

UNIVERSITATEA NAȚIONALĂ DE ȘTIINȚĂ SI TEHNOLOGIE
POLITEHNICA DIN BUCUREȘTI
FACULTATEA DE AUTOMATICĂ SI CALCULATOARE
DEPARTAMENTUL DE CALCULATOARE



PROIECT DE DIPLOMĂ

Aplicație pentru învățare șah
Versiunea 2024

Lucas Lăzăroiu

Coordonator științific:

Conf. Dr. Ing. Anca Morar

BUCUREȘTI
2024

NATIONAL UNIVERSITY OF SCIENCE AND TECHNOLOGY
POLITEHNICA BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE AND ENGINEERING DEPARTMENT



DIPLOMA PROJECT

Chess Learning App
2024 version

Lucas Lăzăroiu

Thesis advisor:
Conf. Dr. Ing. Anca Morar

BUCHAREST
2024

CUPRINS

1 Introducere	1
1.1 Context	1
1.2 Obiective	1
1.3 Solutia propusă	2
1.4 Rezultatele obținute	2
1.5 Structura lucrării	2
2 Analiza și Specificarea Cerințelor	4
3 Studiu de Piață	7
4 Soluția Propusă	11
5 Detalii de implementare	14
5.1 GameState	14
5.2 GameStateManager	16
5.3 KingSafety	17
5.4 MoveGenerator	19
5.5 Game	22
5.6 PerfIt	24
5.7 Funcții statice de evaluare	26
5.7.1 Funcții bazate pe pătrate controlate	27
5.7.2 Funcții bazate pe amplasarea pieselor	27
5.8 Minimax cu Alpha Beta Pruning	28
5.9 Iterative Deepening Search	31
5.10 Ordonarea Mutărilor	33
5.11 Tabele de transpoziție	34

5.12 Al	36
6 Evaluare	37
6.1 Contextul Evaluării	37
6.2 Rezultatele evaluării	40
7 Concluzii	44
7.1 Dezvoltări ulterioare	44
Anexe	46
Anexa A Extrase de cod	54

SINOPSIS

Jocul de șah posedă o capacitate uimitoare de a conecta oamenii. Această atracțivitate este dată atât de regulile relativ simple, cât și de nenumăratele posibilități pe care acestea le oferă. Din fericire, începând cu anul 2020, șahul a crescut în popularitate datorită promovării intense de către creatorii populari de conținut, pe platforme online precum Twitch.tv și YouTube.com. În mod natural, a apărut și un influx de jucători noi, cu un nivel scăzut de cunoștințe despre șah. În plus, odată cu migrarea în mediul online, multe tipuri de resurse de învățare au devenit ușor accesibile pentru publicul general. Printre acestea se numără și motoarele de șah, programe software care și-au depășit demult proprii creatori, folosind o combinație între forță brută și tehnici dezvoltate de-a lungul unei perioade extinse de timp. Acest proiect prezintă o aplicație destinată publicului țintă menționat mai sus, cu un accent pus pe dezvoltarea diferitelor motoare de șah din cadrul acesteia și compararea lor cu unul dintre cele mai puternice motoare de șah la momentul actual, Stockfish 16.1.

ABSTRACT

The game of chess possesses an amazing capacity to connect people. The appeal lies in its relatively simple rules, as well as in the uncountable number of possibilities they offer. Fortunately, starting with the year 2020, chess has been increasing in popularity due to its active promotion by popular content creators on online platforms such as Twitch.tv and YouTube.com. Naturally, there was an influx of new players, with low-level chess knowledge. Furthermore, with the migration to the online environment, many types of learning resources became easily accessible to the general public. Among these are also chess engines, software programs that have long surpassed their own creators, using a combination between brute force and techniques developed over a long time span. This project presents an application meant for the aforementioned target audience, with an emphasis on the development of the various chess engines within it, and their comparison to one of the strongest chess engines to date, Stockfish 16.1.

1 INTRODUCERE

1.1 Context

Datorită naturii activității, jucătorii de șah pot beneficia de dezvoltare personală [1], profesională [2], chiar și de niveluri reduse de anxietate generală și stres [3]. Proiectul urmărește dezvoltarea unei aplicații Desktop cu o interfață simplă dar intuitivă, ce oferă utilizatorilor o experiență valoroasă de învățare a fundamentelor jocului de șah. Se va satisface cererea naturală de resurse de învățare a noilor jucători. Acest lucru este posibil prin prisma perioadei actuale, perioadă în care șahul a atinge o semnificație considerabilă, în special în lumea divertismentului [4].

Cea mai fascinantă parte este dezvoltarea mai multor motoare de șah [5] și evaluarea lor, punând în perspectivă efectele diverselor tehnici de optimizare sau euristici aplicate. Este de notat faptul că acestea au un număr foarte mare de aplicabilități în lumea ingineriei (e.g. în sisteme GNSS [6] sau în evaluarea utilizabilității aplicațiilor mobile pentru învățare [7]), jocul de șah combinat cu publicul țintă de începători fiind doar un caz particular ce va fi abordat în cele ce urmează.

1.2 Obiective

Obiectivul final al proiectului este de a oferi o aplicație funcțională, cu un caracter educativ, ce va stârni interesul utilizatorilor. Ne propunem ca aplicația să fie ușor de utilizat pentru cineva atât cu experiență tehnică minimală, cât și pentru cineva ce nu este deloc familiar cu regulile jocului, prin intermediul tutorialelor.

Numeroasele motoare de șah, integrate în aplicație și unice în adevăratul sens al cuvântului, vor putea oferi o experiență inedită și constructivă, ideală pentru învățare. Acestea nu trebuie să fie perfecte, astfel încât să descurajeze jucătorii, ci să ofere un nivel adecvat de dificultate ce facilitează progresul. Un utilizator trebuie să aibă o stare de satisfacție internalizată și sentimente de auto-creștere atunci când învinge un adversar AI, ci nu să simtă frustrare sau dezinteres.

Din punct de vedere tehnic, ne propunem să tragem niște concluzii bazate pe rezultate ale diferitelor tehnici de dezvoltare a motoarelor de șah: ce tehnici funcționează cel mai bine împreună, ce tehnici oferă rezultatele cele mai bune și de ce, dar și cum se compară motoarele cu unele deja existente.

1.3 Soluția propusă

Aplicația conține două moduri de joc: unul introductiv, ce conține tutoriale vizate spre învățarea regulilor jocului, și modul principal, împotriva adversarilor AI.

Tutorialele conțin informații despre diferitele tipuri de piese, modul în care acestea se mișcă pe tablă, reguli speciale precum en-passant sau rochada, până la concepte generale despre modul de abordare al tuturor stagiiilor jocului.

Modul principal conține mai mulți adversari AI, fiecare cu propriul "stil" de joc, cu scopul de a facilita adaptabilitatea și recunoașterea tipelor de către utilizatori.

Stilurile diferite ale motoarelor de șah sunt date de diversi algoritmi și euristică, despre care se va detalia în capitolele următoare.

1.4 Rezultatele obținute

Mai multe motoare de șah au fost dezvoltate și evaluate atât din punct de vedere obiectiv, folosind date obținute de la sisteme existente, dar și subiectiv, din punctul de vedere al utilizatorilor.

Din punct de vedere obiectiv, au fost comparate cu succes tehniciile folosite și au fost puse în contrast implicațiile, avantajele, și dezavantajele lor.

Din punct de vedere subiectiv, au existat păreri pozitive cu privire la funcționalitățile și resursele propriu-zise oferite de aplicație, dar și descrieri sugestive ale stilurilor de joc și nivelurilor de dificultate ale adversarilor AI.

1.5 Structura lucrării

Pe lângă **Introducere**, în lucrarea prezentată se disting încă 6 capitole importante:

- **Analiza și Specificarea Cerințelor:** prezintă cerințele aplicației propuse, deduse prin prisma informațiilor colectate de la potențiali clienți.
- **Studiu de Piață:** se face o analiză a competitorilor de pe piață curentă și se pun în perspectivă caracteristicile produselor lor cu diferite statistici colectate, cu scopul final de a crea o aplicație competitivă.
- **Soluția Propusă:** prezintă arhitectura și componentele aplicației propuse și descrie procesul iterativ al dezvoltării acestieia.
- **Detalii de Implementare:** Se face introducerea unor concepte teoretice sau de context și se intră în detaliu cu privire la tehnici și algoritmi utilizați. Sunt folosite imagini și figuri sugestive pentru o parcurgere naturală și facilă.

- **Evaluare:** Se propune un model riguros dar logic de evaluare. Folosind valori numerice și grafice, se discută cu argumente pertinente despre efectele tehnicielor folosite.
- **Concluzii:** Se sumarizează tot conținutul lucrării, dar se prezintă și posibile dezvoltări ulterioare.

2 ANALIZA ȘI SPECIFICAREA CERINȚELOR

Pentru a ne face o idee despre nevoile potențialilor clienți, s-a întocmit un formular¹ pentru centralizarea informațiilor în mod anonimizat.

Întrebările din formular au scopul de a identifica cât de folositoare ar fi o astfel de aplicație pentru publicul general și care sunt cele mai importante caracteristici ale acesteia. Vom afla cât de mare este cererea pentru un astfel de produs (câte persoane doresc și ar beneficia de pe urma lui) și ne vom da seama și de potențialele funcționalități ce pot fi adăugate în urma feedback-ului obținut.

Formularul a fost completat de către 46 de participanți, printre care se numără studenți din cadrul Facultății de Automatică și Calculatoare UNSTPB, dar și de la alte facultăți. Printre aspectele cheie ce pot fi desprinse din răspunsurile participanților se află:

- 60.9% dintre participanți raportează că joacă săh în mod activ. Printre ceilalți 39.1%, un procent semnificativ, 77.7%, și-au exprimat interesul în a învăța (Figura 1).

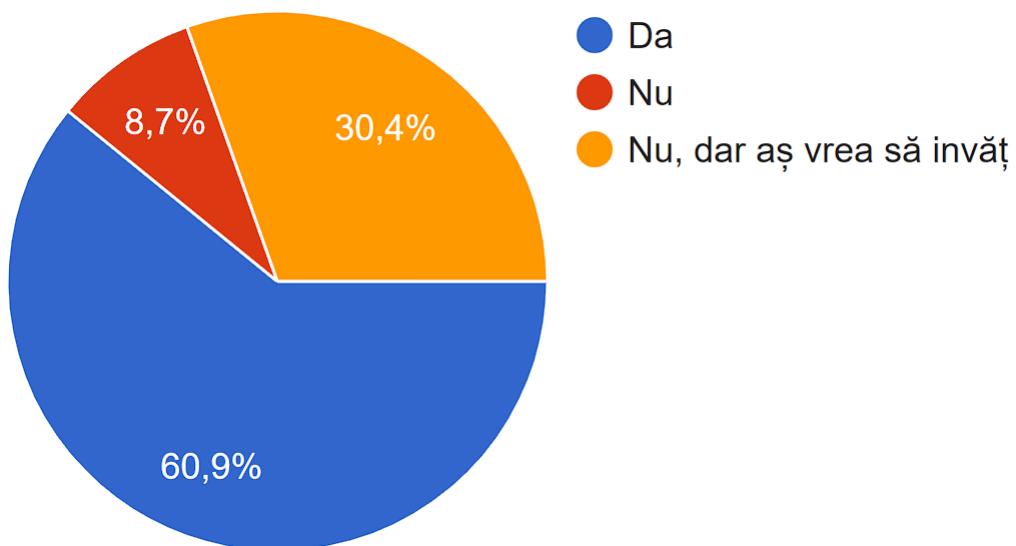


Figura 1: Răspunsuri la întrebarea **"Jucăti săh?"**

- 78.3% dintre participanți își descriu nivelul de experiență cel mult "intermediar", cu cel mult un an de experiență de joc. Dintre aceștia, peste 66.6% se caracterizează fie ca începători, fie ca începători totali/fără niciun fel de experiență (A se vedea Figura 2).

¹<https://forms.gle/vfUQMxHWNigYraaJ7>

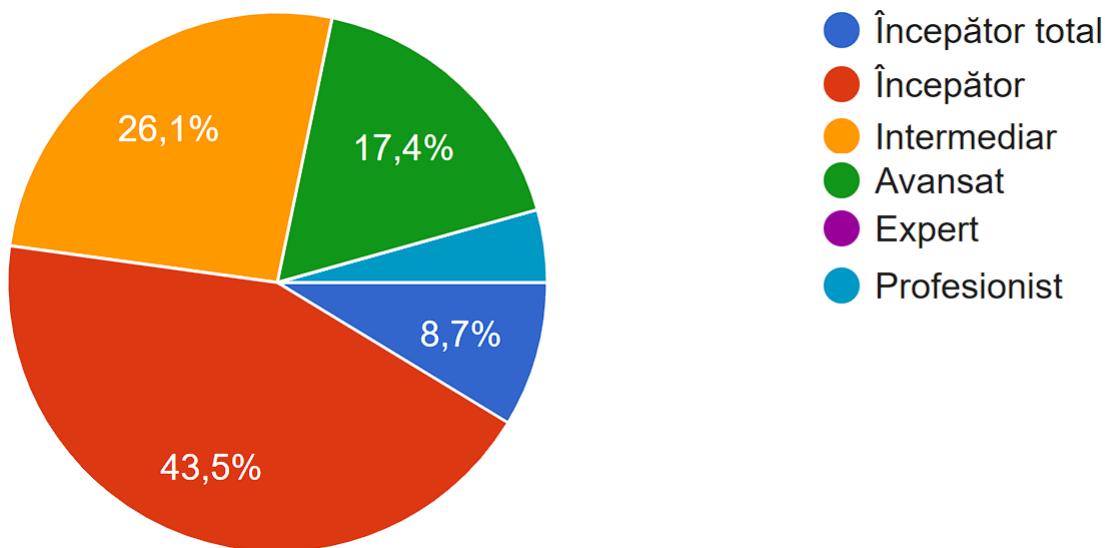


Figura 2: "Care ați estima că este nivelul dvs. de experiență la șah?"

- Un procent complet de 100% dintre participanți ar fi considerat utilă o aplicație simplă și intuitivă care să-i ajute să învețe șah, fie acum, fie în trecut.
- 93.5% dintre participanți consideră important ca o astfel de aplicație să fie gratis.
- 58.7% ar dori să nu fie necesară o conexiune la internet.
- 89.1% doresc tutoriale fundamentale, iar 82.6% preferă și tutoriale mai avansate.
- 78.3% consideră importantă existența adversarilor AI de diferite niveluri de dificultate.
- 69.6% spun că este importantă diversitatea AI-urilor și prin stilul de joc, nu doar prin nivelul de dificultate. Adițional, 100% dintre participanți sunt de părere că acest lucru le-ar îmbunătăți adaptabilitatea.

A se vedea Figura 3.

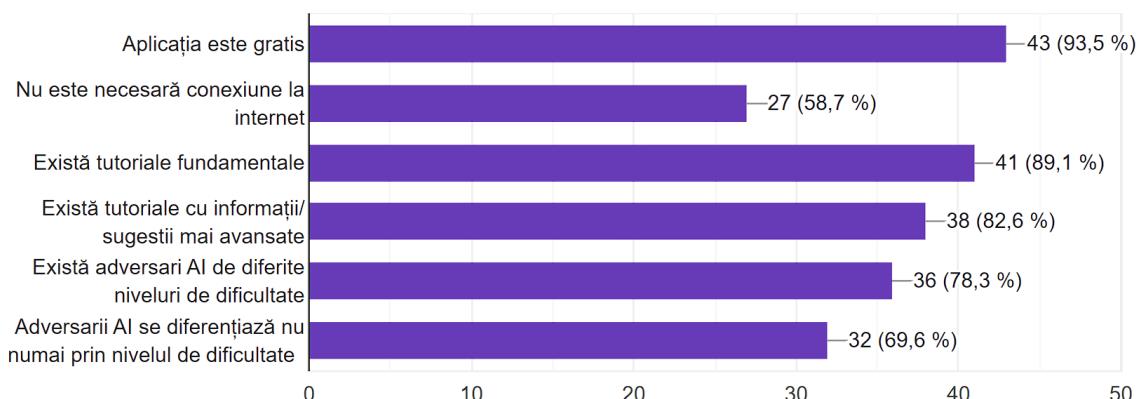


Figura 3: "Care dintre următoarele aspecte vi se par importante?"

- În final, 97.8% consideră că o opțiune de "hint", sugestie de mutare, ar fi utilă în cadrul aplicației.

După analizarea răspunsurilor, putem trage concluzia că există o cerere mare pentru o astfel de aplicație, fapt dat de familiaritatea sau intriga participantilor cu săhul, dar și de nivelul lor de experiență. Majoritatea funcționalităților existente au obținut păreri pozitive.

De asemenea, am decis să implementez și funcționalitatea de "hint", fiind potrivită pentru momente când utilizatorii se blochează și au nevoie de asistență în plus.

Așadar, lista completă de funcționalități este:

1. Nu este necesară o conexiune la internet.
2. Există efecte sonore de fundal și specifice anumitor acțiuni (mutarea pieselor, apăsarea butoanelor).
3. Se oferă acces la tutoriale fundamentale și avansate.
4. Există adversari AI de diferite niveluri de dificultate și cu diverse stiluri de joc.
5. În cadrul modului de joc împotriva AI, există opțiunea de "hint", sau sugestie de mutare.
6. Se poate schimba perspectiva tablei de joc (fie din partea pieselor albe, fie negre).

Pentru o modelare vizuală a cazurilor de utilizare ale sistemului, a se vedea Figura 4.

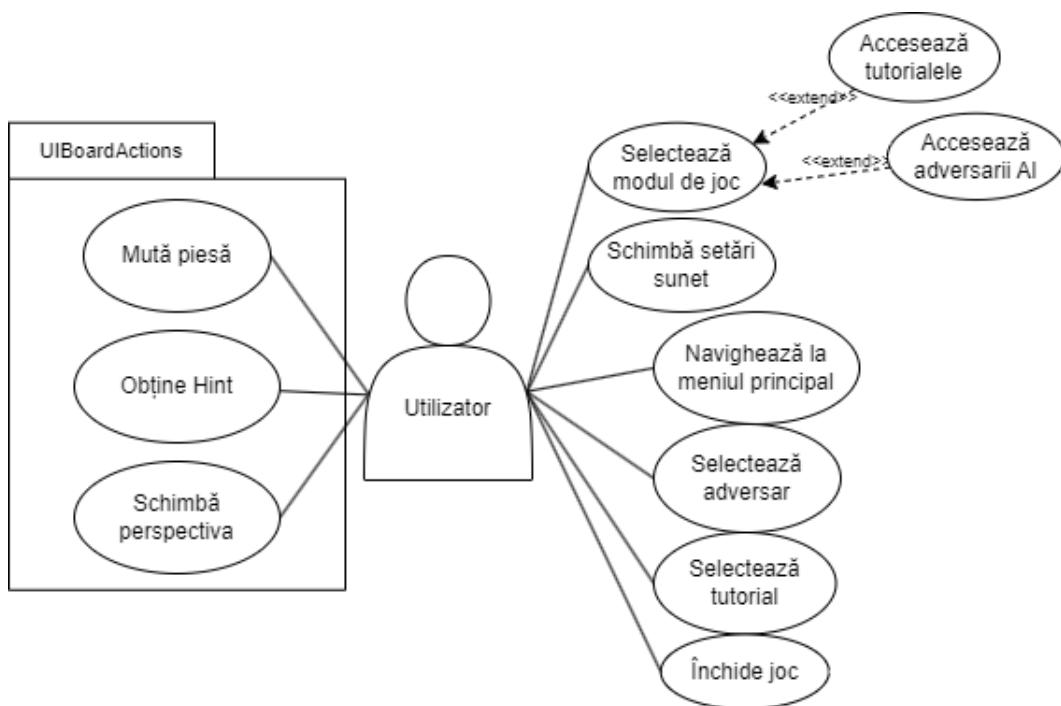


Figura 4: Diagrama Cazurilor de utilizare

3 STUDIU DE PIATĂ

Întorcându-ne temporar la formularul menționat în capitolul precedent, la întrebarea "Folosiți vreuna dintre următoarele pentru scopul de a învăța săh?", s-au obținut următoarele rezultate, vizibile în Figura 5.

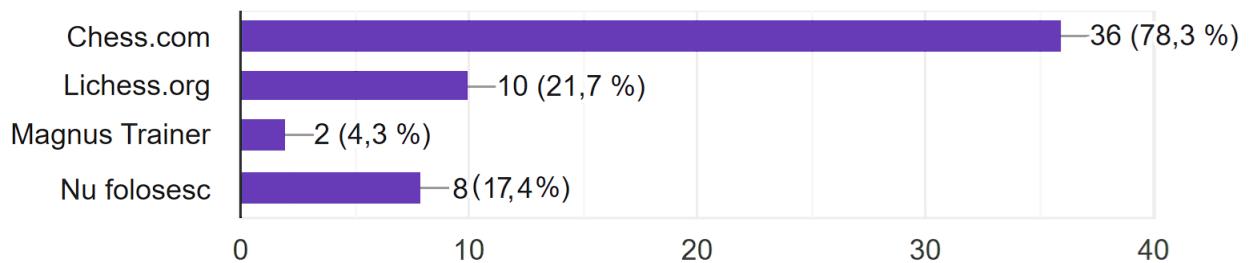


Figura 5: Cele mai populare platforme pentru învățare săh

Observăm o acaparare incontestabilă asupra pieței de către Chess.com și Lichess.org. Dar de ce este acest lucru? Ambele au în comun două funcționalități: prezența tutorialelor intuitive fundamentale și avansate, dar și faptul că sunt gratis, cu mențiunea că Chess.com oferă și abonamente plătite, dar există totuși numeroase funcționalități fără cost (Freemium). Acestea se diferențiază de Magnus Trainer prin gratuitate, și de alte variante pe piată precum Shredder Chess sau Lucas Chess prin interfață mai intuitivă. A se referi la Figurile 6, 7, 9 și 10.



Figura 6: Interfață Lichess.org [8]

Totuși, popularitatea Chess.com este cu mult peste Lichess.org. Acest fapt se datorează promovării active și organizării numeroaselor evenimente și parteneriate. De cele mai multe ori, acestea sunt acompaniate de... noi adversari AI în cadrul platformei, lucru pe care Lichess.org nu îl prezintă. Este adevărat, există motoare dezvoltate de comunitate sau boti experimentalii, dar acestea nu fac parte din procesul de învățare. Pe Chess.com, există un întreg ecosistem de boti ce se integrează armonios cu tutorialele. Tocmai din acest motiv, Chess.com își permite să implementeze funcționalități limitate doar pentru utilizatori cu abonament, cererea pentru produsul lor fiind atât de mare.

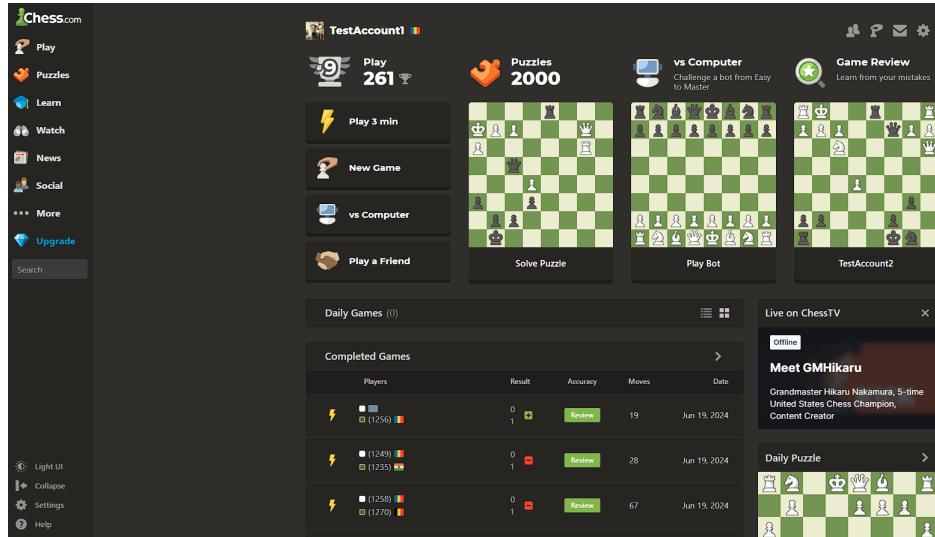


Figura 7: Interfață Chess.com [9]

În plus, Chess.com folosește propriul motor pentru acești boti, Komodo. Acesta a fost achiziționat recent în Mai 2018 și se distinge prin capabilitatea crescută de personalizare și configurare, făcându-l perfect pentru adversarii AI. Doar câțiva dintre acești boti se pot observa în Figura 8.

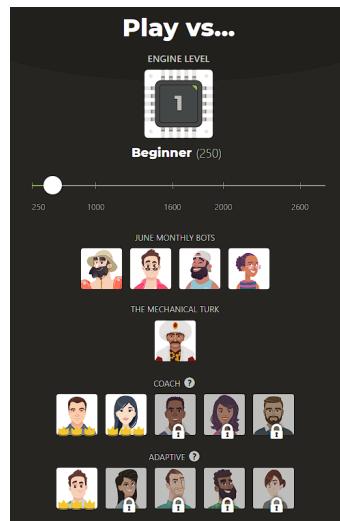


Figura 8: Adversarii AI pe Chess.com [9]

Pentru sumarul analizei soluțiilor similare pe piață, a se vedea Tabela 1.

Tabela 1: Soluții similare pe piață

Produs/Platformă	Gratuitate	Tutoriale intuitive	Adversari AI pentru învățare	Engine integrat pentru adversari AI	Disponibil Offline
Chess.com	Partial (Freemium)	DA	DA	DA (Komodo)	Partial (funcționalități limitate)
Lichess.org	DA	DA	NU (doar experimen-tali)	N/A	Partial
Magnus Trainer	Partial (Freemium)	DA	NU	N/A	Partial
Lucas Chess	DA	NU	DA	DA	DA
Shredder Chess	NU	NU	DA	DA (Shredder)	DA

Aplicația propusă combină toate caracteristicile prezente în tabel și profită de formula Chess.com de a utiliza motoarele de șah flexibile și personalizabile pentru a crea diversi adversari AI și a oferi o experiență plăcută pe parcursul învățării.

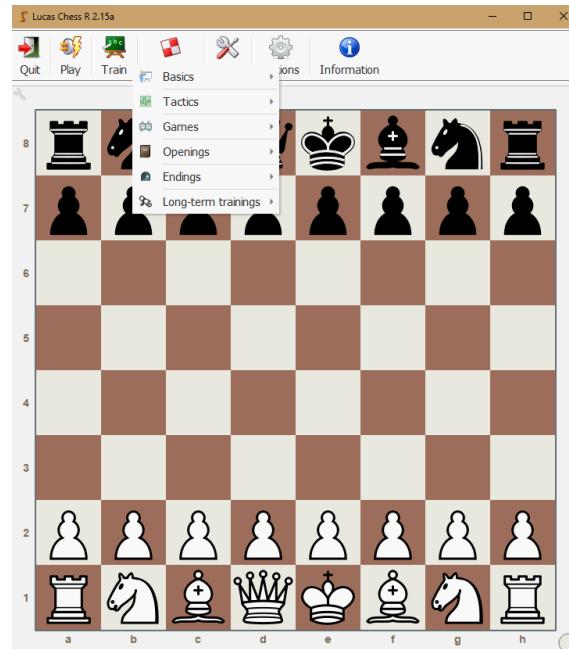


Figura 9: Interfață Lucas Chess (R) [10]



Figura 10: Interfață Shredder 13 Series for Linux [11]

4 SOLUȚIA PROPUȘĂ

Aplicația a fost dezvoltată în Unity, un motor de joc multiplatformă ce folosește C Sharp pentru scripting. Această alegere este dată de flexibilitatea în dezvoltare a interfeței grafice [12] și ușurința extensibilității jocului cu noi funcționalități. Având în vedere că jocul pe care îl abordăm este săhul și dorim câteva elemente de gamificare, concentrându-ne pe partea de algoritmică, mi s-a părut o unealtă mai mult decât potrivită.

Figura 11 expune arhitectura sistemului, ce se împarte în subsisteme și componente:

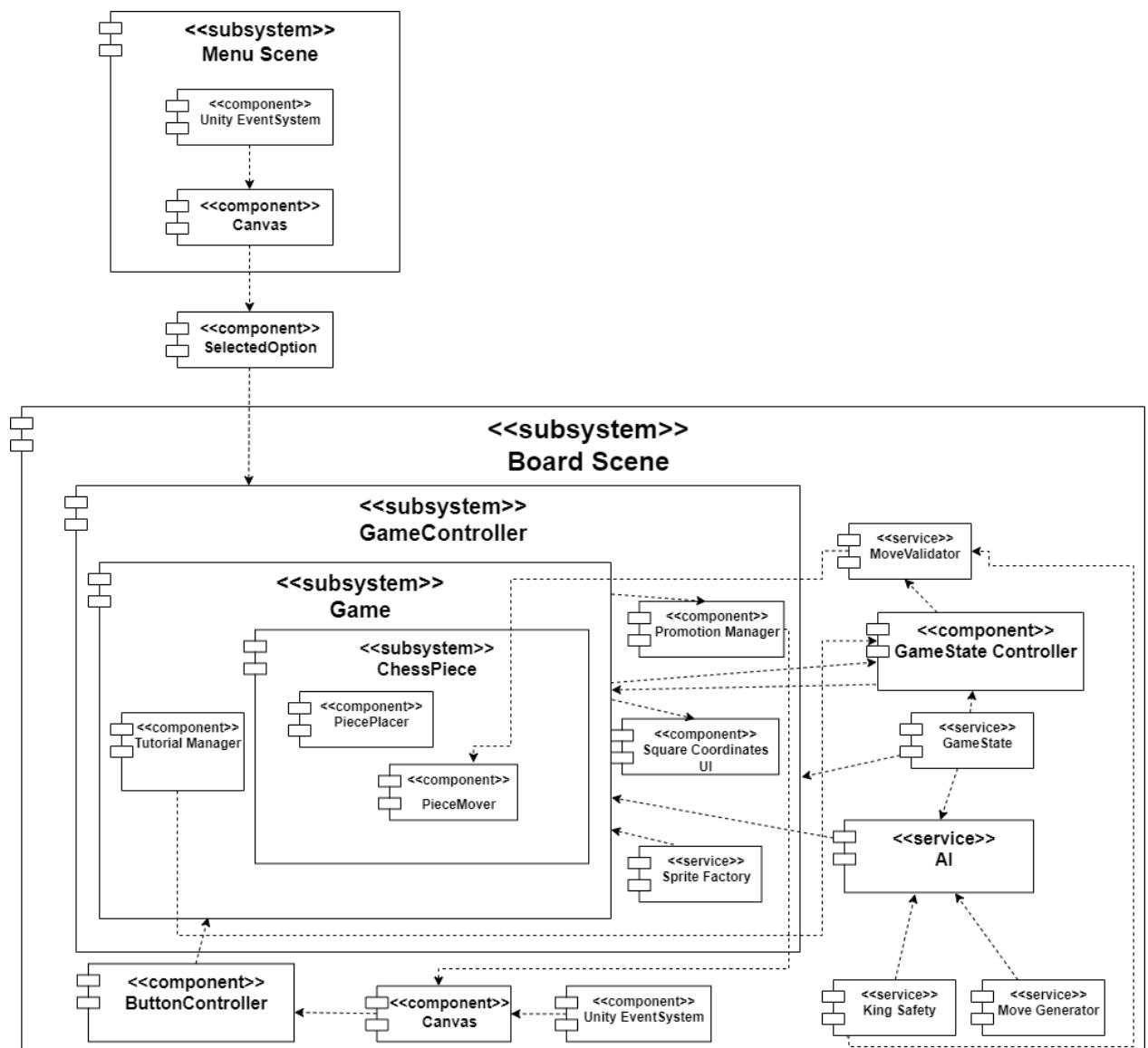


Figura 11: Diagrama De Componente

În cele ce urmează, vom discuta pe scurt despre fiecare componentă în parte: rolul ei și modul în care interacționează cu celealte componente în cadrul logicii aplicației:

1. **Menu Scene**: componenta care se ocupă de primul lucru pe care un utilizator îl vede atunci când lansează aplicația, și anume meniu principal.
Subcomponentele sale **Canvas** și **EventSystem** se ocupă de navigarea prin meniu cu ajutorul inputului de la utilizator.
În urma selectării unui tutorial sau al unui adversar AI, se trimit informația către componenta **SelectedOption**.
2. **SelectedOption**: un ScriptableObject, primește informații referitoare la ce mod de joc a fost selectat de utilizator, spre a fi utilizate de următoarea scenă, **Board Scene**, în cadrul componentei sale **GameController**.
3. **Board Scene**: Este hub-ul principal pentru logica aplicației, încapsulează toate celealte componente ce urmează.
4. **GameController**: Contine logica cu privire la generarea tablei de joc și a pieselor, precum și logică pentru interpretarea datelor primite de la **SelectedOption**.
5. **GameState Controller**: Se ocupă de gestionarea stării globale a tablei, utilizând componenta serviciu **GameState**. Această stare globală se furnizează și altor componente, la nevoie.
6. **Tutorial Manager** transmite starea tablei de joc corespunzătoare tutorialului necesar către **GameState Controller** și se folosește de metode proprii, dar și din **Game**, pentru a orchestra completarea acestuia.
7. **ButtonController**: cu ajutorul componentelor **EventSystem** și **Canvas**, oferă diferite funcționalități specifice butoanelor. Resetează poziția, schimbă perspectiva tablei, face o cerere pentru o mutare "hint" sau chiar indică schimbarea scenei înapoi la **MenuScene**, toate posibile comunicând cu componenta **Game**.
8. **Square Coordinates UI**: generează în interfața utilizator litere de la 'a' la 'z' și numere de la 1 la 8 pentru identificarea pătrățelelor tablei, în funcție de perspectiva de joc.
9. **Chesspiece**: un prefab cu ajutorul căruia **Game** generează o piesă de șah pe tablă, în funcție de tip și locație, folosind **Sprite Factory**, respectiv **PiecePlacer**.
10. **MoveValidator**: atunci când o piesă generată este mutată de către utilizator folosind drag&drop, această componentă este folosită de către **PieceMover** pentru a valida legalitatea mutării (e.g. se mută o piesă proprie, piesa respectă regulile de mișcare).
Se folosește de componentele **GameState Controller**, pentru acces la starea globală a jocului, și de **King Safety**, pentru verificarea logicii legată de șah la rege.
11. **Promotion Manager**: gestionează meniul de selectare a piesei dorite în caz de promovare a pionilor, în funcție de perspectiva de joc și poziția necesară pe tablă.
12. **AI**: o componentă ce primește o stare de joc arbitrară și returnează cea mai bună mutare în condițiile date. Folosită de **Game** atunci când este rândul adversarului AI.
Se folosește de serviciile **Move Generator** și **King Safety** pentru a construi arborele Minimax și a face căutarea.

Pe parcursul dezvoltării a fost folosită o abordare bottom-up, pornind de la componentele cele mai granulare. Acest lucru a facilitat testarea continuă în timpul realizării proiectului și descoperirea bug-urilor critice în timp util, când sistemul nu era încă foarte complex.

S-a început cu **ChessPiece**, **Sprite Factory**, **GameState**, **MoveValidator** și un **ButtonController** minimal.

Arhitectura a fost treptat extinsă cu **KingSafety**, **Promotion Manager**, **Square Coordinates UI** și funcționalități extinse pentru **GameState** și **GameStateController**: generarea stării jocului după un custom input, funcții de hashing pentru tabele de dispersie și refactorizarea codului într-un mod modularizat și ușor extensibil.

În stagiile finale s-au adăugat **Move Generator** și componentele **AI**, **Tutorial Manager** și **Menu Scene** au marcat ultimele adiții.

Pentru a urmări în mod detaliat procesul iterativ de dezvoltare, se poate consulta repository-ul¹ de GitHub.

¹<https://github.com/luksy26/Unity-Chess>

5 DETALII DE IMPLEMENTARE

5.1 GameState

Această componentă reprezintă "inima" aplicației, cu ajutorul ei putem reprezenta starea jocului, genera piesele în pozițiile corespunzătoare și reprezenta nodurile în arborele Minimax folosit de motoarele de săh.

Algorithm 1: Pseudocod pentru clasa GameState

```
1 class GameState:  
2     /* Fields */  
3     char[] boardConfiguration  
4     char whoMoves  
5     char enPassantFile  
6     int enPassantRank  
7     int[】 kingsLocations  
8     bool[】 castlingRights  
9     int moveCounter50Move  
10    int moveCounterFull  
11    /* Methods */  
12    int GetHashCode()  
13    bool Equals()  
14    void MakeMove(move)  
15    void UnmakeMove(move)  
16    string ToString()
```

Putem folosi această oportunitate să discutăm despre FEN (Forsyth-Edwards Notation). Acesta este un standard pentru a codifica pozițiile de săh și toate detaliile despre ele într-un singur sir de caractere.

Clasa **GameState** poate reține toate aceste informații în câmpuri relevante. Există și metode pentru efectuarea unei anumite mutări pe GameState-ul descris de câmpuri, dar și de "revocare" a mutării: lucru folositor în algoritmii recursivi de căutare.

Metodele **GetHashCode()** și **Equals()** sunt suprascrisse pentru adăugarea instanțelor de GameState în tabele de dispersie folosite de **Game** și **AI**, vom vedea mai târziu în cadrul acestui capitol de ce.

Pentru a observa structura FEN-urilor, a se consulta în Figura 12, poziția initială.

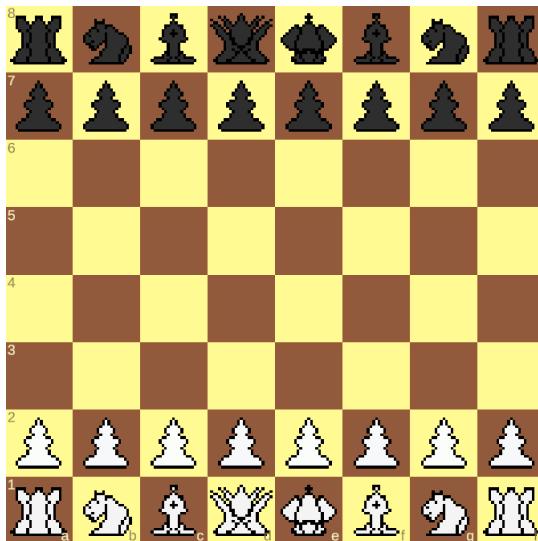


Figura 12: FEN rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1

FEN-ul constă în 6 câmpuri, separate prin spațiu:

1. **Plasamentul pieselor:** un sir de caractere pentru toate cele 8 linii de pe tabla de șah, separate prin '/'.

Format din litere mici (semnificând piese negre), litere mari (semnificând pisele albe), și numere de la 1 la 8 pentru a reprezenta în mod condensat spațiile libere dintre piese. Semnificația literelor este următoarea:

- 'r': tură (rook)
- 'n': cal (knight)
- 'b': nebun (bishop)
- 'q': regină (queen)
- 'k': rege (king)
- 'p': pawn (pawn)

De exemplu, o linie descrisă de sirul de caractere "8" este o linie goală, iar o linie descrisă de sirul de caractere "p6p" reprezintă doi pioni negri la stânga și la dreapta, la 6 spații goale depărtare unul de celălalt.

2. **Jucătorul la mutare:** 'w' pentru alb și 'b' pentru negru.

3. **Drepturile de rocadă:** litere mari pentru drepturile jucătorului cu pisele albe și litere mici pentru jucătorul cu pisele negre. Semnificația literelor este următoarea:

- 'k': rocadă scurtă (kingside castle)
- 'q': rocadă lungă (queenside castle)

Dacă nu există niciun drept de rocadă, câmpul se reprezintă cu un '-'.

4. **Coordonatele pe tablă ale țintei en-passant.** Atunci când un pion se deplasează două pătrate, acest câmp va conține coordonatele pătratului din "spatele" pionului. (e.g. pentru mutarea e2-e4, acest câmp devine e3). Este important de menționat că nu este necesar ca pătratul să poată fi atacat de jucătorul la mutare, câmpul conține informația orice ar fi. Dacă niciun pion nu s-a deplasat două pătrate la tura imediat precedentă, atunci câmpul este reprezentat de '-'.
5. **Contorul pentru jumătăți de mutare (sau pentru regula de 50 de mutări):** acesta începe de la 0 și este incrementat la fiecare mutare ce nu presupune mutarea unui pion sau o capturare. Altfel, este resetat înapoi la 0. Atunci când ajunge la 100, jocul este declarat o remiză.
6. **Contorul pentru mutări complete:** acesta începe de la 0 și este incrementat de fiecare dată când jucătorul cu piesele negre face o mutare, indiferent de tipul mutării.

5.2 GameStateManager

Această componentă gestionează un GameState global și conține câteva metode relevante:

Algorithm 2: Pseudocod pentru clasa GameStateManager

```

1 class GameStateManager:
    /* Fields */                                                 */
2     static GameStateManager Instance {get; private set}
3     GameState globalGameState
4     string defaultFEN
    /* Methods */                                              */
5     void GenerateGameState(string inputFEN)
6     GameConclusion GetDrawConclusion(gameState, gameStates)
7     GameConclusion GetMateConclusion(gameState)

```

Clasa urmează modelul Singleton și expune globalGameState către alte componente. Se implementează două metode (liniile 6 și 7) ce detectează finalul jocului pentru un anumit GameState.

GetDrawConclusion acceptă un argument adițional, gameStates, care este un Hashtable cu toate GameState-urile atinse precedent în cadrul meciului curent de sah. Motivul pentru care avem nevoie să reținem aceste date este pentru că, în sah, dacă se atinge o poziție identică de 3 ori (nu neapărat consecutiv), jocul este declarat o remiză. Astfel, putem vedea și motivul implementării **GetHashCode()** și **Equals()** în cadrul componentei **GameState** de mai devreme.

Partea cea mai interesantă este metoda **GenerateGameState()**, care generează gameState-

ul global în funcție de un FEN primit ca și input, lucru foarte folositor pentru generarea pozițiilor din cadrul tutorialelor. Se pot genera poziții complexe doar printr-un simplu apel de metodă, a se vedea Figura 13.



Figura 13: FEN r4rk1/pp2bfff/4p3/2p1P1P1/3p1q1P/2P5/PPQ2P2/1K1R3R b - - 1 18

5.3 KingSafety

Această componentă este o clasă statică și o implementare eficientă a metodelor sale este absolut crucială pentru o performanță bună a generării mutărilor de către **MoveGenerator**.

Algorithm 3: Metodele Clasei KingSafety

```

1 class KingSafety:
2     static List <int> GetKingAttackers(gameState, move)
3     static bool IsKingSafeFromBishop(kingLocation, bishopLocation, move)
4     static bool IsKingSafeFromRook(kingLocation, rookLocation, move)
5     static bool IsKingSafeFromKnight(kingLocation, knightLocation)
6     static bool IsKingSafeFromPawn(pawnLocation, gameState, move)
7     static bool IsKingSafeFromDiagonalDiscovery(gameState, move)
8     static bool IsKingSafeFromLineDiscovery(gameState, move)
9     static bool IsKingSafeAt(newKingLocation, gameState, move)

```

Metoda **GetKingAttackers()** întoarce o listă cu pozițiile pieselor ce atacă regele jucătorului la mutare. Deoarece într-un joc normal de șah nu este posibil ca regele să fie atacat de mai mult de două ori¹, funcția își oprește verificările în cazul în care sunt găsiți doi atacatori. Argumentul **move** este optional și indică că verificarea se face după ce este făcută respectiva mutare pe tablă.

¹<https://chess.stackexchange.com/questions/18064/is-triple-check-possible>

Metodele de tipul **IsKingSafeFromPiece()** verifică în mod explicit dacă o piesă încă atacă regele, fiind dată o mutare optională care ar putea rezolva acest atac (fie prin blocare, fie prin capturare). Dacă aceasta nu este furnizată, se verifică dacă regele s-a mutat astfel încât să nu mai fie atacat. Toată logica din cadrul acestor funcții se face lucrând strict pe coordonatele pieselor și nu prin bucle. Pentru o înțelegere mai detaliată, a se consulta capitolul **Anexe**, Figura 41. Pentru un exemplu vizual, a se vedea Figura 14.

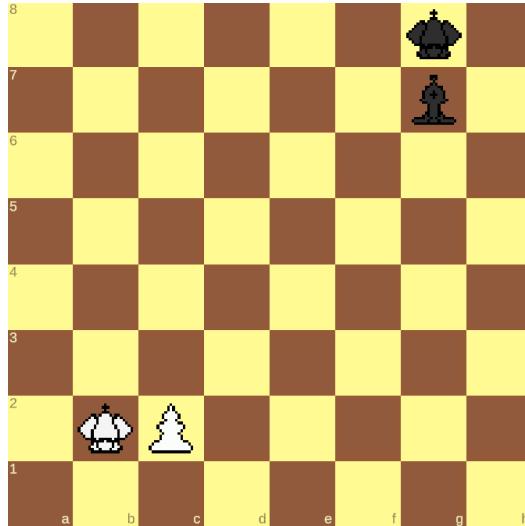


Figura 14: IsKingSafeFromBishop(b2, g7, c2c3) este True, pionul blocând diagonala nebunului

Cele două metode de tipul **IsKingSafeFrom*Discovery()** verifică dacă în urma unei mutări regele intră în șah prin descoperire (o piesă fiind mutată), caz în care mutarea nu ar fi legală. Pentru un exemplu vizual, a se vedea Figura 15.

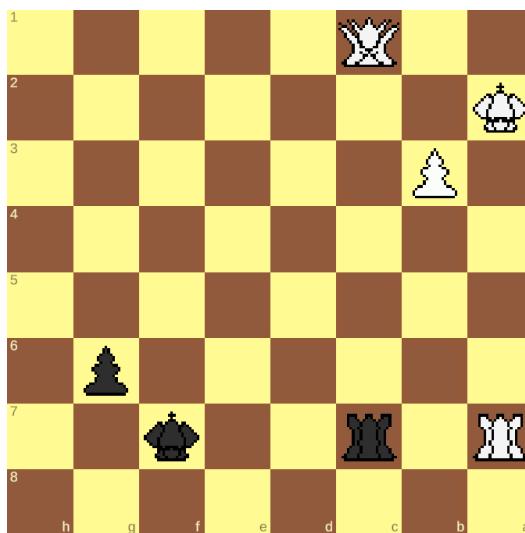


Figura 15: IsKingSafeFromLineDiscovery(f7, c7c1) este False, tura de la a7 ar ataca regele

În final, metoda **IsKingSafeAt()** verifică dacă regele ar fi atacat într-o locație arbitrară și este folositoare atunci când se face o mutare cu regele (sau rocada).

Vom vedea când exact sunt folosite metodele menționate în următoarea secțiune.

5.4 MoveGenerator

Această componentă ajută la generarea propriu-zisă a arborelui Minimax, plecând de la un GameState inițial și generând mutările legale. Pentru fiecare mutare în parte, după ce se face mutarea pe tablă și obținem alt GameState, putem folosi din nou **MoveGenerator**.

Algorithm 4: Metodele clasei MoveGenerator

```
1 class MoveGenerator:  
2     static List <IndexMove> GetLegalMoves(gameState)  
3     static void AddLegalPawnMoves(gameState, oldLocation, attacker, moveList)  
4     static void AddLegalBishopMoves(gameState, oldLocation, attacker, moveList)  
5     static void AddLegalKnightMoves(gameState, oldLocation, attacker, moveList)  
6     static void AddLegalRookMoves(gameState, oldLocation, attacker, moveList)  
7     static void AddLegalQueenMoves(gameState, oldLocation, attacker, moveList)  
8     static void AddLegalKingMoves(gameState, oldLocation, attackers, moveList)
```

Fiecare dintre metodele **AddLegalPieceMoves()** este folosită de metoda principală, **GetLegalMoves()**, în funcție de tipul piesei jucătorului la mutare, pentru toate piesele acestuia. În final o listă cu toate mutările generate este întoarsă de metodă.

Logica este următoarea:

Algorithm 5: Metoda GetLegalMoves(gameState)

```
1 attackers = GetKingAttackers(gameState) ; doubleCheck = (attackers.Count == 2)  
2 attacker = null  
3 if attackers.Count > 0 then  
4     attacker = attackers[0]  
5 moveList = new List<IndexMove>()  
6 foreach piece in currentPlayerPieces do  
7     if doubleCheck and piece.type != king then  
8         continue // only the king can resolve a double check  
9     oldLoc = piece.location  
10    switch piece.type do  
11        Case pawn: AddLegalPawnMoves(gameState, oldLoc, attacker, moveList)  
12        Case bishop: AddLegalBishopMoves(gameState, oldLoc, attacker, moveList)  
13        Case knight: AddLegalKnightMoves(gameState, oldLoc, attacker, moveList)  
14        Case rook: AddLegalRookMoves(gameState, oldLoc, attacker, moveList)  
15        Case queen: AddLegalQueenMoves(gameState, oldLoc, attacker, moveList)  
16        Case king: AddLegalKingMoves(gameState, oldLoc, attackers, moveList)  
17    return moveList
```

Este de menționat că dacă regele este atacat de două ori (lucru posibil doar în caz de săh prin descoperire), nu se ia în considerare nicio mutare ce nu este făcută chiar de regele atacat (linia 7). În cazul în care orice altă piesă s-ar muta, nu ar putea rezolva două atacuri simultane. Dacă presupunem prin absurd că acest lucru s-ar putea face, atunci piesa ar trebui să blocheze/captureze (în cazul de atac din partea cailor) două piese, deci să rezolve două căi de atac către un rege staționar în mod simultan, ceea ce nu este posibil. Pentru un exemplu vizual, a se vedea Figura 16.

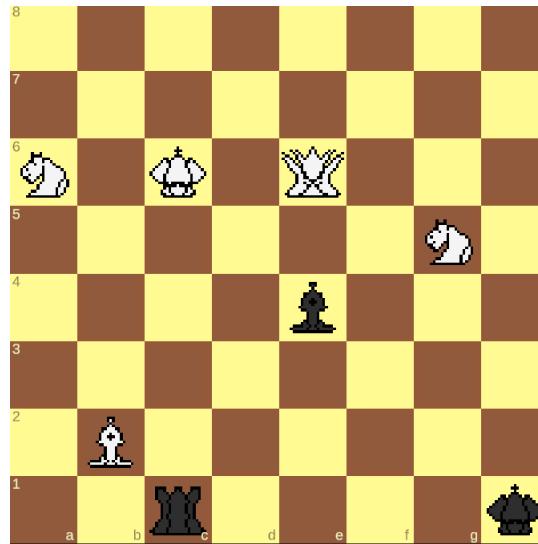


Figura 16: Singurele mutări care rezolvă ambele atacuri (nebun și tură) sunt ale regelui alb

În cazul generării mutărilor legale pentru rege, logica este următoarea:

Algorithm 6: AddLegalKingMoves(gameState, oldLocation, attackers, moveList)

```

// first check regular, one-square moves
1 foreach move in kingMovesFrom(oldLocation) do
2     newLocation = move.DestinationSquare
3     CheckSafeFromAttackers(newLocation, attackers) and
        CheckKingSafeAt(newLocation)
4     moveList.Add(move)

// now check castling moves
5 if attackers.Count > 0 then
6     // can't castle while in check
    return

7 getLongCastleMove(oldLocation, gameState)
8 getShortCastleMove(oldLocation, gameState)
9 checkKingSafety(longCastleMove)
10 moveList.Add(longCastleMove)
11 checkKingSafety(shortCastleMove)
12 moveList.Add(shortCastleMove)

```

Pentru mutări normale, dacă regele este atacat, ne asigurăm că vechii atacatori au fost rezolvăți înainte de a verifica că nu există și alte piese ce atacă regele în noua locație.

Pentru rocade, întâi se verifică dacă există atacatori, caz în care mutarea nu ar fi legală (nu se poate face rocadă din săh). Apoi, **checkKingSafety()** verifică folosind **IsKingSafeAt()** dacă regele ajunge într-o locație sigură și nu se face rocadă "prin" săh (da, nici acest lucru nu este legal). Pentru o analiză mai detaliată a codului, a se consulta capitolul **Anexe**, Figura 42.

În cazul generării mutărilor legale pentru celelalte piese, logica este următoarea:

Algorithm 7: AddLegalPieceMoves(gameState, oldLocation, attacker, moveList)

```

1 foreach move in PieceMovesFrom(oldLocation) do
2     newLocation = move.DestinationSquare
        // check if the move resolves the attacker
3     if IsSafeFromAttackers(newLocation, attackers) then
        // check if the piece is not pinned to the king
4         if IsKingSafeFromDiagonalDiscovery(gameState, move) and
            IsKingSafeFromLineDiscovery(gameState, move) then
                moveList.Add(move)
5

```

De data aceasta nu se mai folosește **IsKingSafeAt()**, tot ce avem nevoie să verificăm este dacă atacatorii sunt rezolvăți, exemplu Figura 17, și dacă piesa nu este "lipită" de rege (fapt ce înseamnă că mutarea respectivă ar descoperi un atac), exemplu Figura 18.

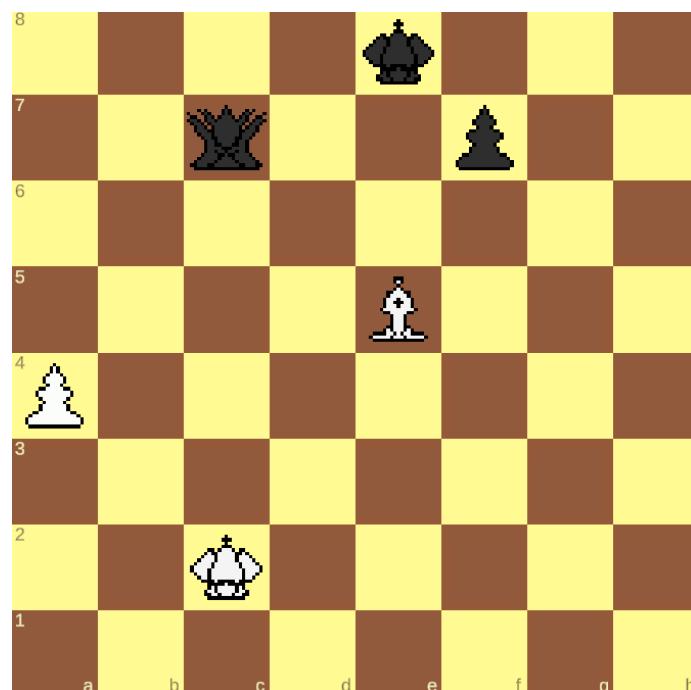


Figura 17: e5c7 și e5c3 sunt mutări legale, ambele rezolvă atacul reginei de la c7

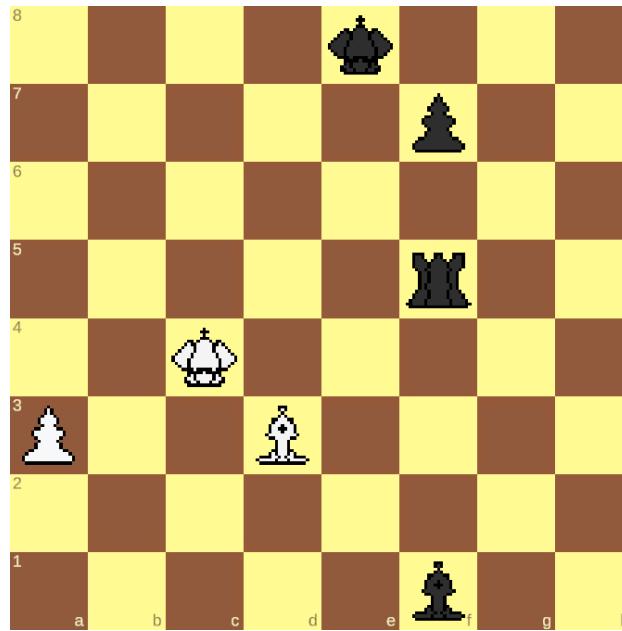


Figura 18: d3f5 și d3c2 nu sunt mutări legale, acestea ar descoperi atacul nebunului de la f1

5.5 Game

Aceasta este componenta care se ocupă cu generarea elementelor în interfața grafică și este un hub de funcționalități folosite de alte componente: **PieceMover** trimite mutarea făcută de către utilizator și **ButtonManager** solicită o mutare "hint" sau o schimbare a perspectivei tablei. În plus, **Game** se ocupă și cu gestionarea concluziei jocului, afișarea de prompt-uri către utilizator și efectuarea de mutări provenite de la **AI**.

Algorithm 8: Pseudocod pentru clasa Game

```

1 class Game:
2     GameObject[,] currentPieces
3     List<GameObject> blackPieces, whitePieces
4     char currentPlayer, AIPlayer
5     float timeToMove
6     bool timeExpired
7     Hashtable gameStates
8     async void GeneratePosition()
9     async void MovePiece(move)
10    MoveEval SolvePosition()
11    void HandleGameState(gameState, gameStates)
12    void SwapPlayer()
13    void SwapPerspectives()
14    void DestroyPosition()
```

Algorithm 9: Pesudocod pentru metoda GeneratePosition()

```
1 SquareCoordinatesUI.GenerateFilesAndRanks(playerPerspective)
2 globalGameState = GameManager.Instance.globalGameState
3 gameStates.Add(globalGameState, 1)
4 foreach piece in globalGameState.boardConfiguration do
5     updateCurrentPieces()
6     updateBlackPieces()
7     updateWhitePieces()
8 HandleGameState(globalGameState, gameStates)
9 if currentPlayer == AIPlayer then
10    timeExpired = false
11    StartCoroutine(MoveTimerCoroutine(timeToMove))
12    await Task.Run(() => bestMove = SolvePosition())
13    MovePiece(bestMove)
```

GeneratePosition() folosește componentele **SquareCoordinatesUI** și **GameManager** pentru a genera tabla de șah, coordonatele ei, și piesele în pozițiile corespunzătoare, stocându-le în structuri interne de date ca și *GameObjects*. Se gestionează concluzia jocului și, dacă AI-ul trebuie să joace prima mutare, se inițiază un timer. Se folosește metoda **SolvePosition()** pentru a obține mutarea de făcut pe tablă, apoi se face această mutare folosind metoda **MovePiece()**.

Algorithm 10: Pesudocod pentru metoda MovePiece(move)

```
1 UpdateInternalData()
2 if move.type == promotion then
3     newPiece = await PromotionManager.GeneratePromotionMenu(move.newFile)
4     if newPiece == null then
5         return
6     UpdatePieceSprite(move, newPiece)
7 SwapPlayer()
8 gameState = GameManager.Instance.globalGameState
9 gameState.makeMove(move)
10 ++gameStates[gameState]
11 HandleGameState(gameState, gameStates)
12 if currentPlayer == AIPlayer then
13    timeExpired = false
14    StartCoroutine(MoveTimerCoroutine(timeToMove))
15    await Task.Run(() => bestMove = SolvePosition())
16    MovePiece(bestMove)
```

MovePiece() actualizează structurile de date interne pentru piese în funcție de mutarea făcută. În cazul special de promovare a pionilor, se folosește componenta **PromotionManager** și se generează UI-ul pentru promovare. Sprite-ul piesei se actualizează în funcție de alegerea făcută de utilizator sau, în caz că s-a selectat opțiunea de anulare, toată metoda își termină execuția, mutarea nu va mai fi făcută.

Urmează schimbarea jucătorului la mutare și actualizarea GameState-ului global și a tabelei de dispersie. Se gestionează concluzia jocului cu HandleGameState(), care este doar o metodă ce se folosește de GetDrawConclusion() și GetMateConclusion() din cadrul componentei **GameManager**. Aceasta afișează promptul corespunzător în caz de final de joc și nu mai permite noi mutări până la începerea unuia nou.

În final, dacă acum este rândul AI-ului, se pornește timer-ul, se caută cea mai bună mutare și se cheamă din nou **MovePiece()**. Apelul **SwapPlayer()** este foarte important pentru a nu intra în ciclu infinit.

Despre **SolvePosition()** vom discuta la secțiunea de **Iterative Deepening Search**.

5.6 Perft

Pentru a verifica logica din **MoveGenerator** și **KingSafety**, vom defini RunPerft(gameState) în componenta **Game**, ce se va folosi de o componentă de testare **PositionCounter**.

În esență, Perft² este un utilitar de debugging care calculează numărul total de poziții posibile pentru un GameState dat, până la o anumită adâncime. De exemplu, Perft(4, startingPos) va avea ca rezultat 197281 de poziții "frunză".

Logica pentru **PositionCounter** este următoarea:

Algorithm 11: Pseudocod pentru clasa PositionCounter

```

1 class PositionCounter:
2     static long SearchPositions(gameState, depth) begin
3         if depth == maxDepth then
4             return 1
5         long sum = 0
6         legalMoves = GetLegalMoves(gameState)
7         foreach move in legalMoves do
8             gameState.MakeMove(move)
9             sum += SearchPositions(gameState, depth + 1)
10            gameState.UnmakeMove(move)
11        return sum

```

²<https://www.chessprogramming.org/Perft>

Dacă se ajunge la adâncimea maximă, se returnează 1. Altfel, se generează toate mutările legale folosind componenta **MoveGenerator** și, pentru fiecare mutare, se caută recursiv poziții la o adâncime mai mare. La întoarcerea din recursivitate se cheamă **UnmakeMove()** pentru pregătirea gameState-ului pentru următoarea mutare din listă.

Este important de menționat că nu oprim căutarea decât la adâncimea maximă. Nu se iau în considerare reguli speciale de remiză prin repetiție sau material insuficient (e.g. ramân doar 2 regi pe tablă). Aceasta este convenția și are sens din două motive:

- Scopul nostru principal este să testăm corectitudinea generării mutărilor, nu logica de remiză.
- Se salvează putere computațională.

Pentru datasetul ³ de testare au fost folosite 6838 de FEN-uri cu valori precalculate ale Perft până la o adâncime de 6. Având în vedere că valorile pot ajunge la ordinul miliardelor pentru Perft(6), toate pozиїile din dataset au fost verificate până la Perft(4). Ulterior, 136 dintre acestea au fost verificate pentru Peft(5).

Printre numeroasele bug-uri găsite cu ajutorul Perft se numără:

- Verificarea strictă > 0 în loc de ≥ 0 pentru generarea mutărilor de capturare folosind pionii. Astfel, erau omise mutările când pionii capturau o piesă de pe coloana 0 a tablei.
- Omiterea verificării unei condiții legate de regula specială "en-passant", ce ducea la o incosistență între tabla logică, globală, și cea reținută de **Game** pentru afișarea grafică.

În final, toate rezultatele obținute au fost corecte. Metoda definită pentru testare este disponibilă la capitolul **Anexe**, Figura 43.

³<https://github.com/elcabesa/vajolet/blob/master/tests/perft.txt>

5.7 Funcții statice de evaluare

Pe când în capitolul precedent doar am numărat nodurile frunză, tranziția spre algoritmul Minimax presupune asignarea unei scor fiecărui dintre ele. Vom considera un scor pozitiv ca fiind în avantajul jucătorului cu piesele albe și un scor negativ ca fiind în avantajul jucătorului cu piesele negre, un scor de 0 însemnând o poziție balansată. În plus, în caz că poziția se află într-un stadiu final de joc (șah mat, pat, remiză prin material insuficient etc.), stabilim următoarele convenții:

- O poziție cu mat pentru alb va avea scorul de $+1000 - \text{depth}$.
- O poziție cu mat pentru negru va avea scorul de $-1000 + \text{depth}$.
- O poziție în care se ajunge la remiză va avea scorul 0.

Unde 'depth' este adâncimea în cadrul arborelui Minimax unde a fost găsită poziția. Facem acest lucru pentru a diferenția între poziții care duc la șah-mat în funcție de adâncime, este evident că AI-ul va prefera poziția cu adâncimea mai mică, pentru a câștiga mai repede. Analog, va prefera poziția cu adâncime mai mare în caz că este în dezavantaj, pentru a pierde mai lent (și eventual dând mai multe oportunități utilizatorului să facă greșeli în drum spre mat).

Scorul este calculat pe o poziție statică, de către o funcție de evaluare. Folosim cuvântul "static" pentru a evidenția faptul că nu generăm mutări pentru a investiga cum ar putea arăta pozițiile viitoare, pur și simplu ne folosim de niște euristici ce sunt aplicate doar pe poziția curentă și calculăm scorul. În dezvoltarea motoarelor acestei aplicații, se disting două tipuri de funcții de evaluare.

Acstea au în comun câteva lucruri:

- Fiecăruia tip de piesă îi este asignat o valoare absolută
- Această valoare este ajustată în funcție de diferite euristici pentru a obține o valoare relativă
- Scorul poziției este dat de suma valorilor relative ale tuturor pieselor albe, din care se scade suma valorilor relative ale tuturor pieselor negre
- În majoritatea cazurilor, pasul final este o ajustare minimală a scorului în funcție de ce jucător este la mutare.

Conversia din valoare absolută în valoare relativă a piesei este factorul ce diferențiază funcțiile de evaluare între ele.

5.7.1 Functii bazate pe pătrate controlate

Acest tip de funcții iau în vedere natura pătratelor controlate de o anumită piesă pe tablă. Ca și factori considerați putem avea:

- Numărul de pătrate controlate
- Pătrate controlate unde se află o piesă proprie/inamică (piesa va fi protejată/atacată). În acest caz se poate face oferă un bonus pentru o piesă de valoare mică ce atacă/apără o piesă de valoare mai mare (e.g. un pion atacă o regină, un cal apără o tură etc.)
- Bonus pentru lanțuri de pioni (ioni care se apără reciproc)
- Pentru rege, se aplică o penalizare dacă acesta are o mobilitate crescută (deci apără/atacă alte piese) în deschidere. Atunci când se tranziționează în finalul meciului, când se află mai puține piese pe tablă, penalizarea se poate transforma în bonus
- Bonus pentru pătrate controlate aproape de centrul tablei, cu cât mai aproape cu atât mai mare bonusul
- Bonus pentru pătrate controlate aproape de regele inamic, cu cât mai aproape cu atât mai mare bonusul

Acest tip de funcție este folosit pentru **Alv1**, **Alv2**, **Alv3**, **Alv6** și **Alv7**.

5.7.2 Functii bazate pe amplasarea pieselor

Acest tip de funcții este mult mai simplu, se bazează pe PST⁴ (Piece Square Tables). PST sunt structuri de date predefinite ce indică, pentru fiecare tip de piesă, o ajustare a valorii absolute într-o valoare relativă bazată strict pe poziționarea piesei pe tablă.

De exemplu, PST-ul pentru pioni se poate vedea în Figura 19. Valorile din PST în acest caz descurajează cei doi pioni central să rămână nemăscăti, încurajează avansul în teritoriul inamic (din nou, controlul centrului este mai important), cu un bonus masiv pentru pionii ce sunt pe cale să promoveze.

Deoarece regii nu sunt niste piese atât de unidimensionale, vor avea două PST-uri: unul pentru deschidere și unul pentru endgame. În funcție de numărul de piese rămase pe tablă, vom interpola între cele două tabele folosind următoarea formulă:

$$endgameStage = \frac{\max(0, noPiecesLeft - endgameTransition)}{32 - endgameTransition}$$

$$trueValue = openingPST \cdot (1 - endgameStage) + endgamePST \cdot endgameStage$$

endgameTransition este o constantă și poate fi modificată, reprezintă momentul (descriș de numărul de piese rămase pe tablă) când *trueValue* depinde doar de *endgamePST*.

⁴https://www.chessprogramming.org/Piece-Square_Tables

0	0	0	0	0	0	0	0
50	50	50	50	50	50	50	50
10	10	20	30	30	20	10	10
5	5	10	25	25	10	5	5
0	0	0	20	20	0	0	0
5	-5	-10	0	0	10	-5	5
5	10	10	-20	-20	10	10	5
0	0	0	0	0	0	0	0

Figura 19: Ajustări PST pentru pioni

Este de menționat faptul că există un singur set de PST-uri, valorile pentru piesele negre putând fi obținute prin simpla oglindire a tabelului.

Acest tip de funcție este folosit pentru **Alv4** și **Alv5**.

5.8 Minimax cu Alpha Beta Pruning

În cazul nostru, algoritmul clasic Minimax presupune existența a doi jucători: un minimizator (jucătorul cu piesele negre) și un maximizator (jucătorul cu piesele albe).

Rădăcina arborelui Minimax este reprezentată de poziția de evaluat, nivelul 0. Pornind de la acest nod, se calculează valorile nodurilor copil, dintre care se alege scorul cel mai bun pentru jucătorul de pe nivelul 0. Bineînțeles, acest lucru ar presupune doar o adâncime 1.

Astfel, pentru o adâncime maximă *maximumDepth*, pentru a calcula valoarea unui nod copil, acesta va fi tratat ca rădăcina propriului său arbore Minimax cu o adâncime incrementată cu 1. Același lucru se procedează cu celelalte noduri copil, cu copiii copiilor etc., până când rădăcina arborelui de calculat este la *maximumDepth*.

De asemenea, chiar dacă rădăcina nu este la *maximumDepth*, ne putem opri mai devreme dacă se ajunge la o concluzie a jocului și nu se mai pot genera copii (i.e. nu mai există mutări legale).

Algoritmul este următorul:

Algorithm 12: Pseudocod pentru algoritmul MiniMax(gameState, depth)

```
1 if depth == maximumDepth then
2   return StaticPositionEvaluator(gameState)
3 conclusion = GameManager.Instance.GetGameConclusion(gameState)
4 if conclusion == Draw then
5   return 0
6 else
7   if conclusion == Mate then
8     if gameState.whoMoves == 'w' then
9       return -1000 + depth
10    else
11      return 1000 - depth
12 float bestScore = worstValueForCurrentPlayer
13 moveList = GetLegalMoves(gameState)
14 foreach move in moveList do
15   gameState.MakeMove(move)
16   float score = MiniMax(gameState, depth + 1)
17   gameState.UnmakeMove(move)
18   if score is better than bestScore then
19     bestScore = score
20 return bestScore
```

Bineînțeles, acești arbori pot deveni destul de mari. Presupunând un factor mediu de ramificare de 30, la adâncimea 5 putem ajunge deja la peste 24 de milioane (30^5) de noduri frunză. Principalul mod în care putem reduce acest număr fără să ignorăm în mod nejustificat noduri este să folosim **Alpha-Beta Pruning**.

Alpha-Beta Pruning presupune ignorarea anumitor ramuri din arborele Minimax folosind două valori, Alpha și Beta. În cadrul unui nivel, Alpha reprezintă cel mai bun scor pe care maximizatorul îl poate obține, iar Beta reprezintă cel mai bun scor pe care minimizatorul îl poate obține (nu neapărat doar pe acel nivel, ci pentru toate nodurile calculate precedent). Inițial, Alpha primește o valoare foarte mică și Beta primește o valoare foarte mare, valorile fiind actualizate pe măsură ce arborele este parcurs. Alpha și Beta sunt actualizate la niveluri de paritate diferită, în funcție de jucătorul la mutare.

Dacă la un moment Beta devine \leq decât Alpha, acest fapt ar însemna că ramura curentă va oferi întotdeauna o mutare mai dezavantajoasă pentru jucătorul de pe un nivel superior, caz în care nu mai are sens să fie explorată.

Pentru un exemplu vizual, a se vedea Figura 20.

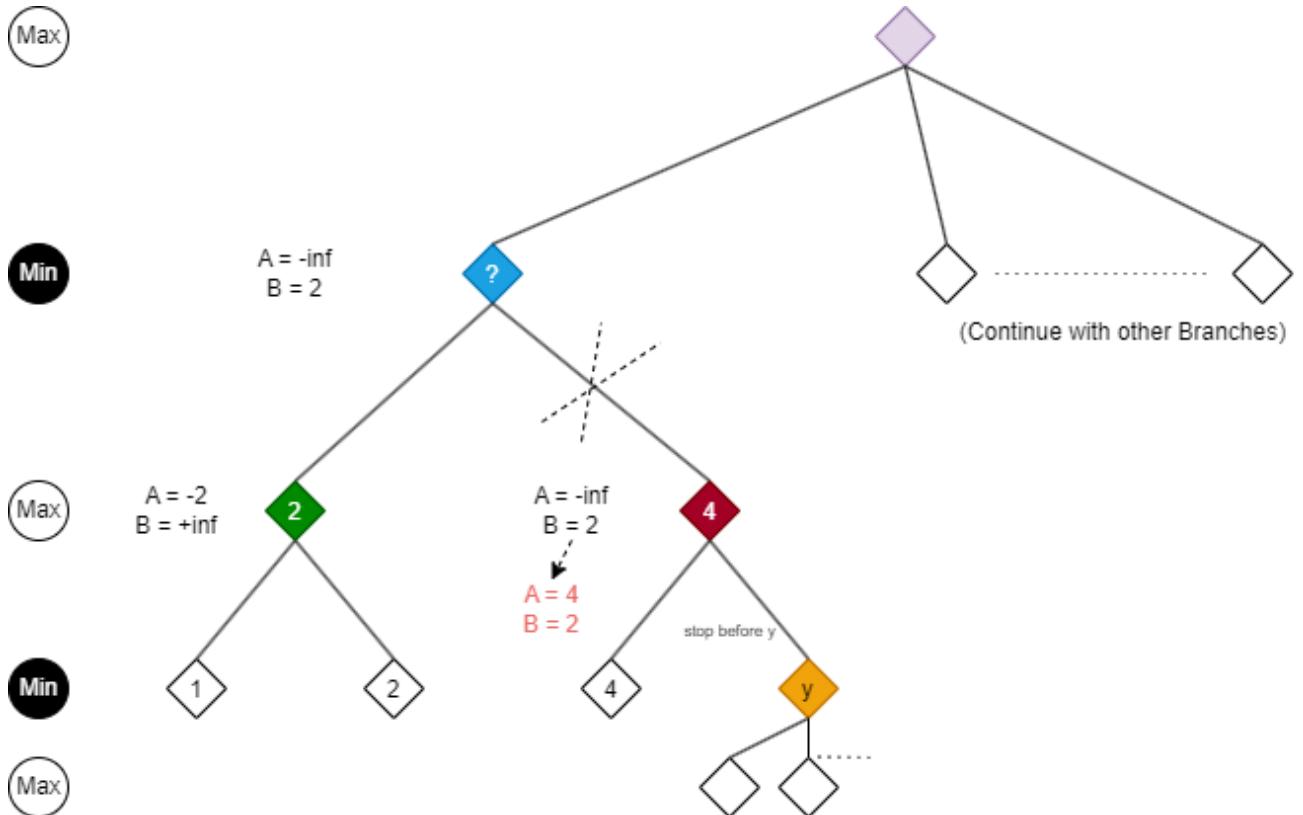


Figura 20: Pruning în arborele Minimax (sunt expuși doar 2 copii ai nodului albastru)

Atunci când se ajunge în nodul verde, suntem pe nivel de maximizator, deci se va actualiza Alpha pe măsură ce parcurgem copiii (în acest caz frunzele). Alpha devine 2, dar tot nu devine niciodată mai mare decât Beta = $+\infty$, aşa că deocamdată nu avem pruning.

Mai departe, se revine în nodul albastru și, fiind pe nivel de minimizator, se va actualiza Beta în 2, Alpha rămâne $-\infty$. La nodul roșu, primul copil găsit are valoarea 4 și, fiind pe nivel de maximizator, se actualizează Alpha în +4. Observăm că Alpha devine mai mare decât Beta, înseamnă că toată ramura nodului roșu devine irelevantă și se propagă în sus 4, nu mai explorăm nodurile copil ale acestuia.

Intuitiv, dacă am fi căutat mai departe, valoarea nodului roșu ar fi fost în final cel puțin 4 (deoarece ne aflăm pe nivel de maximizator), deci nu ar fi fost aleasă în defavoarea nodului verde, 2, pe nivelul de minimizator.

Observăm că toate potențialele noduri galbene 'y', inclusiv toate potențialele ramificații ce pornesc din acestea sunt ignorate, nu influențează decizia finală. Mai departe, s-ar fi putut continua cu alți copii ai nodului albastru pentru a-i afla valoarea și a începe procesarea pentru nodul mov, rădăcină.

Este important de reținut că Alpha și Beta nu se propagă în sus, ci doar în jos. Singurul lucru pe care îl comandă aceste valori este oprirea parcurgerii nodurilor copil. Ca și consecință, se propagă în sus o valoare "minim necesară", care nu influențează rezultatul final (în exemplul prezentat, această valoare este 4).

Asadar, algoritmul Minimax se modifică astfel:

Algorithm 13: Algoritmul MiniMax(gameState, depth, alpha, beta)

```
1 if depth == maximumDepth then
2   return StaticPositionEvaluator(gameState)
3 conclusion = GameStateManager.Instance.GetGameConclusion(gameState)
4 if conclusion == Draw then
5   return 0
6 else
7   if conclusion == Mate then
8     if gameState.whoMoves == 'w' then
9       return -1000 + depth
10    else
11      return 1000 - depth
12 float bestScore = worstValueForCurrentPlayer
13 moveList = GetLegalMoves(gameState)
14 foreach move in moveList do
15   gameState.MakeMove(move)
16   float score = MiniMax(gameState, depth + 1, alpha, beta)
17   gameState.UnmakeMove(move)
18   if score is better than bestScore then
19     bestScore = score
20   if gameState.whoMoves == 'w' and score > alpha then
21     alpha = score
22   if gameState.whoMoves == 'b' and score < beta then
23     beta = score
24   if beta ≤ alpha then
25     break
26 return bestScore
```

5.9 Iterative Deepening Search

Pentru a ne asigura că motoarele de săh au un timp consistent de gândire, este impusă o limită de 5 secunde pentru calcularea evaluării și a mutării celei mai bune. În condiții normale, apare o problemă, dacă arborele Minimax nu este generat complet și timpul expiră, nu putem asigura o mutare, deoarece nu toate posibilitățile au fost luate în calcul. Ne putem imagina că, prin ghinion, au fost evaluate doar 3 mutări din 24 de mutări legale posibile până ca timpul să

expire, iar acestea ar fi avut cele mai rele valori din tot arborele (dacă ar fi fost generat complet).

În plus, setarea unei adâncimi fixe nu rezolvă problema, diferite poziții de săh au factori diferenți de ramificare și aceleași adâncimi nu vor fi atinse în aceleași timpuri de rulare. Ori setăm o adâncime prea mică și obținem un rezultat neoptim, ori setăm una prea mare și nu obținem niciun rezultat.

Iterative Deepening Search (IDS) presupune căutarea incrementală a celei mai bune mutări, întâi la adâncime 1, apoi la adâncime 2, 3 și aşa mai departe, până când expiră timpul. Astfel, se poate anula căutarea pentru adâncimea x și folosi rezultatul de la adâncimea $x - 1$.

Astfel, revenim la funcția `SolvePosition()`, din componenta **Game**:

Algorithm 14: `SolvePosition()`

```
1 moveToMakeFound = new Move()
2 for searchDepth = 1, true, ++searchDepth do
3     moveToMake = GetBestMove(globalGameState, searchDepth)
4     if !timeExpired then
5         moveToMakeFound = moveToMake
6     else
7         break
8     ++searchDepth
9 return moveToMakeFound
```

GetBestMove() este doar un wrapper peste `MiniMax()`. Acesta initializează Alpha și Beta și trece prin toate mutările posibile de la adâncimea 0, cheamă `MiniMax()` pe ele și le află evaluarea. Se face acest lucru deoarece este nevoie și de mutarea propriu-zisă, nu doar de evaluare, iar adăugarea unui overhead pentru reținerea acestei informații nu trebuie făcută la toate adâncimile, nivelul 0 este suficient.

(9) : O altă optimizare ce poate fi făcută cu IDS este ca prima mutare explorată la iterată $x + 1$ să fie cea mai bună mutare găsită la iterată x . Astfel, dacă timpul expiră, dar am apucat să evaluăm complet cel puțin o mutare, atunci cea mai bună dintre acestea poate fi:

- Cea de la iterată precedentă, caz în care nu ne rănește cu nimic.
- Altă mutare, dar întrucât cea mai bună mutare de la iterată precedentă a fost evaluată prima, această mutare va avea o evaluare chiar mai bună, ceea ce ar fi ideal.

Astfel, ne folosim de rezultatele de la iterărilor precedente pentru a garanta că un rezultat parțial obținut poate fi folosit. În plus, acest mod de explorare poate duce și la mai mult pruning, având o valoare initială mai mare pentru Alpha sau mai mică pentru Beta.

Acest tip de optimizare este folosit pentru **Alv2**, **Alv3**, **Alv4**, **Alv5**, **Alv6** și **Alv7**.

5.10 Ordonarea Mutărilor

Alpha-Beta Pruning este o tehnică bună, dar nu este folositoare decât dacă reduce suficiente ramuri. În cel mai rău caz, cele mai slabe mutări sunt explorate și evaluate primele, caz în care pruning-ul este practic zero. În mod ideal, ne dorim să evaluăm cele mai bune mutări primele, pentru a maximiza pruning-ul. Dar cum putem să ne dăm seama ce mutări sunt bune dacă fix pentru acest lucru facem căutarea, să gasim mutările bune? Răspunsul este că trebuie să ghicim, cu alte cuvinte aplicăm euristici ce descriu mutări bune:

- Se preferă mutări ce capturează o piesă de valoare mai mare cu o piesă de valoare mai mică
- Se descurajează mutări ce mută o piesă de valoare mare pe un pătrat controlat de un pion inamic
- Se încurajează mutările care promovează pionii

Algorithm 15: Pseudocod pentru OrderMoves(moveList, gameState)

```
1 foreach move in moveList do
2     int moveScore = 0
3         // capturing a higher value piece
4         if move.capturedPiece.value > move.movingPiece.value then
5             moveScore += capturedPiece.value - move.movingPiece.value
6
7         if move.movingPiece.value > pawn.value then
8             // moving to a square attacked by a pawn
9             if move.destination isControlledBy enemyPawn then
10                movescore -= move.movingPiece.value
11
12            // if a pawn is promoting
13            if move.movingPiece is pawn and move.destination == lastRank then
14                movescore += queen.value
15
16        move.score = moveScore
17
18 Sort(moveList, compareByScore)
```

Acest tip de optimizare este folosit de către **Alv3**, **Alv4**, **Alv5** și **Alv7**.

Este de menționat că, pentru poziții foarte tactice, în care mutarea cea mai bună nu este neapărat cea mai evidentă, ordonarea mutărilor clar nu va ajuta pruning-ul. Chiar și în acest caz, frecvența acestui tip de poziții într-un meci real nu este suficient de mare încât să anuleze efectul pozitiv, în medie, al ordonării mutărilor.

5.11 Tabele de transpoziție

În arborele Minimax, este posibil ca poziția descrisă de două sau mai multe noduri să fie identică. Acest lucru este posibil prin diferite variații ale ordinii mutărilor sau a deplasării pieselor. De exemplu, o tură se poate deplasa 2 pătrate la o mutare și 3 pătrate la altă mutare, în aceeași direcție. Dacă tura s-ar fi deplasat altfel, un singur pătrat la o mutare și 4 pătrate la alta, practic s-ar fi ajuns la aceeași poziție (presupunând că și adversarul a făcut un lucru similar). A se vedea Figura 21.

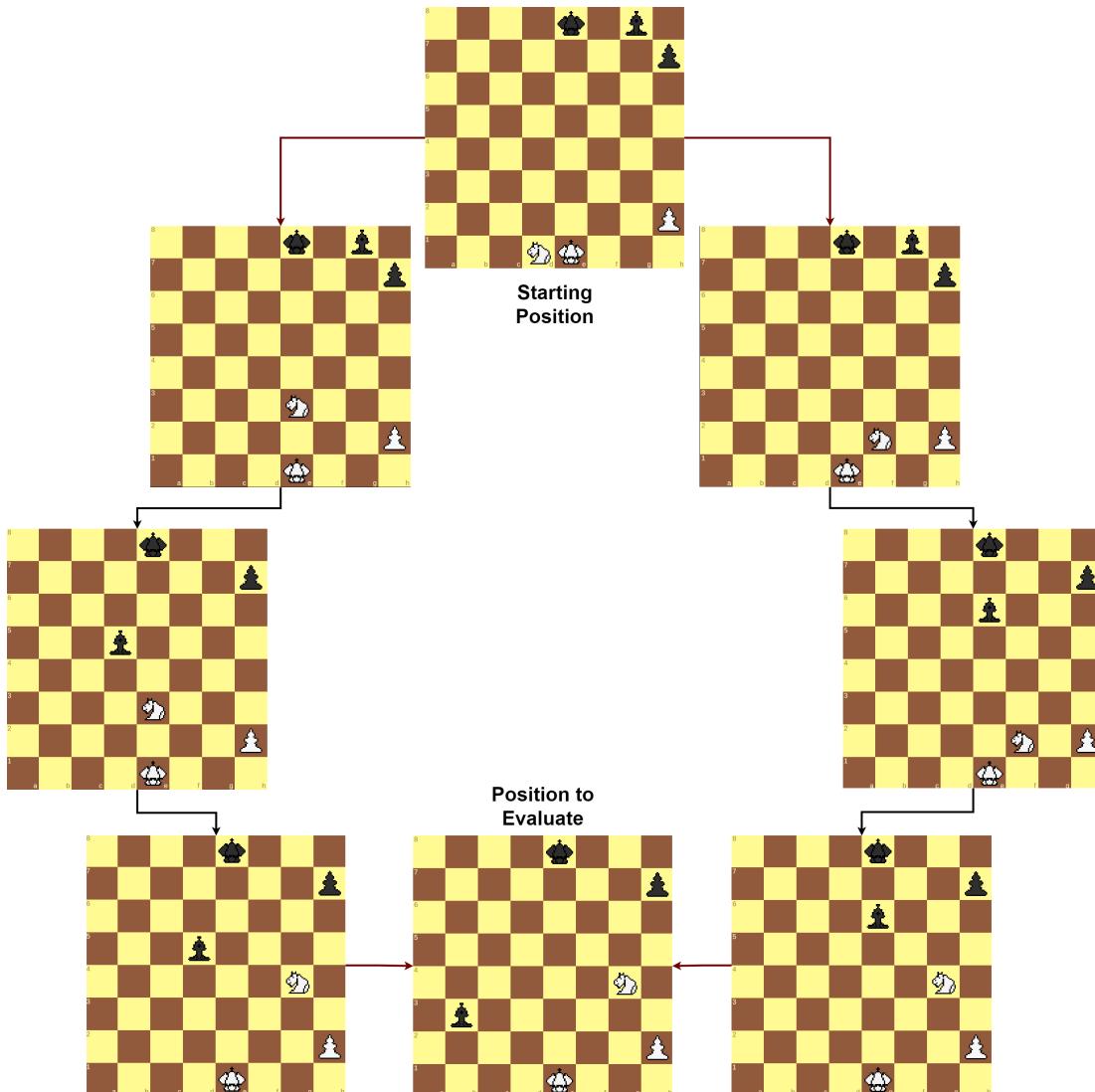


Figura 21: Se ajunge la aceeași poziție pe două căi diferite

Totuși, nu este nevoie să evaluăm din nou un astfel de nod, dacă a mai fost deja întâlnit. Așadar, în cadrul unei iterări de căutare, se folosește o tabelă de dispersie în care se adaugă treptat elemente, pe măsură ce arborele Minimax este parcurs. Atunci când se ajunge la un nod nou, prima dată se va verifica tabela de transpozitie și, dacă există o intrare, nu mai este necesară explorarea sa. Altfel, se face explorarea și se adaugă o nouă intrare în tabelă.

Algoritmul Minimax se modifică astfel:

Algorithm 16: Metoda MiniMax() ce folosește o tabelă de transpoziție

```
1 if transpositionTable.ContainsKey(gameState) then
2   return transpositionTable[gameState]
3 if depth == maximumDepth then
4   float score = StaticPositionEvaluator(gameState)
5   transpositionTable.Add(gameState, score)
6   return score
7 conclusion = GameStateManager.Instance.GetGameConclusion(gameState)
8 if conclusion == Draw then
9   transpositionTable.Add(gameState, 0)
10  return 0
11 else
12  if conclusion == Mate then
13    if gameState.whoMoves == 'w' then
14      transpositionTable.Add(gameState, -1000)
15      return -1000 + depth
16    else
17      transpositionTable.Add(gameState, 1000)
18      return 1000 - depth
19 float bestScore = worstValueForCurrentPlayer
20 moveList = GetLegalMoves(gameState)
21 foreach move in moveList do
22   gameState.MakeMove(move)
23   float score = MiniMax(gameState, depth + 1, alpha, beta)
24   gameState.UnmakeMove(move)
25   if score is better than bestScore then
26     bestScore = score
27   if gameState.whoMoves == 'w' and score > alpha then
28     alpha = score
29   if gameState.whoMoves == 'b' and score < beta then
30     beta = score
31   if beta ≤ alpha then
32     break
33 transpositionTable.Add(gameState, bestScore)
34 return bestScore
```

5.12 AI

În cadrul aplicației au fost dezvoltate 7 motoare principale pentru adversarii AI. Toate folosesc aceleasi valori absolute pentru piese, Minimax cu Alpha Beta Pruning, și Iterative Deepening. Se diferențiază în combinațiile dintre celelalte tehnici și euristică prezentate după cum urmează:

- **Alv1:**

- Funcție statică de evaluare bazată pe pătrate controlate (variantă minimală)
- Iterative deepening (fără modificări)

- **Alv2:**

- Funcție statică de evaluare bazată pe pătrate controlate (variantă minimală)
- Iterative deepening optimizat pentru rezultate parțiale

- **Alv3:**

- Funcție statică de evaluare bazată pe pătrate controlate (variantă minimală)
- Iterative deepening optimizat pentru rezultate parțiale
- Ordonarea mutărilor

- **Alv4:**

- Funcție statică de evaluare bazată pe PST
- Iterative deepening optimizat pentru rezultate parțiale
- Ordonarea mutărilor

- **Alv5:**

- Funcție statică de evaluare bazată pe PST
- Iterative deepening optimizat pentru rezultate parțiale
- Ordonarea mutărilor
- Tabelă de transpoziție

- **Alv6:**

- Funcție statică de evaluare bazată pe pătrate controlate (variantă mai complexă, se ia în calcul distanța la rege și la centru)
- Iterative deepening optimizat pentru rezultate parțiale

- **Alv7:**

- Funcție statică de evaluare bazată pe pătrate controlate (variantă mai complexă, se ia în calcul distanța la rege și la centru)
- Iterative deepening optimizat pentru rezultate parțiale
- Ordonarea Mutărilor

IDS-ul optimizat este explicitat în cadrul secțiunii 5.9, la (9).

6 EVALUARE

6.1 Contextul Evaluării

Pentru a pune în perspectivă capabilitatea motoarelor dezvoltate, vom face o comparație cu motorul de șah Stockfish 16.1. Pentru a face acest lucru ne vom folosi de un dataset¹ cu 575 de FEN-uri ce reprezintă poziții diversificate, create special să testeze capacitatea motoarelor de șah din diferite perspective: tactici și înțelegere pozitională. Pozițiile tactice beneficiază mai mult de o generare eficientă a mutărilor și de pruning, fiind necesară o căutare destul de adâncă pentru a găsi soluția optimă.

În faza inițială, se vor trece aceste poziții prin Stockfish 16.1. și se va afla cea mai bună mutare pentru fiecare dintre ele, împreună cu valoarea numerică a evaluării. Se va folosi un timp de gândire de 5 secunde. Scriptul ce face acest lucru se poate găsi la **Anexe**, Figura 44.

Știind aceste informații, pozițiile vor fi rulate din nou prin fiecare dintre motoarele dezvoltate și se va afla evaluarea pentru mutarea sugerată de Stockfish. Bineînțeles, se va folosi tot un timp de gândire de 5 secunde.

Ca și adăugare, vom avea grija ca nodul corespunzător mutării sugerate în arborele Minimax să fie parcurs primul, pentru a nu exista posibilitatea ca ramura sa să fie tăiată în urma alpha-beta pruning, fapt ce ar duce la pierderea evaluării. Această schimbare nu oferă niciun avantaj motorului, ba chiar potențial îl rănește, în caz că mutarea nu ar fi aceeași cu cea găsită dacă nu ar fi fost făcută modificarea (existând posibilitatea să se salveze timp de computare prin pruning). Dacă mutarea sugerată este evaluată prima și duce la pruning-ul altor ramuri, fie este mutarea cea mai bună, deci ar fi fost găsită oricum, fie nu este cea mai bună, caz în care mutarea cea mai bună ar fi dus la același pruning, dacă nu chiar mai mult.

După acest proces, pentru fiecare motor, avem la dispoziție 575 de perechi de valori de care trebuie să ne folosim pentru a calcula o diferență între acesta și Stockfish. Se propune următoarea formulă pentru calcularea diferenței din cadrul unei perechi de valori (x,y):

$$f(x, y) = \begin{cases} |x - y| & , \text{ if } |x| \leq 1 \text{ or } |y| \leq 1 \text{ or } x * y < 0 \\ \frac{|x-y|}{\min(|x|, |y|)} & , \text{ otherwise} \end{cases}$$

Se poate observa că formula este simetrică, practic putem spune că, pe de o parte, calculăm diferența de la motorul nostru la Stockfish și că, pe de altă parte, calculăm diferența de la Stockfish la motorul nostru, pentru o anumită mutare.

¹<https://www.chessprogramming.org/Test-Positions>

Legat de condițiile funcției:

- dacă fie x , fie y se află în intervalul $[-1, 1]$, atunci diferența este pur și simplu distanța între valori
- dacă valorile se află de părți opuse ale lui 0, adică au semne diferite, facem același lucru, doar distanța între valori
- ultimul caz, și anume dacă x și y se află de aceeași parte a lui 0, deci au același semn, dar și sunt ambele mai mari decât 1 în valoare absolută, se va face o diferență relativă la magnitudinea celei mai mici valori în valoare absolută.

Să luăm câteva exemple:

- $f(-3, 3) = |3 - (-3)| = 6$
- $f(5, -1) = |-1 - 5| = 6$
- $f(1, 7) = |1 - 7| = 6$

Până acum, totul arată bine, putem spune că diferența de la o evaluare de -3 (ce indică un avantaj considerabil pentru negru) la una de $+3$ (ce indică un avantaj considerabil pentru alb) ar fi similară cu diferența de la o evaluare de $+5$ la una de -1 , sau de la una de $+1$ (un mic avantaj pentru alb) la una de $+7$ (un avantaj incontestabil pentru alb).

Dar dacă luăm niște exemple din ultimul caz, fără să facem diferență relativă:

- $f'(8, 14) = |8 - 14| = 6$
- $f'(-4, -6) = |-4 - (-6)| = 2$
- $f'(4, 7) = |4 - 7| = 3$

Ne putem pune întrebarea, "oare diferența de la o evaluare de $+8$ la una de $+14$ este la fel ca de la -3 la $+3$?" Ambele evaluări denotă un avantaj incontestabil. Dar o diferență de la -4 la -6 (ambele denotă câstig decent pentru negru) este la fel ca una de la 0 (egalitate totală) la 2 (avantaj de 2 pioni)? Dar pentru ultimul caz, este diferența de la $+4$ la $+7$ mai mare decât cea de la -1 la $+1.5$?

În mod evident, trebuie să ținem cont de magnitudinea valorilor, cu cât sunt mai îndepărtate de 0, cu atât diferența relativă scade. Folosind funcția propusă, vechile valori devin:

- $f(8, 14) = \frac{|8-14|}{\min(8,14)} = 0.75$
- $f(-4, -6) = \frac{|-4-(-6)|}{\min(-4,-6)} = 0.5$
- $f(4, 7) = \frac{|4-7|}{\min(4,7)} = 0.75$

Această "euristică" este aplicabilă și intuitivă deoarece zona de evaluare $[-1, 1]$ este o zonă "gri" în săh, în care nu se poate spune cu certitudine dacă jucătorul ce deține infimul avantaj poate converti jocul într-o victorie (presupunând un joc perfect, fără greșeli).

De asemenea, pentru o reprezentare corectă a schimbării, funcția propusă este continuă. Dacă presupunem că x este valoarea mai mică în modul, atunci când $|x|$ se apropi de 1, numitorul pentru condiția a doua devine 1, aşadar funcția este continuă în 1 și -1 . Acest lucru se poate observa și în Figura 22, realizată în GeoGebra².

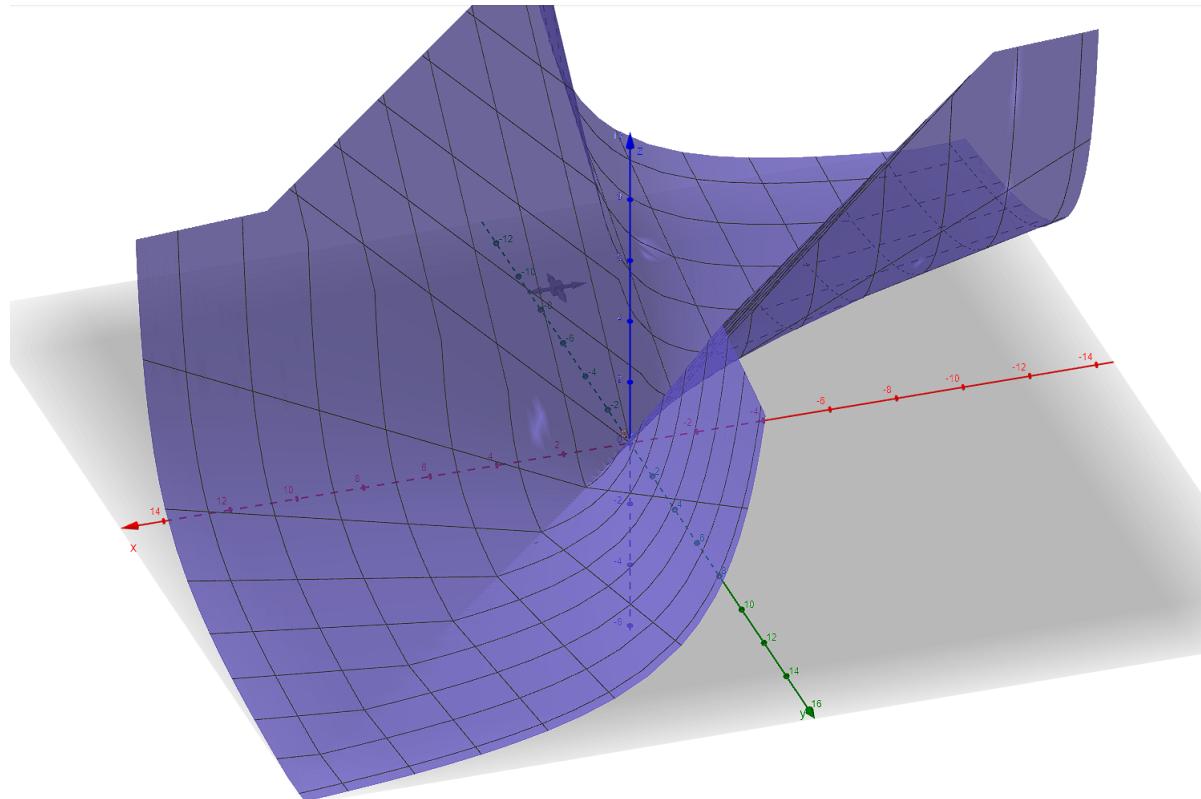


Figura 22: Funcția Propusă

Se observă că valoarea funcției este 0 pentru valori egale și apare o creștere liniară pentru perechi în care cel puțin una dintre valori este în $[-1, 1]$ sau valorile au semne opuse. Funcția se "aplatizează" (are valori mici) pe măsură ce magnitudinea valorilor din celelalte tipuri de perechi crește.

Pentru a cuantifica abilitatea motoarelor într-un singur număr, se va face media pătratică a tuturor diferențelor relative, pentru a scoate în evidență diferențele mari. Este important de menționat că acest număr nu poate descrie în întregime capacitatea actuală de joc a motoarelor. Foarte multe subtilități sunt ascunse, puterea de joc putând fi estimată mai bine în jocuri propriu-zise, unde există doar pierdere sau victorie (nu contează dacă motorul a jucat perfect 29 de mutări din 30, dacă acea mutare imprecisă a fost o eroare crucială).

²<https://www.geogebra.org/3d/ru64txh>

6.2 Rezultatele evaluării

Mediile diferențelor relative pentru fiecare motor se pot regăsi în Tabela 2:

Tabela 2: Rezultatele Evaluării motoarelor

Alv1	Alv2	Alv3	Alv4	Alv5	Alv6	Alv7
3.70	3.64	3.56	3.56	3.27	3.94	4.08

Putem trage următoarele concluzii:

- Se observă efectul pozitiv al optimizării IDS de la **Alv1** la **Alv2**
- Se observă efectul pozitiv al optimizării de ordonare a mutărilor de la **Alv2** la **Alv3**
- Se observă efectul pozitiv al optimizării folosind tabela de transpoziție de la **Alv4** la **Alv5**, care a obținut chiar cele mai bune rezultate
- Folosirea PST duce la rezultate mai bune decât a funcțiilor bazate pe pătrate controlate, fapt ce indică preferarea căutării mai adânci peste o funcție mai complexă de evaluare statică
- Introducerea unei funcții statice mai complexe bazată pe pătrate controlate a avut efecte negative de la **Alv2** la **Alv6**. În plus, ordonarea mutărilor a avut și aceasta efecte negative pentru astfel de motoare de la **Alv6** la **Alv7**.

Dacă analizăm chiar și rezultatele obținute în urma aplicării mediei aritmetice (deci valorile mari au o influență mai mică asupra rezultatului) a diferențelor relative, conform Tabelei 3, ajungem la aceeași constatări:

Tabela 3: Rezultatele Evaluării motoarelor folosind medie aritmetică

Alv1	Alv2	Alv3	Alv4	Alv5	Alv6	Alv7
2.47	2.42	2.36	2.35	2.18	2.71	2.87

Putem trage niște concluzii și din punct de vedere al distribuției valorilor în cadrul fiecărui motor referitor la stilul de joc sau la ce fel de greșeli este susceptibil, în funcție de euristicile sau algoritmii folosiți.

Pentru început, vom compara **Alv3**, cel mai bun motor ce folosește o funcție simplă bazată pe controlul pătratelor, cu **Alv4**, ce folosește PST. Histogramele se pot vedea la Figurile 23 și 24.

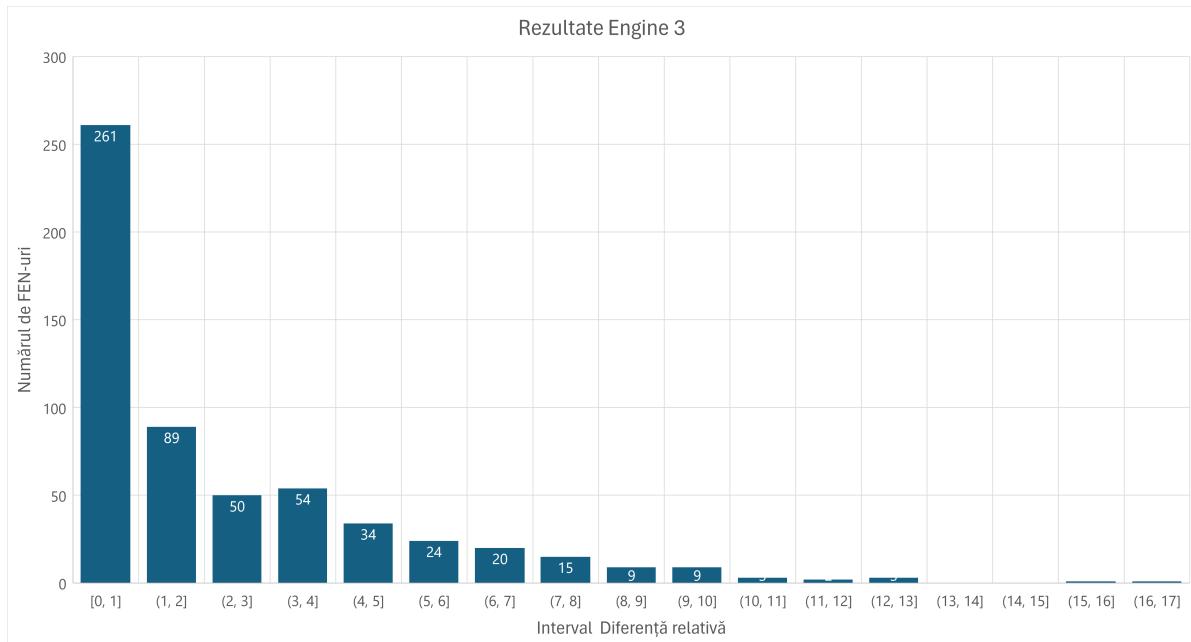


Figura 23: Histogramă **Alv3**

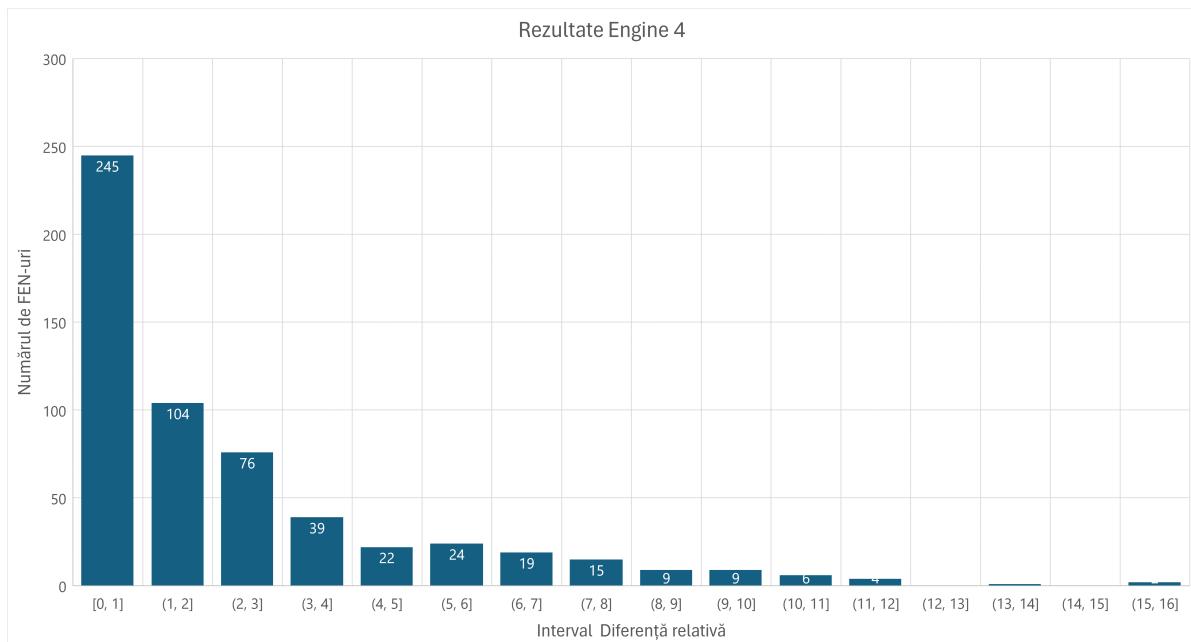


Figura 24: Histogramă **Alv4**

Deși motoarele au aproximativ aceeași evaluare (3.56), distribuția valorilor este puțin diferită. **Alv3** tinde să acumuleze mai multe valori în intervalul [0, 1] și are o distribuție mai uniformă în intervalul (1,5], comparativ cu **Alv4**. Se pare că motorul bazat pe PST tinde să facă mai multe greșeli, dar acestea sunt mai puțin grave (diferență < 3) decât cele făcute de **Alv3** (mai multe valori în (4,6]). Acest lucru indică un stil de joc mai consistent.

Mai departe, vom investiga de ce introducerea unei funcții de evaluare mai complexe la **Alv6** și **Alv7** are efecte negative prin compararea **Alv6** cu **Alv2**, conform Figurilor 25 și 26.

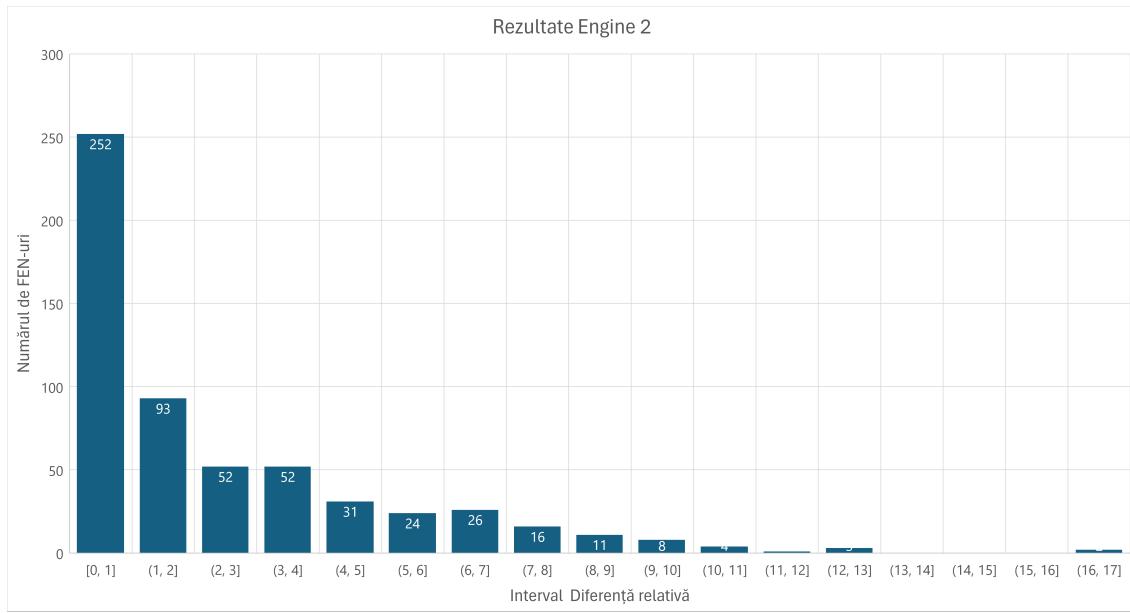


Figura 25: Histogramă **Alv2**

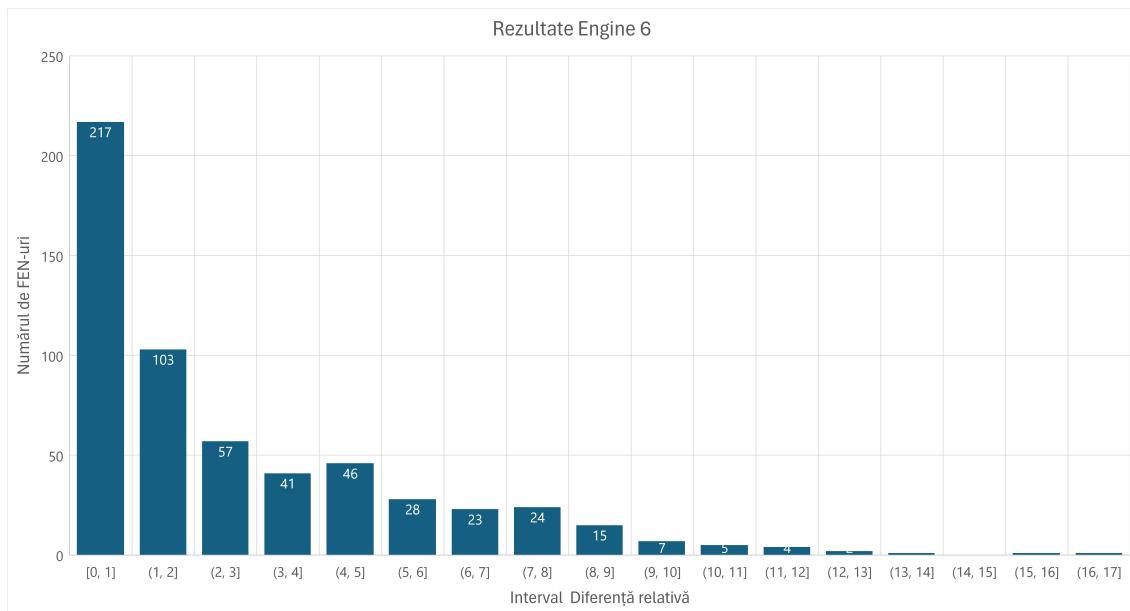


Figura 26: Histogramă **Alv6**

Un lucru interesant este că ambele motoare acumulează aproximativ același număr de valori în intervalul $(1, 4]$, dar **Alv6** obține rezultate chiar mai bune, cu multe dintre valori în $(1, 2]$. Slăbiciunea vine de la numărul de valori subunitare, diferențele mici, unde **Alv2** performează mai bine. Relativ vorbind, **Alv2** face mai multe mutări aproape perfecte decât **Alv6**, fapt ce poate fi cauzat de intensitatea computațională a funcției de evaluare. Acest lucru indică o slăbiciune pentru pozițiile tactice.

Toate histogramele (împreună cu scatter plot-urile lor corespunzătoare) pot fi găsite la capitolul **Anexe**, Figurile 27 - 40.

Din punctul de vedere al stilurilor de joc ale motoarelor, s-au obținut următoarele constatări de la jucători de șah începători și intermediari:

- Alv1, Alv2 și Alv3 preferă să dezvolte regina în deschidere și să joace pasiv, ținându-si majoritatea pieselor apărate, menținând o poziție solidă.
- Alv1 mută regele destul de des când este atacat și nu se folosește de piese.
- Alv5 și Alv3 calculează cel mai bine tactici.
- Alv4 și Alv5 își dezvoltă piesele natural, dar într-un mod mai agresiv în deschidere, ulterior începând să joace pasiv.
- Alv7 își dezvoltă piesele cel mai natural, dar tinde să nu vadă tactici de 6-7 mutări în față.

Din punctul de vedere al funcționalităților, interfața a fost descrisă ca fiind simplă și intuitivă, iar tutorialele interesante și educative.

7 CONCLUZII

Proiectul a condus la obținerea unei aplicații funcționale și complete, ce este o unealtă folosită în scopul învățării șahului de către începători. Aceasta prezintă caracteristici relevante din perspectiva utilizatorilor, adaptându-se bine la situația curentă de pe piață. S-a pus în evidență complexitatea tehnicii folosite pentru dezvoltarea motoarelor de șah din cadrul jocului și s-au evaluat rezultatele obținute printr-o analiză detaliată. Informațiile obținute în urma evaluării sunt utile pentru o înțelegere mai în profunzime a interacțiunilor dintr-un astfel de sistem complex.

7.1 Dezvoltări ulterioare

Pentru îmbunătățirea viitoare a proiectului îmi propun să implementez alte euristici ce nu au fost abordate. Printre acestea se numără "killer heuristic", "history heuristic" și PST-uri reglate mai fin. Din punct de vedere al eficienței de bază a programului (pentru generarea mutărilor), îmi propun să tranziționez către niște structuri de date mai compacte și să fac comunicarea între componente cât mai accesibilă.

Din punct de vedere al evaluării, îmi propun un model și mai riguros, ce nu ia în considerare doar evaluarea celei mai bune mutări a sistemului 2 în sistemul 1, dar și invers. Această abordare ne-ar duce cu un pas mai aproape de estimarea puterii practice de joc: nu se va considera doar cum este evaluată mutarea corectă, ci și cum este evaluată (de către un sistem standard, de mare performanță) mutarea aleasă. Si mai mult, îmi propun evaluarea motoarelor prin organizarea de meciuri propriu-zise de șah, nu doar prin compararea cu un motor de referință.

În final, îmi doresc și să extind numărul de tutoriale, împreună cu dezvoltarea unui sistem de învățare dinamică, ce își adaptează comportamentul în funcție de acțiunile utilizatorului.

BIBLIOGRAFIE

- [1] I. V. Mikhaylova, A. S. Makhov, and A. I. Alifirov, "Chess as multicomponent type of adaptive physical culture," *Theory and practice of physical culture*, vol. (12), p. 56, 2015.
- [2] A. Jankovic and I. Novak, "Chess as a powerful educational tool for successful people," *7th International OFEL Conference on Governance, Management and Entrepreneurship: Embracing Diversity in Organisations. April 5th-6th, 2019, Dubrovnik, Croatia. Zagreb: Governance Research and Development Centre (CIRU)*, pp. 425–441, 2019.
- [3] I. V. Mikhaylova, I. N. Medvedev, O. N. Makurina, E. D. Bakulina, N. Y. Ereshko, and M. V. Eremin, "The effect of playing chess on an aging or pathological organism," *Journal of Biochemical Technology*, vol. 12(3-2021), pp. 47–52, 2021.
- [4] S. G. Majhi, "'a brave new world': Exploring the implications of online chess for the sport post the pandemic. singapore: Springer nature singapore," *In Sports Management in an Uncertain Environment*, pp. 255–270, 2023.
- [5] F. Stöckel, "Beating humans at their own game." 2022.
- [6] A. Siddharth, S. and Ali, E.-S. S., G. N., C. L., and Z. F. Syed, "A game-theoretic approach for calibration of low-cost magnetometers under noise uncertainty." *Measurement Science and Technology*, vol. 23, no. 2, 2012.
- [7] B. A. Kumar and M. S. Goundar, "Usability heuristics for mobile learning applications," *Education and Information Technologies*, vol. 24, pp. 1819–1833, 2019.
- [8] lichess.org, accesat ultima dată: 24 iunie 2024.
- [9] chess.com, accesat ultima dată: 24 iunie 2024.
- [10] <https://lucaschess.pythonanywhere.com/downloads>, accesat ultima dată: 24 iunie 2024.
- [11] <https://www.shredderchess.com/linux/deep-shredder-13.html>, accesat ultima dată: 24 iunie 2024.
- [12] H. Hussain, A. amd Shakeel, F. Hussain, N. Uddin, and T. L. Ghouri, "Unity game development engine: A technical survey." *Univ. Sindh J. Inf. Commun. Technol*, vol. 4, no. 2, pp. 73–81, 2020.

ANEXE

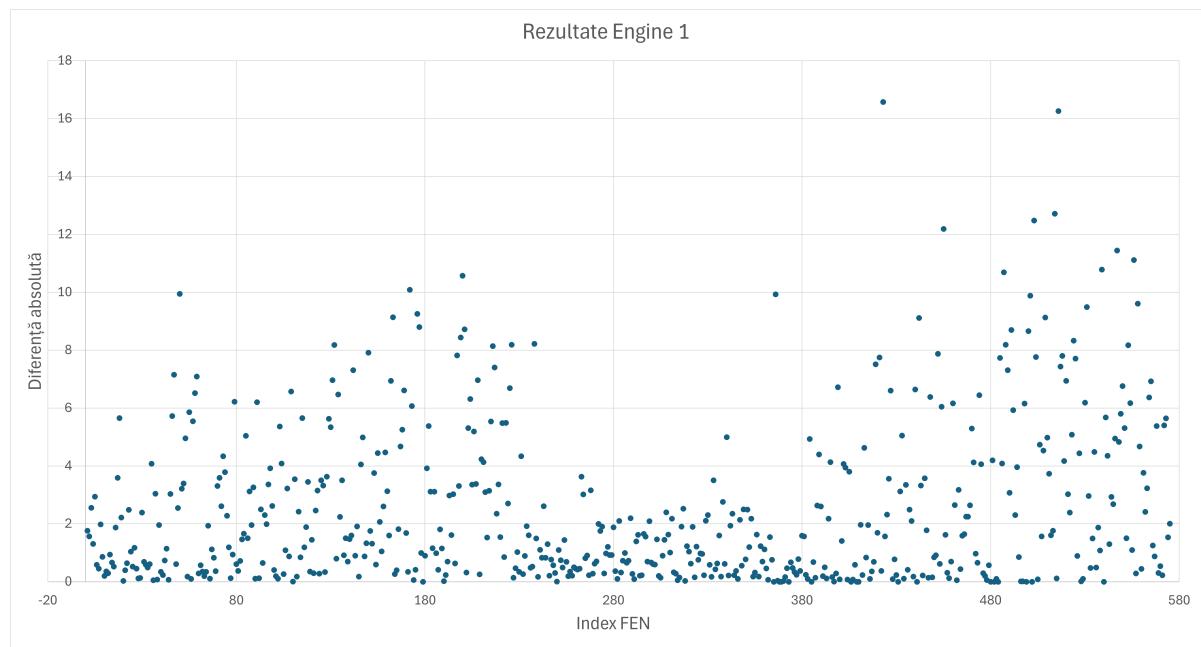


Figura 27: Scatter Plot **Alv1**

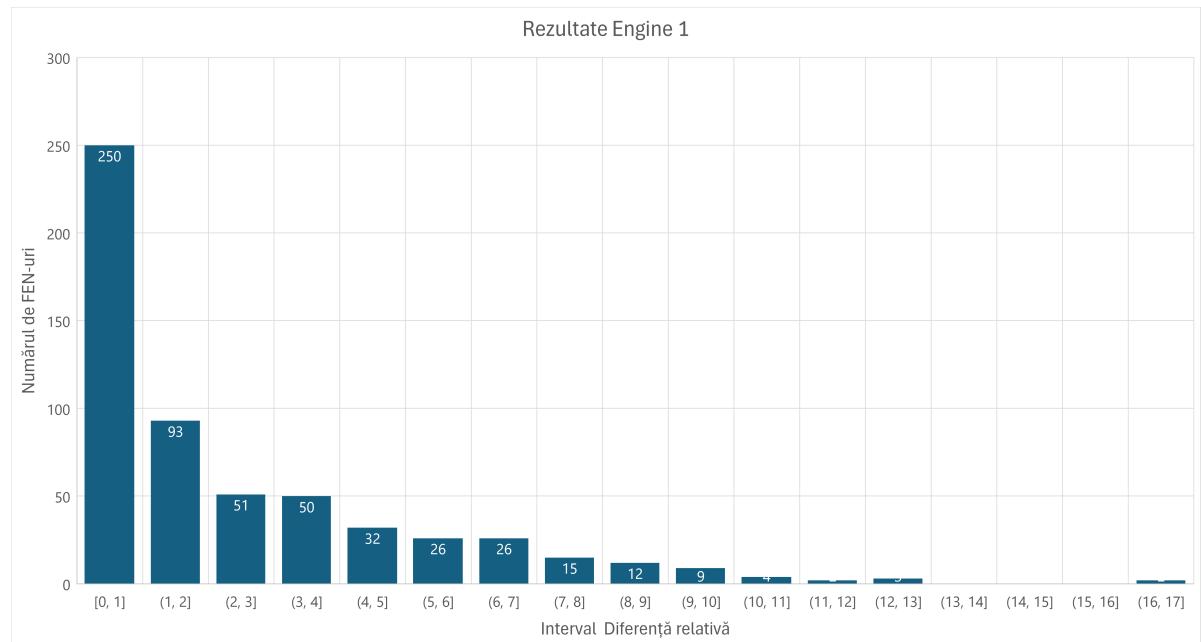


Figura 28: Histogramă **Alv1**

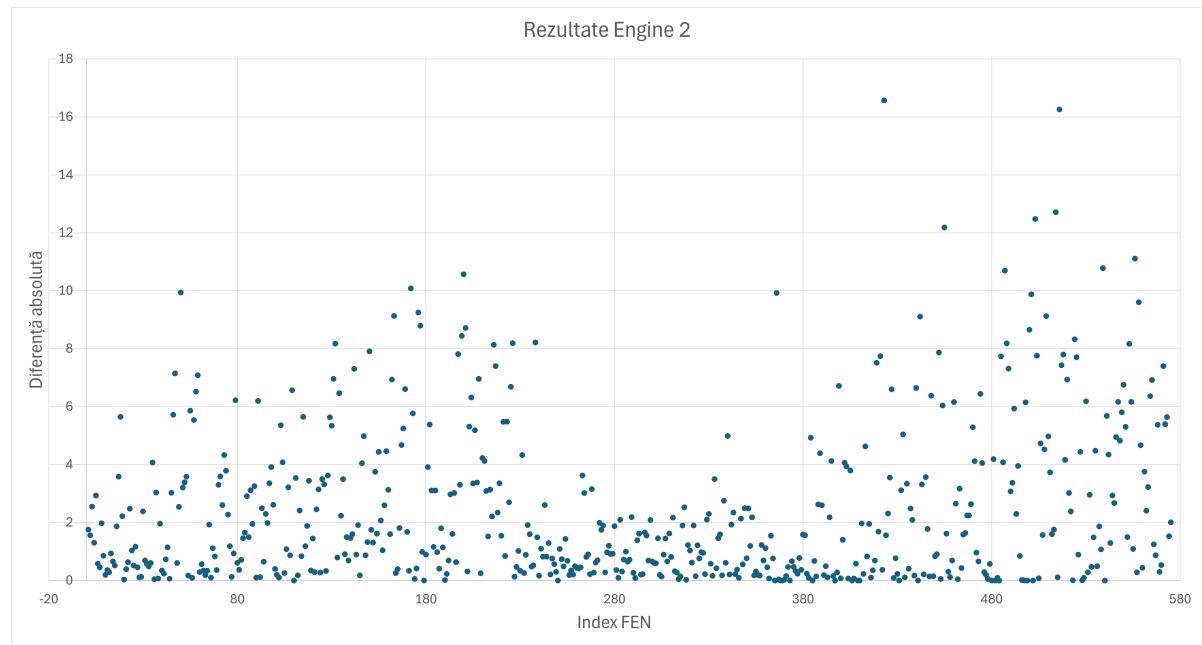


Figura 29: Scatter Plot **Alv2**

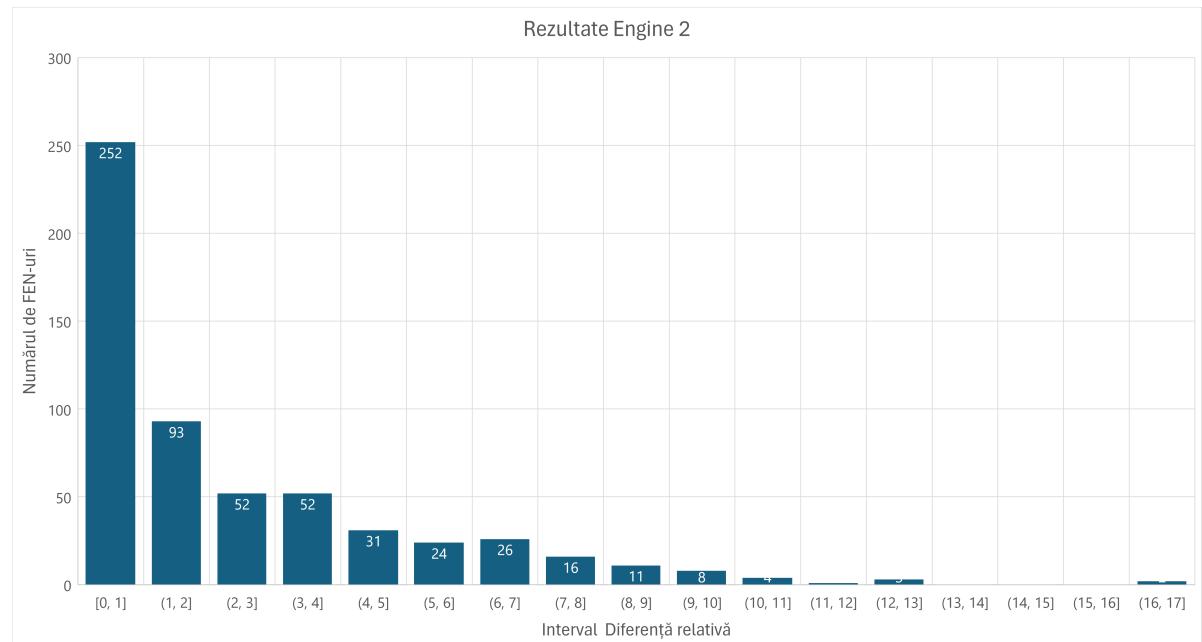


Figura 30: Histogramă **Alv2**

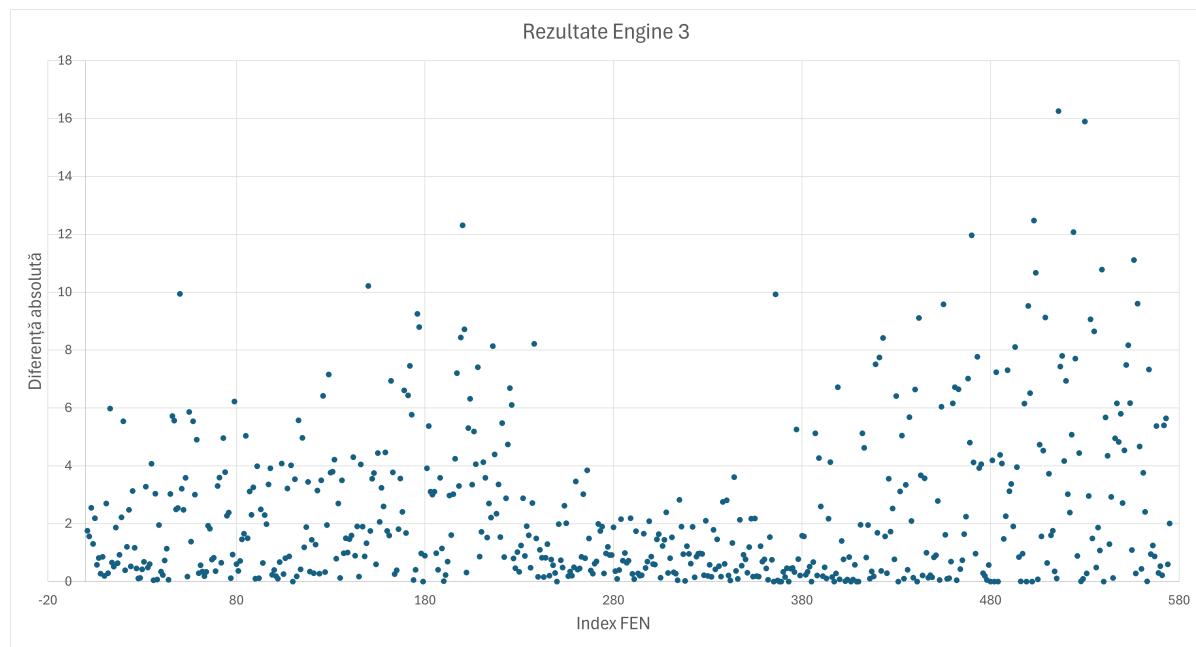


Figura 31: Scatter Plot **Alv3**

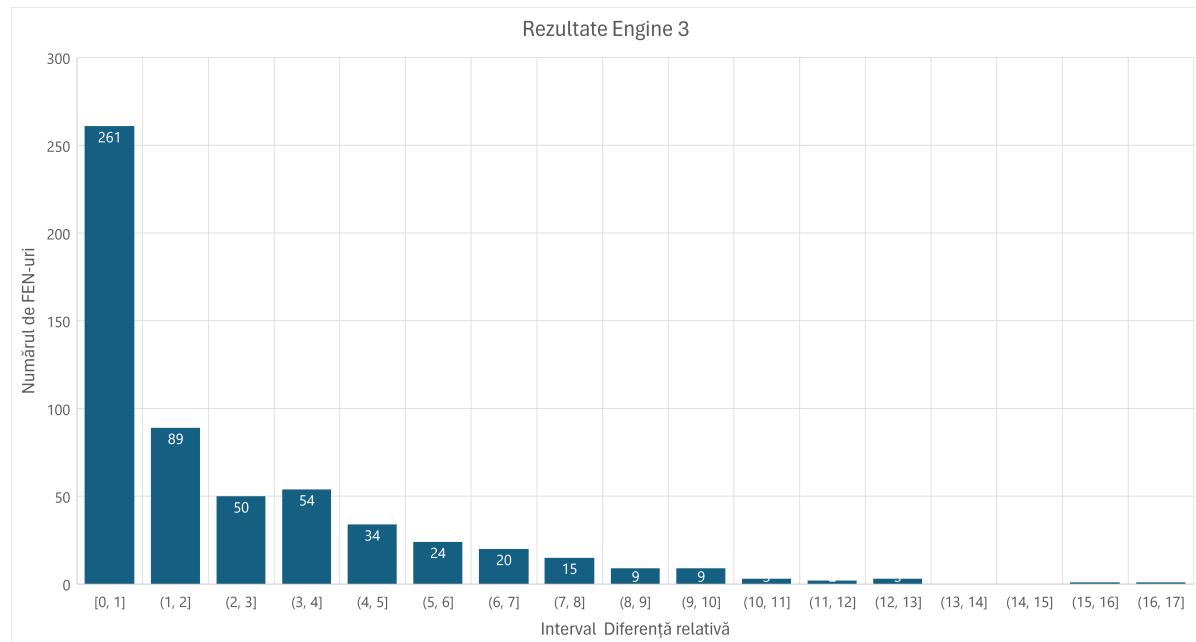


Figura 32: Histogramă **Alv3**

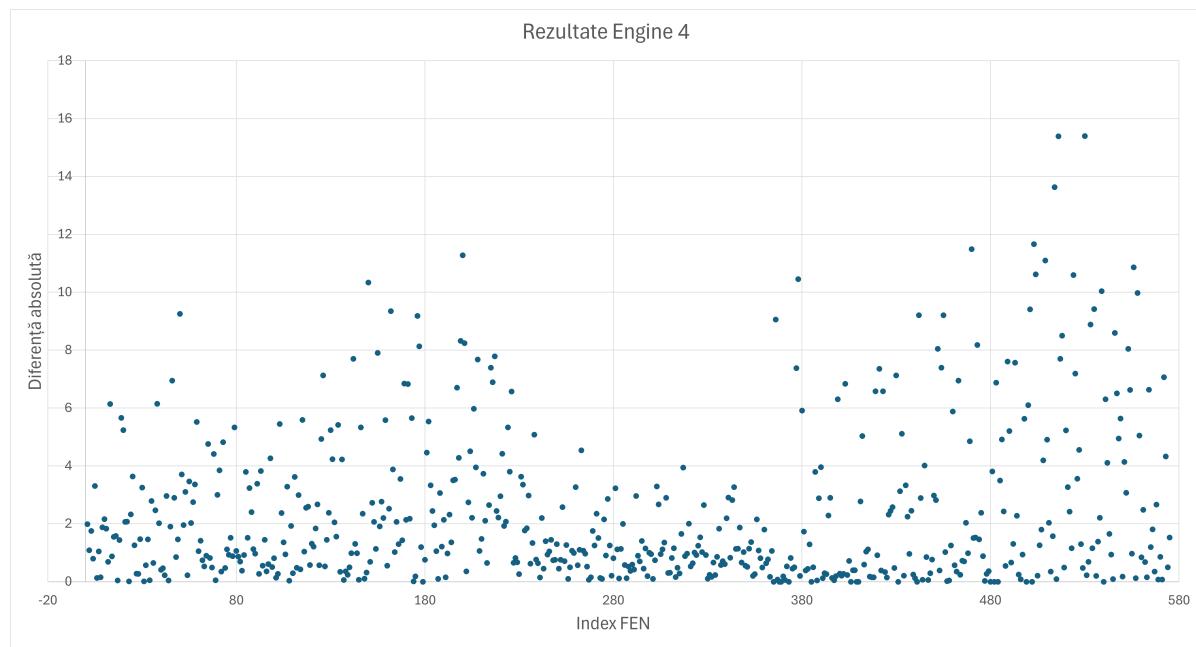


Figura 33: Scatter Plot **Alv4**

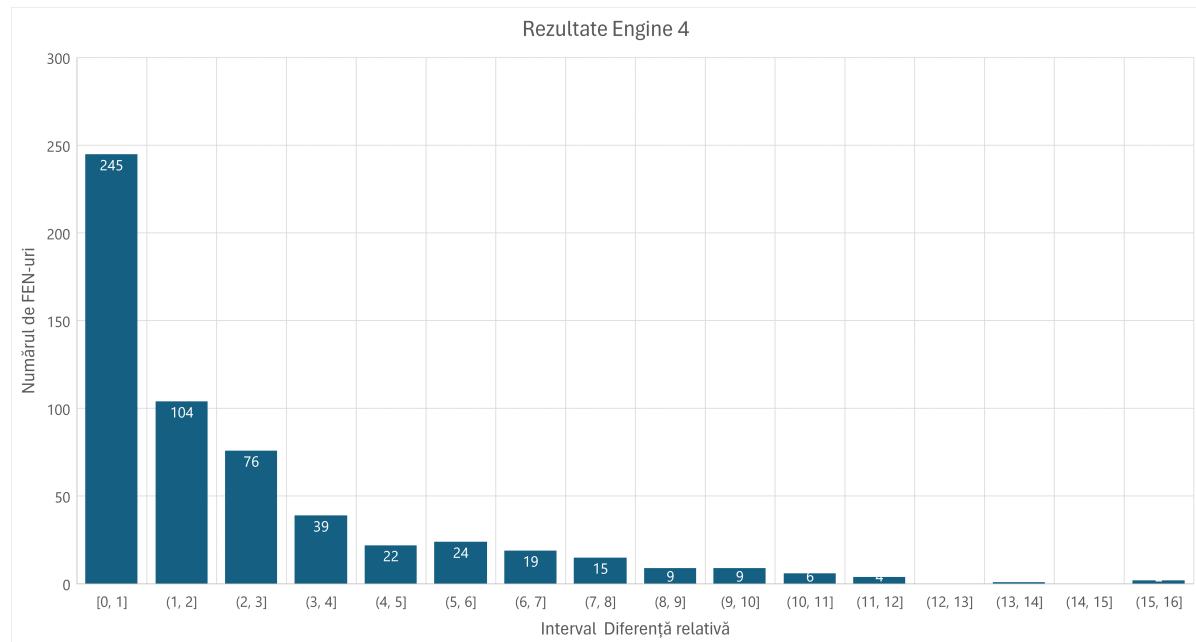


Figura 34: Histogramă **Alv4**

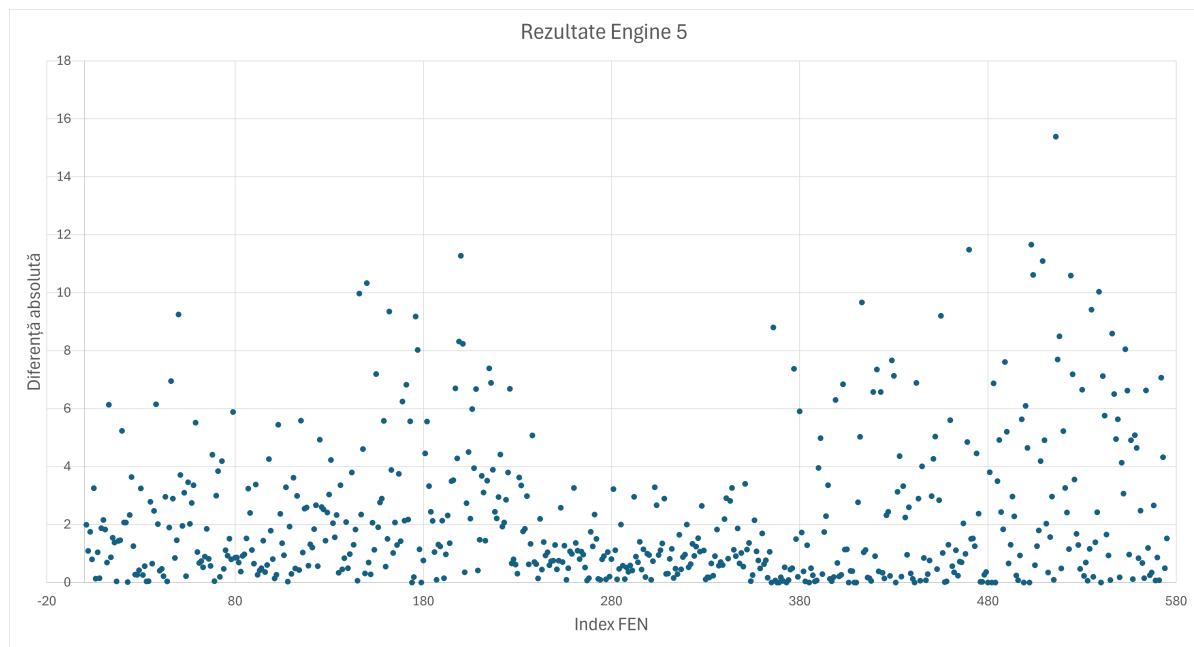


Figura 35: Scatter Plot **Alv5**

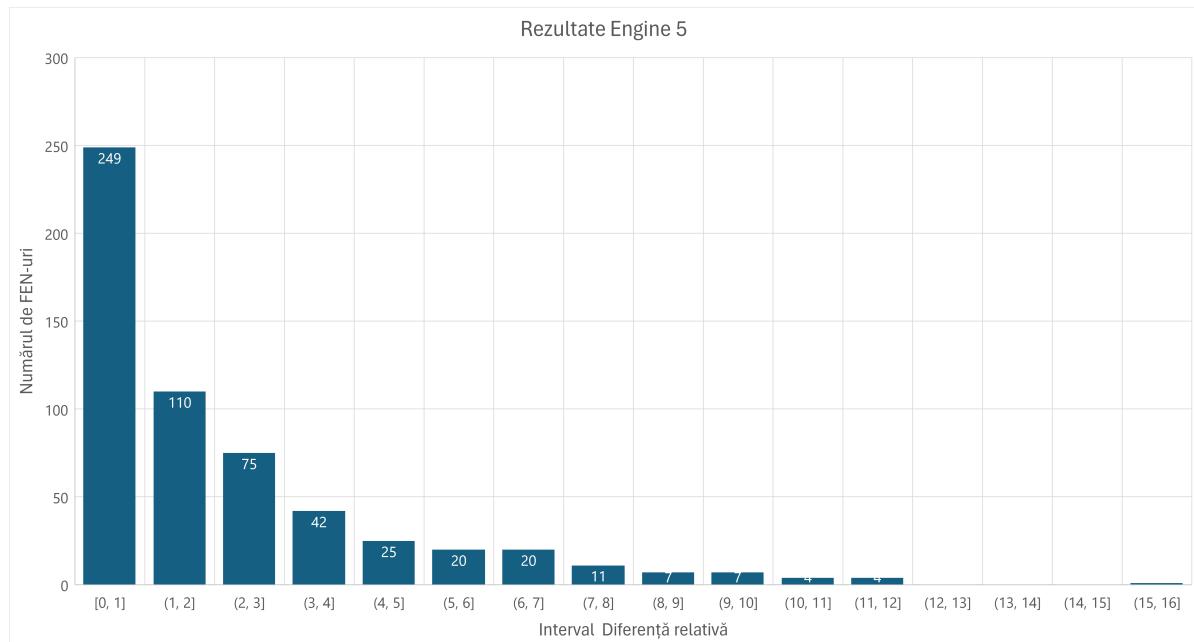


Figura 36: Histogramă **Alv5**

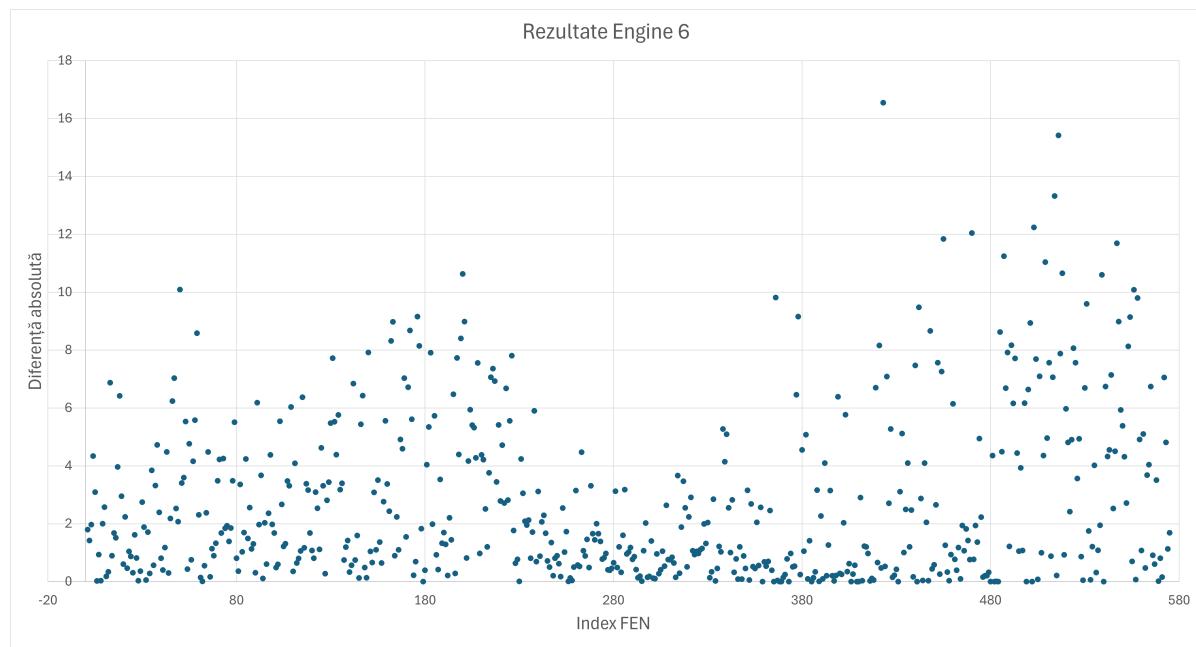


Figura 37: Scatter Plot **Alv6**

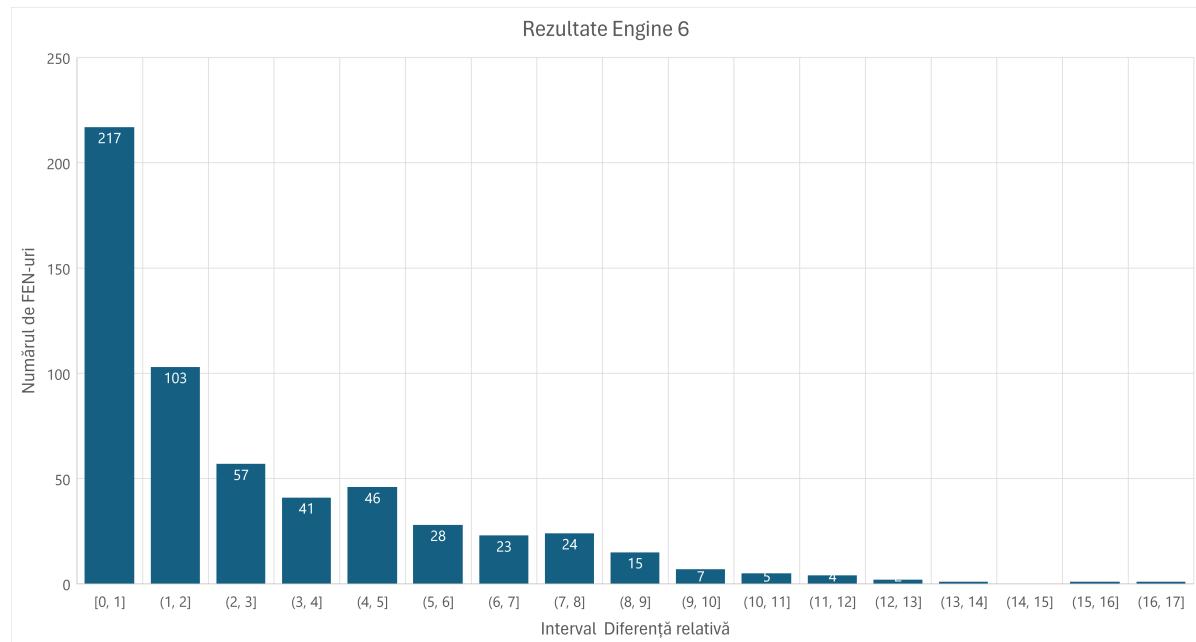


Figura 38: Histogramă **Alv6**

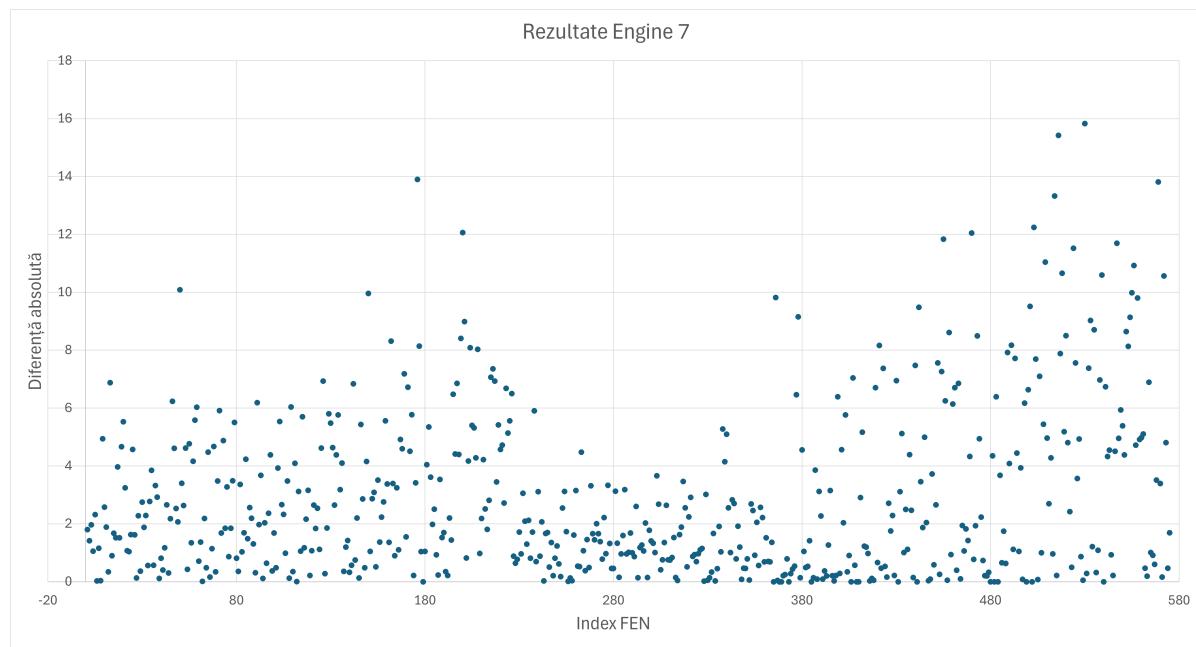


Figura 39: Scatter Plot **Alv7**

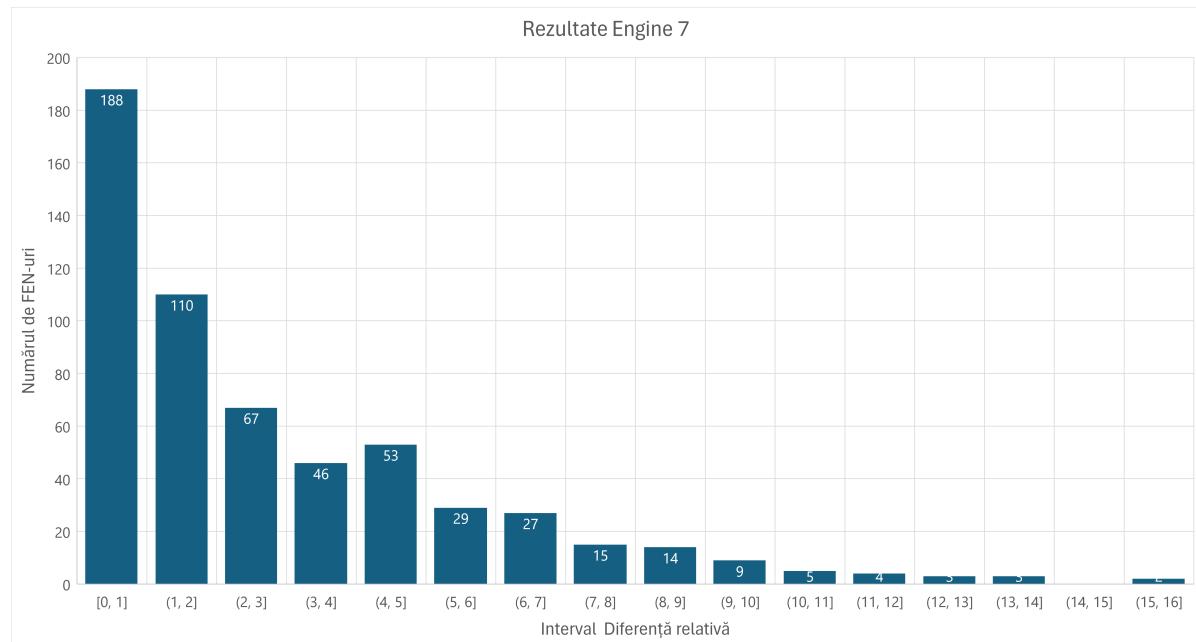


Figura 40: Histogramă **Alv7**

A EXTRASE DE COD

```
1 public static bool IsKingSafeFromRook(int king_i, int king_j, int rook_i, int rook_j,
2     IndexMove indexMove = null) {
3     if (indexMove == null) { // a move that would resolve the check was not provided
4         // king and rook are not even on the same file
5         if (king_i != rook_i && king_j != rook_j) {
6             return true;
7         }
8         // king captured the rook
9         if (king_i == rook_i && king_j == rook_j) {
10            return true;
11        }
12        return false;
13    }
14    int piece_i = indexMove.newRow, piece_j = indexMove.newColumn;
15    // moved piece is not on the same file or rank as the rook and king, therefore it can't resolve the check
16    if ((king_i == rook_i && piece_i != king_i) ||(king_j == rook_j && piece_j != king_j)) {
17        return false;
18    }
19    // if king and rook are on the same rank
20    if (king_i == rook_i) {
21        // if rook is blocked or captured
22        if ((king_j < rook_j && piece_j > king_j && piece_j <= rook_j)
23            || (king_j > rook_j && piece_j < king_j && piece_j >= rook_j)) {
24            return true;
25        }
26        return false;
27    }
28    // here king and rook are on the same file, we check if rook is blocked or captured again
29    if ((king_i < rook_i && piece_i > king_i && piece_i <= rook_i)
30        || (king_i > rook_i && piece_i < king_i && piece_i >= rook_i)) {
31        return true;
32    }
33    return false;
34 }
```

Figura 41: Metoda IsKingSafeFromRook()

...

```

1  public static void AddLegalKingMoves(GameState gameState, int old_i, int old_j,
2      Attacker attacker1, Attacker attacker2, List<IndexMove> legalMoves) {
3
4      char[,] boardConfiguration = gameState.boardConfiguration;
5
6      // The following arrays contain row and column increments for one-square king-movement
7      int[] smallSquareXIncrements = new int[8] { -1, 0, 1, -1, 1, -1, 0, 1 };
8      int[] smallSquareYIncrements = new int[8] { -1, -1, -1, 0, 0, 1, 1, 1 };
9
10     for (int idx = 0; idx < 8; ++idx) {
11         int new_i = old_i + smallSquareYIncrements[idx];
12         int new_j = old_j + smallSquareXIncrements[idx];
13         // in bounds
14         if (new_i >= 0 && new_i <= 7 && new_j >= 0 && new_j <= 7) {
15             char pieceOwner = GetPieceOwner(boardConfiguration[new_i, new_j]);
16             // king is blocked by own piece
17             if (pieceOwner == gameState.whoMoves) {
18                 continue;
19             }
20             IndexMove kingMove = new(old_i, old_j, new_i, new_j);
21             bool resolvesFirstAttacker = false, resolvesSecondAttacker = false;
22             switch (attacker1.type) {
23                 case -1: resolvesFirstAttacker = true; break;
24                 case 0: resolvesFirstAttacker = true; break; // king can't move into the same pawn's check
25                 case 1: resolvesFirstAttacker = IsKingSafeFromBishop(new_i, new_j, attacker1.row, attacker1.column); break;
26                 case 2: resolvesFirstAttacker = IsKingSafeFromRook(new_i, new_j, attacker1.row, attacker1.column); break;
27                 case 3: resolvesFirstAttacker = IsKingSafeFromKnight(new_i, new_j, attacker1.row, attacker1.column); break;
28             }
29             if (resolvesFirstAttacker) {
30                 switch (attacker2.type) {
31                     case -1: resolvesSecondAttacker = true; break;
32                     case 0: resolvesSecondAttacker = true; break; // king can't move into the same pawn's check
33                     case 1: resolvesSecondAttacker = IsKingSafeFromBishop(new_i, new_j, attacker2.row, attacker2.column); break;
34                     case 2: resolvesSecondAttacker = IsKingSafeFromRook(new_i, new_j, attacker2.row, attacker2.column); break;
35                     case 3: resolvesSecondAttacker = IsKingSafeFromKnight(new_i, new_j, attacker2.row, attacker2.column); break;
36                 }
37                 if (resolvesFirstAttacker && resolvesSecondAttacker && IsKingSafeAt(new_i, new_j, gameState)) {
38                     legalMoves.Add(kingMove);
39                 }
40             }
41         }
42     }
43     // if king is in check we cannot castle
44     if (attacker1.type != -1) {
45         return;
46     }
47     // try to short castle
48     if ((gameState.whoMoves == 'w' && gameState.canWhite_0_0)
49         || (gameState.whoMoves == 'b' && gameState.canBlack_0_0)) {
50         // it's implied the king is on its starting square
51         if (boardConfiguration[old_i, old_j + 1] == '-' &&
52             boardConfiguration[old_i, old_j + 2] == '-') { // no blocking pieces
53             // not castling through or into check
54             if (IsKingSafeAt(old_i, old_j + 1, gameState) && IsKingSafeAt(old_i, old_j + 2, gameState)) {
55                 legalMoves.Add(new IndexMove(old_i, old_j, old_i, old_j + 2));
56             }
57         }
58     }
59     // try to long castle
60     if ((gameState.whoMoves == 'w' && gameState.canWhite_0_0_0)
61         || (gameState.whoMoves == 'b' && gameState.canBlack_0_0_0)) {
62         // it's implied the king is on its starting square
63         if (boardConfiguration[old_i, old_j - 1] == '-' &&
64             boardConfiguration[old_i, old_j - 2] == '-' &&
65             boardConfiguration[old_i, old_j - 3] == '-') { // no blocking pieces
66             // not castling through or into check
67             if (IsKingSafeAt(old_i, old_j - 1, gameState) && IsKingSafeAt(old_i, old_j - 2, gameState)) {
68                 legalMoves.Add(new IndexMove(old_i, old_j, old_i, old_j - 2));
69             }
70         }
71     }
72 }

```

Figura 42: Metoda AddLegalKingMoves() din **MoveGenerator**

```

1  async void OnRunTestsButtonClicked() {
2      string filePath = Path.Combine(Application.streamingAssetsPath, "perft.txt");
3      string outPath = Path.Combine(Application.streamingAssetsPath, "outputChess.txt");
4      if (File.Exists(filePath)) {
5          using StreamReader reader = new(filePath);
6          using StreamWriter writer = new(outPath, false);
7          string line;
8          int idx = 0;
9          FENskipChunk = 50; // make this larger to process less data (2 is 50% of the data, 10 is 10% of the data etc.)
10         while ((line = reader.ReadLine()) != null) {
11             if (idx % FENskipChunk != 0) {
12                 ++idx;
13                 continue; // only process one portion of the data
14             }
15             string[] parts = line.Split(',');
16             string fen = parts[0];
17             GameStateManager.Instance.GenerateGameState(fen);
18             long[] legalMovesDepth = new long[7];
19             for (int i = 1; i <= 6; ++i) {
20                 legalMovesDepth[i] = long.Parse(parts[i]);
21             }
22             bool ok = true;
23             for (int perftDepth = 1; perftDepth <= 5; ++perftDepth) { // increase upper limit for deeper searches
24                 maxDepth = perftDepth;
25                 long result = 0;
26                 await Task.Run(() => result = SearchPositions(GameStateManager.Instance.globalGameState, 0));
27                 if (result != legalMovesDepth[perftDepth]) {
28                     ok = false;
29                     writer.WriteLine("Incorrect results for depth 1: FEN: " + fen +
30                         " ; expected " + legalMovesDepth[perftDepth] + " got " + result);
31                 }
32             }
33             UnityEngine.Debug.Log(fen + "was checked, it is" + (ok ? "" : " not") + " ok");
34             ++idx;
35         }
36     } else {
37         UnityEngine.Debug.Log("file not found");
38     }
39 }

```

Figura 43: Metoda de testare pentru valori Perft

```

1 #!/bin/bash
2
3 # Input file containing FEN strings
4 FEN_FILE="FENSv2.txt"
5
6 # Output file for Stockfish analysis results
7 OUTPUT_FILE="FENS_Outputs.txt"
8
9 # Move time for Stockfish analysis (in milliseconds)
10 MOVETIME=5000
11
12 # Estimated analysis time per FEN (seconds)
13 ANALYSIS_TIME=$((MOVETIME / 1000 + 1)) # Move time in seconds + 1 second buffer
14
15 # Count lines in FEN_FILE (estimated number of FEN strings)
16 NUM_FENS=$(wc -l < "$FEN_FILE")
17
18 # Estimated total analysis time (seconds)
19 ESTIMATED_TOTAL_TIME=$((NUM_FENS * ANALYSIS_TIME))
20
21 # Calculate estimated time in minutes and seconds
22 ESTIMATED_MINUTES=$((ESTIMATED_TOTAL_TIME / 60))
23 ESTIMATED_SECONDS=$((ESTIMATED_TOTAL_TIME % 60))
24
25 echo -e "FEN\nTree and eval Info\nbest move\n" > "$OUTPUT_FILE"
26 echo "Getting Stockfish evals..."
27 echo "Estimated analysis time: $ESTIMATED_MINUTES minutes and $ESTIMATED_SECONDS seconds"
28
29 # Loop through each FEN string in the file
30 while IFS= read -r FEN; do
31     # Construct the command with the current FEN
32     # Convert ms to s and add 1 second buffer
33     COMMAND="echo -e \"position fen $FEN\\n go movetime $MOVETIME\"; sleep $((MOVETIME/1000 + 1))"
34
35     # Execute the command and capture output using Stockfish
36     OUTPUT=$(eval "$COMMAND" | ./stockfish-windows-x86-64-avx2.exe | tail -3)
37     echo "Analysis done for $FEN"
38
39     # Write FEN and analysis result to output file
40     echo -e "$FEN\\n$OUTPUT\\n" >> "$OUTPUT_FILE"
41 done < "$FEN_FILE"
42
43 echo "Analysis results written to $OUTPUT_FILE"
44

```

Figura 44: Scriptul de evaluare a FEN-urilor prin Stockfish