2022.

[3] A. Bernstein, "A CHESS PLAYING PROGRAM FOR THE IBM 704," https://www.computerhistory.org/chess/doc-4316153963418/, p. 2, 1959-03, Ultima accesare: 5.07.2022.

[4] F. hsiung Hsu, *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton University Press, 2002. [Online]. Available: https://archive.org/details/isbn_9780385510103/page/n1/mode/2up

[5] Stockfish, "Stockfish 15," https://stockfishchess.org/blog/2022/stockfish-15/, 2022-04-18, Ultima accesare: 05.07.2022.

[6] J. S. David Silver, Thomas Hubert, *A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play*, ser. 6419. Science, 2018, vol. 362, DOI: 110.1126/science.aar6404.

[7] J. Veness and A. Blair, "Effective use of transposition tables in stochastic game tree search," in *2007 IEEE Symposium on Computational Intelligence and Games*, 2007, pp. 112–116, ISBN: 1-4244-0709-5, DOI: 10.1109/CIG.2007.368086.

[8] A. L. Zobrist, "A new hashing method with applications for game playing," University of Wisconsin, Madison, Wisconsin, Tech. Rep. 88, 1969. [Online]. Available: https://research.cs.wisc.edu/techreports/1970/TR88.pdf

[9] R. Basbous and B. Nagy, "Generalized game trees and their evaluation," in *2014 5th IEEE Conference on Cognitive Infocommunications (CogInfoCom)*, 2014, pp. 55–60, DOI: 10.1109/CogInfoCom.2014.7020518.

[10] F. Leon, "Note de curs-ia," http://florinleon.byethost24.com/curs_ia.html, 2022.

[11] ——, "Note de curs-inva," http://florinleon.byethost24.com/curs_ml.html, 2022.

[12] F.-H. Hsu, "Ibm's deep blue chess grandmaster chips," *IEEE Micro*, vol. 19, no. 2, pp. 70–81, 1999, DOI: 10.1109/40.755469.

[13] T. T. C. GmbH, "How do modern chess engines work?" https://youtu.be/pUyURF1Tqvg, 2015, Ultima accesare: 5.07.2022.

[14] PyGame, "Pygame," https://www.pygame.org/docs/, 2022, Ultima accesare: 5.07.2022.

[15] N. Developers, "Numpy," https://numpy.org/doc/stable/, 2022, Ultima accesare: 5.07.2022.

[16] M. Developers, "Mongoengine," https://docs.mongoengine.org/, 2022, Ultima accesare: 5.07.2022.

[17] J. K, "All about mongodb: The nosql database," https://acodez.in/mongodb-nosql-database/, 2029, Ultima accesare: 5.07.2022.