
UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ


Lucrare de licență

# APLICAȚIE SINGLEPLAYER-MULTIPLAYER DE ȘAH

Absolvent

Căpățînă Răzvan Nicolae

Coordonator științific

Conf. Dr. Rusu Cristian

București, iunie 2025

## Rezumat

Scopul principal al lucrării a fost dezvoltarea unui program capabil să joace șah cu o performanță cât mai mare posibilă, în același timp ținând cont de limitările de procesare și de stocare ale unui dispozitiv nespecializat pe această problemă și de uz general, precum un laptop. Utilizatorii vor avea oportunitatea să înfrunte un agent de șah optimizat, dar care menține o performanță notabilă în timpul jocului. Tehnicile și implementările abordate în această lucrare pot fi aplicate în cadrul oricărui alt joc determinist, cu informație completă și mutări alternante între doi jucători.

Aplicația a luat în considerare și potențialul său rețelistic, având dezvoltat un sistem de multiplayer, prin care utilizatorii pot juca șah între ei. Atât implementarea, cât și arhitectura de client-server folosită pot avea utilizări într-o gamă variată de proiecte, depășind sfera jocurilor.



## Abstract

The main goal of this thesis was to develop a program capable of playing chess with the highest possible performance while taking into account the processing and storage limitations of a general-purpose device that is not specialized for this particular problem, such as a laptop. Users will have the opportunity to face an optimized chess agent that maintains a notable performance during the game. The techniques and implementations presented in this work can be applied to any other deterministic game with complete information and alternating moves between two players.

The application also took into account its networking potential, having developed a multiplayer system through which users can play chess against each other. Both the implementation and the client-server architecture used can be applied in a wide range of projects, beyond the scope of games.

# Cuprins

# Capitolul 1

# Introducere

## 1.1 Motivație

Jocurile au reprezentat un subiect de interes în Informatică încă de la început, fiind printre primele concepte analizate în domeniu. În privința jocului de șah au existat diverse rezultate obținute, un exemplu fiind algoritmul lui Claude Shannon [1] care descrie cum ar putea o mașină să joace acest joc. Astfel de algoritmi pot fi folosiți pentru a testa limitele de memorie și timp ale diverselor mașini. De asemenea, aceste implementări pot fi ulterior modificate pentru a permite rezolvarea unor probleme întâlnite în situații din lumea reală, în general fiind vorba de optimizări ale unei funcții de cost luând în calcul anumite reguli sau constrângeri.

În prezent, Engine-urile de Șah, precum Stockfish [1], Leela Chess Zero [2] și AlphaZero, sunt un subiect activ de cercetare, noutatea fiind integrarea ideilor clasice cu modele de învățare automată. Tehnicile vechi rămân în continuare competitive, deoarece acestea necesită un efort de calcul moderat în comparație cu abordările noi, cum ar fi, de exemplu, implementările oferite de Leela Chess Zero și AlphaZero, care se bazează pe Reinforcement Learning.

Din punct de vedere personal, șahul a reprezentat un joc de interes pentru mine, iar prin această aplicație mi-am propus să aflu ce performanțe poate atinge un bot dedicat acestui joc, luând de asemenea în calcul limitările hardware existente.

Această lucrare mi-a permis să îmbin și să aprofundez ramuri variate ale informaticii cu scopul de a obține o aplicație eficientă și de sine stătătoare. Astfel, proiectul cuprinde principii ale programării orientate pe obiecte, structuri de date eficiente, optimizări la nivel de bits, rețelistică, grafică, precum și programare cu fire multiple de execuție.

Arhitectura modulară pe care am utilizat-o permite refolosirea engine-ului de șah dezvoltat, dar și modificarea acestuia pentru a putea fi folosit în alte jocuri similare.

Logica de comunicare dintre client și server, ce are loc în timpul unei sesiuni multipla-

[1] Algoritmul lui Claude Shannon, detalii accesibile la https://www.chessprogramming.org/Claude_Shannon

yer, este de asemenea modulară și poate fi ușor modificată pentru refolosirea sa în diverse aplicații, dincolo de sfera jocurilor.

## 1.2 Structura Lucrării

Lucrarea de licență cuprinde capitole ce vor prezenta diverse subdomenii utilizate:

- **Dezvoltarea Proiectului**, având următoarele subcapitole:
  - **Arhitectura Generală**, în care sunt menționate bibliotecile folosite și exemplificate aplicarea principiilor programării orientate pe obiecte pentru modularizarea codului.
  - **Comunicarea Client-Server în timpul Sesiunii de Multiplayer**, în care este prezentată logica de comunicare dintre client și server, dar și metodele folosite pentru sincronizarea acestora. Arhitectura implementată asigură comunicarea la distanță între doi utilizatori, facilitând buna desfășurare a unei sesiuni de multiplayer.
  - **Algoritmul Minimax**, alături de alte optimizări precum Alpha-Beta Pruning, paralelizare, Zobrist Hashing, dar și memoizarea stărilor.
  - **Optimizări folosite pentru Generarea Mutărilor de Șah**, în care sunt menționate abordările realizate la nivel de bits pentru a genera eficient mutările posibile dintr-o configurație oarecare a tablei de șah.
- **Rezultate Obținute**, ce cuprinde:
  - **Performanță**, care a fost obținută prin utilizarea aplicației în meciuri contra agenților găzduiți de site-uri precum chess.com [3] și lichess.org [4].
  - **Studiu asupra Avantajului de un Tempo**, în care agentul obținut este folosit pentru a analiza dacă avantajul de o mutare al jucătorului alb este suficient pentru a forța o victorie în cazul în care ambii participanți pornesc cu material egal și joacă perfect.
  - **Automatizarea Evaluării**, realizată prin implementarea unui script de Python care interacționează folosind Inter-Process Communication cu aplicația dezvoltată.
- **Concluzii**, alături și de câteva zone unde există îmbunătățiri pentru performanța agentului.

# Capitolul 2

# Dezvoltarea Proiectului

## 2.1 Arhitectura Generală

### 2.1.1 Configurarea Inițială

Primul pas în dezvoltare a fost implementarea unui sistem de interfețe și butoane pentru a permite navigarea utilizatorului între modurile de funcționare ale aplicației: singleplayer, unde utilizatorul va juca împotriva unui agent, și multiplayer, unde utilizatorii pot juca între ei.

Mediul de dezvoltare folosit a fost Microsoft Visual Studio 2022 [5]. Implementarea a fost realizată în limbajul `C++`, 64-bit, iar pentru grafică s-a folosit OpenGL. Codul utilizează principiile programării orientate pe obiecte: încapsularea, abstractizarea, moștenirea și polimorfismul, dar și șabloane de proiectare, precum singleton și factory.

Următoarele biblioteci și header-e au fost folosite în cadrul dezvoltării:

- Pentru OpenGL:
  - GLFW [6]
  - GLEW [7]
  - stb_image.h [8]
  - GLM [9]
  - FreeType [10]
- Pentru transmiterea de date în rețeaua locală:
  - ENet [11]
- Pentru efecte sonore:
  - IrrKlang [12]

Pentru utilizarea modului de multiplayer în cazul utilizatorilor aflați în LAN-uri diferite a fost folosită aplicația LogMeIn Hamachi [13], care facilitează conexiunea prin crearea unui VPN.

Pentru texturile pieselor au fost folosite resursele nealterate de pe Wikimedia [14], licențiate sub CC BY-SA 3.0. [1]

Efectele sonore sunt preluate de pe Pixabay [15], fișierele fiind licențiate sub Pixabay Content. [2]

Pentru afișarea de text în cadrul aplicației a fost folosit font-ul Open Sans Regular [16], licențiat sub Apache 2.0. [3]

S-a folosit Git pentru controlul versiunilor în timpul dezvoltării proiectului, repository-ul fiind accesibil la [17].

### 2.1.2 Implementarea Sistemului de Meniuri

Sistemul de meniuri este un exemplu foarte util pentru aplicarea practică a principiilor programării orientate pe obiecte.

Meniurile dintr-o aplicație pot fi interpretate drept un graf orientat, unde nodurile sunt interfețele vizuale ale meniurilor, iar muchiile reprezintă butoanele. Astfel, aplicația se află la un moment de timp într-o anumită stare, iar prin intermediul butoanelor poate traversa într-o altă stare. Acest graf poate admite și bucle, existând butoane care nu schimbă interfața vizuală curentă, cum ar fi butoanele prezente într-un meniu de setări al aplicației.

Avem astfel funcția de tranziție:

$$t : V \times E \to V,$$

unde V este mulțimea meniurilor, iar E este mulțimea butoanelor.

Exemplu de tranziții:
$t(MainMenu, Settings) = SettingsMenu$
$t(SettingsMenu, TurnSoundON/OFF) = SettingsMenu$
$t(MainMenu, Exit) = Exiting,$
unde Exiting este un nod santinelă, prin care aplicația este anunțată că trebuie să-și încheie execuția.

În practică această arhitectură a meniurilor poate fi realizată prin implementarea unei clase VisualInterface, care conține un vector de pointers către obiecte de tip Entity. Clasa Entity este una abstractă, având metodele draw() și update() pur virtuale. Din această

[1] Licența Creative Commons Attribution-ShareAlike 3.0 (CC BY-SA 3.0), accesibilă la https://creativecommons.org/licenses/by-sa/3.0/
[2] Licența Pixabay Content, accesibilă la https://pixabay.com/service/license-summary/
[3] Licența Apache 2.0, accesibilă la https://www.apache.org/licenses/LICENSE-2.0

clasă se moștenesc alte entități concrete, precum TexturableEntity, TextEntity și Button, care modelează conținutul grafic al unei interfețe vizuale.

Deoarece draw() și update() sunt metode virtuale, clasa VisualInterface trebuie doar să parcurgă toate entitățile atașate ei și să apeleze metodele respective, acestea executând logica cea mai joasă din arborele de moșteniri, conform polimorfismului la execuție.

Rămâne doar implementarea unei clase singleton Game, care conține o structură de date de tip *std::map* de forma:

$$m : V \to P,$$

unde V sunt nodurile din graf, care în practică pot fi implementate drept un enum class, iar P este mulțimea pointer-ilor către instanțele de VisualInterface aferente stărilor. Butoanele vor implementa schimbarea din nodul curent în alt nod în suprascrierea lor a metodei update(), iar clasa Game doar va rula o buclă de forma:

```
while (aplicatiaRuleaza)
{
    m[stareCurenta]->update();
    m[stareCurenta]->draw();
}
```

## 2.2 Comunicarea Client-Server în timpul Sesiunii de Multiplayer

Comunicarea dintre server și clienți stă la baza funcționării modului multiplayer oferit de aplicație.

Această comunicare este formată din mai multe etape: cea inițială, de scurtă durată, unde sunt transmise informațiile elementare, urmată de a doua etapă, care are loc pe tot parcursul meciului de șah și care asigură buna desfășurare a acestuia.

### 2.2.1 Inițializarea Comunicării

Aceasta este prima etapă și constă în realizarea cu succes a conexiunii dintre server și clienți și în transmiterea datelor elementare pentru continuarea interacțiunii.

Ordinea în care au loc transmisiile de informație diferă în funcție de clientul implicat, instanța de server folosită fiind mereu cea care aparține utilizatorului creator al sesiunii de joc.

În cazul utilizatorului creator, clientul său se va conecta la instanța locală de server, după care va trimite către acesta configurația tablei de șah, numele utilizatorului, dar și culoarea aleasă de jucător din meniu. Serverul va păstra toate aceste informații pentru a putea răspunde la viitoarele cereri.

Atunci când celălalt utilizator se conectează, acesta va trimite numele său de jucător către server și va cere culoarea atribuită lui și configurația curentă a tablei de șah. Dacă la momentul curent serverul nu poate oferi aceste informații atunci va anunța jucatorul, acesta reîncercând mai târziu să le ceara. De exemplu, utilizatorul creator este cel ce își alege culoarea de joc din meniu, serverul fiind incapabil să atribuie o culoare celuilalt jucător până nu o află pe cea de la creatorul sesiunii.

Fiecare înteracțiune prezentată mai sus poate eșua: conexiunea nu s-a realizat cu succes, mesajul nu a ajuns la destinație sau informațiile cerute nu sunt incă disponibile. Pentru aceste cazuri a fost implementat un retry mechanism prin care clienții nu trec la următoarea etapă dacă nu au primit informațiile necesare pentru pașii anteriori, reîncercând până au succes. Toată această logica a fost implementată pe firul principal de execuție al aplicației, unde rulează și interacțiunea cu sistemul de meniuri, incluzând butoanele. Astfel, pentru a nu bloca interfața vizuală, acest retry mechanism este non-blocking. Transmiterile de informație au loc maxim o singură dată în cadrul aceleiași iterații a buclei globale de update() pentru aplicație. În caz de eșec se reîncearcă la iterația următoare.

### 2.2.2 Comunicarea din timpul jocului

După transmiterea informațiilor de bază între server și clienți, jocul poate începe.

În această etapă există implementat un sistem de pinging prin care serverul și clienții știu dacă există în continuare o conexiune activă. Astfel, clienții transmit periodic un mesaj de ping catre server, iar server-ul răspunde periodic cu alt mesaj de ping care conține de asemenea și starea de conexiune a celuilalt client. Această ultimă informație este ulterior afișată pe ecran pentru fiecare client în parte, utilizatorii cunoscând astfel dacă există o problemă de conexiune la nivelul serverului sau la nivelul jucătorului oponent. Serverul își află aceste stări de conexiune ale clienților în funcție de ultimul moment de timp când fiecare dintre aceștia a trimis un ping.

Atunci când are loc o mutare pe tabla de șah clientul utilizatorului ce a realizat mutarea va transmite către server noua configurație a tablei de șah și mutarea efectuată. Serverul va salva intern aceste informații și le va transmite mai departe către celalalt client, urmând ca acesta să-și modifice la rândul său configurația tablei de șah. Procedeul este apoi repetat.

Serverul are de asemenea implementată o structură de date de tip *std::map*, unde cheia primară este concatenarea dintre IP-ul utilizatorului și port. Această structură permite

serverului să refuze orice nouă conexiune care ar duce la mai mult de 2 clienți activi simultan. În cazul în care utilizatorul care nu este creatorul sesiunii nu a mai dat ping de un anumit timp, serverul îl va elimina din structura de date, micșorând astfel numărul de clienți activi. Acest lucru permite ca utilizatorul care nu este creator să se deconecteze în timpul jocului și să reintre la alt moment de timp și posibil cu alt nume sau să se deconecteze definitiv și alt utilizator să intre în locul său și să preia activitatea, serverul retrimițând culoarea atribuită și configurația curentă a tablei de șah. Întotdeauna la reconectare clientul va relua cele două etape ale comunicării menționate.

Un demo realizat pentru a prezenta modul de multiplayer al aplicației poate fi vizualizat aici [18].

## 2.3 Algoritmul Minimax

Atât algoritmul prezentat mai jos, cât și toate optimizările aferente acestuia, au fost implementate de la zero în cadrul proiectului, limbajul folosit fiind `C++`.

Algoritmul Minimax [19, 20] se bazează pe căutarea exhaustivă în spațiul de configurații posibile pentru a găsi cea mai favorabilă viitoare stare în care agentul poate ajunge, indiferent de cât de bine joacă oponentul.

Astfel, algoritmul presupune că adversarul joacă optim și încearcă să găsească cea mai bună mutare din starea curentă, generând un arbore unde nodurile sunt configurații ale tablei de șah, iar muchiile reprezintă mutările.

Aplicația folosește acest algoritm pentru a genera următoarea mutare a agentului, în rădăcina arborelui aflându-se mereu o configurație în care rândul celui ce mută este acesta. Dacă se consideră stratul rădăcinii ca fiind stratul de nivel 0, atunci pe orice strat de nivel $2\times n$ din arbore este rândul agentului, iar pe cele de forma $2\times n + 1$ este rândul oponentului, unde $n \in \mathbb{N}$. Ambii jucători au ca scop să-și maximizeze câștigul, acest lucru fiind echivalent cu minimizarea câștigului adversarului în cazul jocului de șah.

Jocul de șah este complex, iar limitările hardware nu permit ca algoritmul Minimax să ruleze până în nodurile frunză ale arborelui, care ar corespunde stărilor unde jocul s-a incheiat. Astfel, se setează o adâncime maximă până la care algoritmul va rula. În momentul în care se ajunge la acest maxim se va calcula un scor pentru configurația curentă bazat pe o evaluare statică și euristică a stării. Această valoare este apoi propagată în sus prin arbore pentru a atribui scoruri nodurilor ce nu sunt frunză.

În cazul acestui proiect, s-a folosit un concept modificat al celui de mai sus, adâncimea utilizată fiind 4. Evaluarea statică se bazează pe calculul diferenței dintre materialele fiecărui jucător și a diferenței dintre zonele potențiale de atac pe care le au.

### 2.3.1 Optimizarea Alpha-Beta Pruning

Asupra algoritmului Minimax se poate aplica optimizarea Alpha-Beta Pruning [20], care permite ca subarbori întregi să fie eliminați din cadrul parcurgerii, pe raționamentul că aceștia nu vor contribui niciodată la soluția ce ajunge în rădăcină. Impactul pe care îl are această optimizare depinde de ordinea în care coborâm pe subarbori. În general s-a observat faptul că parcurgerea la început a subarborilor în care s-a ajuns folosind mutări dramatice, cum ar fi capturile de piese, șahurile sau promovările de pion, contribuie la mai multe tăieturi din partea optimizării Alpha-Beta. Aplicația ține cont de acest lucru prin faptul că pentru o configurație a tablei de șah generează mai întâi mutările de regină, apoi cele de tură, nebun, cal, pion și în final cele de rege. Aceastea vor fi luate în aceeași ordine și în timpul parcurgerii Minimax.

### 2.3.2 Introducerea Paralelizării

Algoritmul Minimax poate fi paralelizat, astfel încât în momentul în care începem extinderea din nodul rădăcină să instanțiem câte un fir de execuție separat pentru fiecare subarbore. În figura 2.1 se pot observa posibilele abordări pentru a distribui munca între firele de execuție.

Prima ilustrare este cea mai ușor de implementat, algoritmul Minimax având loc în întregime pe firul principal de execuție. Problema principală cu această abordare este faptul că întreg sistemul de meniuri și de înterfețe vizuale vor fi blocate, deoarece în cadrul aplicației dezvoltate acestea rulează pe firul principal de execuție. De exemplu, utilizatorul va fi incapabil să utilizeze butoanele.

A doua ilustrare nu rezolvă această problemă, firul principal de execuție fiind în continuare nevoit să aștepte pentru ca algoritmul Minimax să se încheie. Această așteptare în schimb este mai scurtă, deoarece munca este distribuită între mai multe fire.

A treia implementare este folosită de aplicație și s-a dovedit cea mai bună în practică, deoarece nu blochează firul principal de execuție și se ocupă și de distribuirea muncii între fire distincte. Firul principal creează un alt fir ce va administra rularea Minimax. Acesta va distribui algoritmul pe mai multe fire și va astepta ca fiecare dintre acestea să-și încheie activitatea, combinând rezultatele obținute. Firul principal de execuție va putea să ruleze între timp altă logică necesară aplicației, fiind nevoit doar să verifice periodic dacă firul de executie ce administrează algoritmul Minimax și-a încheiat activitatea și poate furniza un răspuns final.

Introducerea paralelizării pentru algoritmul Minimax aduce beneficii majore performanței proiectului, însă există o notabilă pierdere, dată de faptul că Alpha-Beta Pruning nu va mai putea realiza tăieturi la nivelul primului strat din arbore, această optimizare funcționând doar dacă subarborii sunt parcurși secvențial. Tăieturile vor avea loc în continuare la nivelele inferioare, în cadrul fiecărui fir de execuție, aici menținându-se

proprietatea de secvențialitate a parcurgerii.

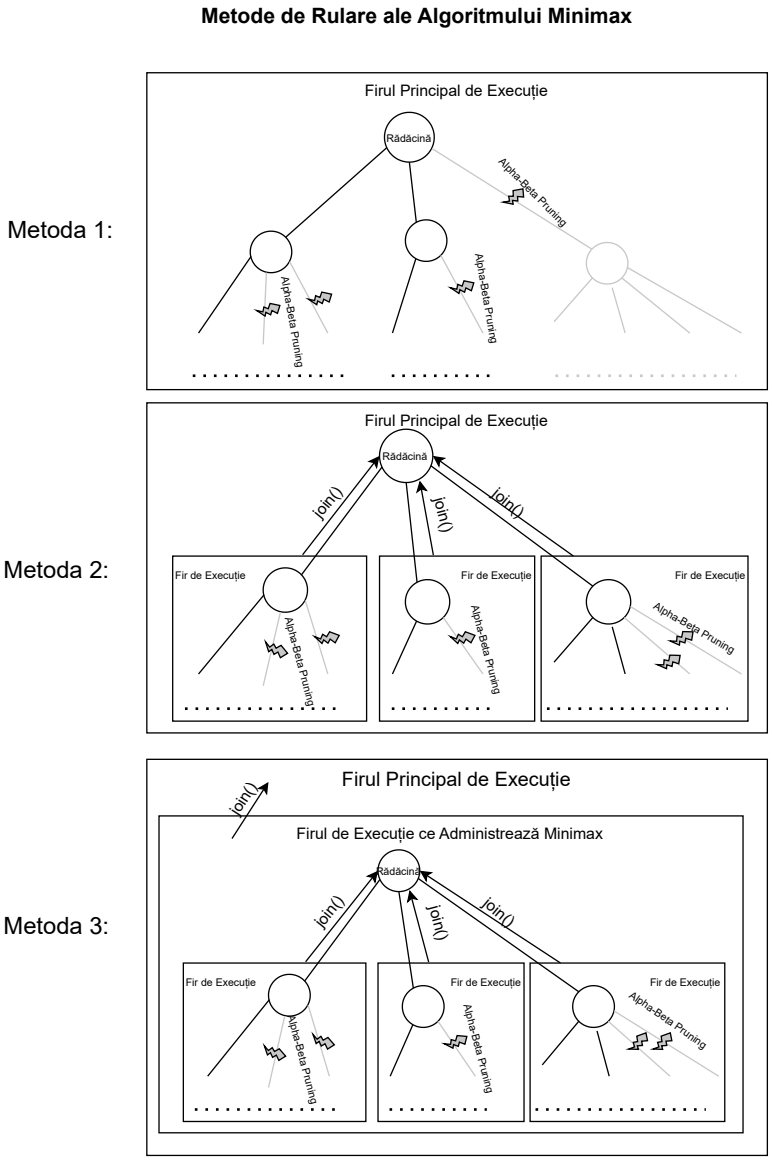



Figura 2.1: Paralelizare Minimax

### 2.3.3 Zobrist Hashing

Zobrist Hashing [21, 22] este o tehnică de a comprima configurația unei table de șah într-o valoare întreagă pe 64 de bits de tipul unsigned long long. Hash-ul se bazează pe generarea aleatoare a unor valori întregi asociate fiecărei perechi de forma (poziție celula, piesă). Aceste valori sunt apoi folosite împreună cu operația de tip xor pentru a construi o valoare finală asociată configurației curente a tablei de șah.

Metoda poate crea coliziuni, datorită faptului că $2^{64}$ este cu mult mai mic decât

numărul posibil de configurații distincte de șah. Probabilitatea să aibă loc o coliziune este însă mică, mai ales dacă numerele generate aleator provin dintr-o distribuție ce tinde a fi uniformă.

### 2.3.4 Memoizarea Transpozițiilor

Există configurații ale tablei de șah în care se poate ajunge în diverse moduri, folosind lanțuri diferite de mutări. Există de asemenea configurații în care se poate cicla la infinit, ambii jucători aplicând repetat aceleași mutări. Ambele tipuri de configurații poartă numele de transpoziții [23, 20]. Algoritmul Minimax va duplica aceste stări în cadrul arborelui său, chiar dacă informația obținută din aceste noduri este aceeași.

Memoizarea Stărilor [4] își propune să rezolve această problemă. Astfel, pentru fiecare stare vizitată din arbore salvăm informațiile obținute în urma analizei subarborelui său. Dacă ulterior ajungem încă o dată în această stare, atunci înseamnă că este vorba despre o transpoziție și putem folosi informațiile salvate anterior, fără să mai parcurgem subarborele. Pentru salvarea informațiilor despre configurații se va folosi o structură de date de forma *std::map*, unde cheia primară este Zobrist Hash-ul stării.

Memoizarea poate fi utilizată atunci când avem o parcurgere depth-first a arborelui, pe același fir de execuție. În cazul în care munca este distribuită între fire distincte avem următoarele posibilități:

- Avem un sistem global de memoizare și folosim o variabilă de tip mutex pentru a gestiona citirea din sistem și modificarea sistemului de către firele de execuție. Această abordare a fost testată în cadrul proiectului și a dus la un overhead considerabil datorat excluderii mutuale, beneficiile memoizării ne mai fiind vizibile.

- Avem câte un sistem de memoizare pentru fiecare fir de execuție în parte. Această abordare elimină necesitatea utilizării unei variabile de tip mutex, dar în schimb scade probabilitatea să avem un memoization hit. În practică a dat rezultate bune, fiind metoda folosită de proiect.

În ambele abordări sistemele de memoizare sunt curățate între rulări consecutive ale algoritmului Minimax. Acest lucru a fost decis pentru a ține sub control cantitatea de memorie RAM alocată, care altfel ar crește neîncetat. Un dezavantaj generat este faptul că probabilitatea unui memoization hit a scăzut.

### 2.3.5 Adâncime de Căutare Adaptivă pentru Minimax

Dacă impunem ca arborele generat de algoritmul Minimax să aibă mereu o adâncime fixă riscăm să nu ne putem adapta evoluției jocului.

[4] Conceptul de Memoizare, detalii accesibile la https://en.wikipedia.org/wiki/Memoization

La începutul unui meci de șah pe tablă se află numărul maxim de piese posibile, branching factor-ul arborelui Minimax fiind maximal în timpul acestei perioade a meciului. Pe măsură ce avansăm către încheierea jocului branching factor-ul tinde să se micșoreze. Acest lucru înseamnă că păstrarea unei adâncimi fixe pentru arborele Minimax va duce de fapt la micșorarea treptată a numărului de stări vizitate în cadrul unei rulări a algoritmului, compromițând astfel performanța agentului.

La finalul unui joc există pericolul ca agentul să nu poată observe promovările de pioni, deoarece acestea se află la o distanță prea mare în viitor, chiar dacă practic le-am putea calcula, deoarece nu au rămas multe mutări disponibile.

Pentru a rezolva aceste probleme putem ca în timpul rulării algoritmului Minimax să stabilim un număr dorit de noduri vizitate în loc de o adâncime maximă.

Astfel, pentru un nod oarecare $x$ din arbore ne propunem să vizităm $k \in \mathbb{N}^*$ stări în subarborele său. Dacă nodul $x$ are $n \in \mathbb{N}$ fii, $n$ fiind și branching factor-ul lui $x$, atunci pentru fiecare fiu al lui $x$ dorim să vizităm $\lfloor (k-1)/n \rfloor$ stări. În situația în care parcurgem secvențial fiii lui $x$ și ne întoarcem dintr-un fiu $f_i$ cu mai puține noduri vizitate decât ne propusesem, atunci viitorilor fii ai lui $x$ ce urmează a fi parcurși le vom distribui uniform diferența dintre cât trebuia să vizităm în subarborele lui $f_i$ și cât am reusit de fapt.

Atunci când ne oprim într-un subarbore mai devreme decât ne-am așteptat nu vom termina mai repede parcurgerea Minimax, ci vom prefera să ne concentrăm atenția asupra altor regiuni din arbore. Practic, timpul câștigat va fi exploatat în altă zonă. Principalele motive pentru care ne-am opri mai devreme decât anticipat ar fi tăieturile Alpha-Beta sau faptul că am ajuns într-o stare terminală a jocului.

Această adâncime de căutare adaptivă funcționează în cazul în care parcurgerea arborelui este depth-first și pe același fir de execuție.

În implementarea sa, proiectul folosește în continuare o adâncime minimă la care trebuie să ajungă expansiunea arborelui, dar îmbină această idee cu o căutare de lungime adaptivă, care este aplicată la nivelul fiecarui fir de execuție din paralelizarea algoritmului Minimax. Oprirea explorării dintr-un nod are loc dacă ambele condiții de oprire sunt îndeplinite, acesta aflându-se la o adâncime cel puțin egală cu cea minimă setată și indeplinind simultan numărul propus de stări vizitate.

S-a experimentat cu valori cuprinse între 20,000,000 și 27,500,000 pentru numărul de stări explorate în intregul arbore, aplicația generând mutarea optimă în aproximativ 7-8 secunde. Acest număr trebuie ales astfel încât să existe o balanță între performanța agentului și timpul de execuție necesar.

## 2.4 Optimizări folosite pentru Generarea Mutărilor de Șah

Generarea mutărilor pentru o configurație dată a tablei de șah reprezintă bottleneck-ul algoritmului Minimax. Dacă generarea mutărilor este una eficientă, atunci timpul pierdut în cadrul unui singur nod din arborele Minimax scade, lucru ce permite o parcurgere mai amplă a arborelui, sporind astfel performanța agentului. Proiectul folosește diverse tehnici pentru a spori eficiența, acestea fiind implementate de la zero în limbajul `C++` [24].

### 2.4.1 Modul de Stocare al Datelor

Putem observa faptul că dimensiunea tablei de șah este $8 \times 8 = 64$, care este numărul de bits al unei variabile întregi de tipul *long long* sau *unsigned long long*. Acest lucru ne permite să stocăm o singură informație binară despre întreaga configurație a tablei doar într-o singura variabilă pe 64 de bits. Folosind suficiente astfel de variabile putem avea stocate toate informațiile utile despre configurația curentă a tablei. Astfel, vom avea 6 variabile pentru fiecare tip de piesă de culoare albă și alte 6 variabile pentru fiecare tip de piesă de culoare neagră.

### 2.4.2 Precalculările Folosite

O primă funcție utilă în cadrul implementării proiectului este cea care pentru o putere de 2 din mulțimea $\{2^0, 2^1, 2^2, \dots, 2^{63}\}$ întoarce exponentul puterii. Practic, pentru o mască de 64 de bits, unde exact un bit este 1 funcția va întoarce poziția acestuia. Dorim ca această funcție să ruleze în timp constant $O(1)$. Funcția $\log_2$ este exclusă, timpul de calcul al acesteia fiind $O(\log)$.

În cadrul aplicației a fost folosită o funcție hash de forma:

$$hash : \{2^0, 2^1, 2^2, \dots, 2^{63}\} \to \{0, 1, 2, \dots, 63\}$$
$$, \text{ unde } \text{hash}(x) = x \bmod \text{MODULO}$$

Această funcție rulează în $O(1)$ dacă nu există coliziuni și ne permite să răspundem la problema menționată.

Variabila MODULO trebuie aleasă astfel încât funcția hash să fie injectivă. În această situație funcția devine chiar bijectivă, deoarece cardinalul domeniului este același cu al codomeniului.

În urma rulării unui brute-force s-a ajuns la concluzia că valoarea minimă a lui MODULO pentru ca funcția hash să fie injectivă este 67. Această valoare este apropiată și de optimul memoriei, deoarece o condiție necesară pentru soluție este MODULO $\geq 64$.

Funcția $\text{hash}^{-1}(x)$ este $2^x$ și se poate calcula în timp $O(1)$, fiind o shift-are a lui 1 la stânga cu x bits.

Alte funcții utile a fi precalculate sunt cele care pentru o anumită piesă și o anumită poziție a sa pe tabla de șah calculează o mască de bits care reprezintă toate pozițiile unde piesa respectivă poate ataca. Astfel de precalculări au fost folosite pentru ture, cai, nebuni, regine și regi.

Pentru a extrage concret mutările din astfel de zone de atac se poate utiliza o implementare de forma:

```
while (mascaBits)
{
    celMaiPutinSemnificativBit = (mascaBits & ((~mascaBits) + 1));
    mascaBits ^= celMaiPutinSemnificativBit;
}
```

Această structură repetitivă rulează în timp $O(k)$, unde k este numarul de bits de 1 din masca primită. Complexitatea este în medie mai bună decât o implementare simplă, care ar rula în $O(64)$.

În cazul pionilor, zonele de atac ale acestora pot fi calculate în grup și în timp constant, având nevoie doar de precalculările măștilor de bits ce determină fiecare coloană în parte. În figura 2.2 se observă cum putem obține toate pozițiile de atac la dreapta ale pionilor aplicând operații de complexitate constantă. Astfel, asupra măștii de bits pentru pionii de o aceeași culoare aplicăm &-logic cu negația măștii de bits precalculate pentru ultima coloană a tablei de șah, deoarece piesele de pe ultima coloană nu au un posibil atac la dreapta. După aceea aplicăm o shift-are de 7 bits la dreapta pentru a muta valorile de 1 în pozițiile zonelor de atac.

Rămâne doar să aplicăm o operație de &-logic între rezultatul obținut și o mască de bits ce semnifică pozițiile tuturor pieselor de culoare opusă, obținând astfel piesele adversarului aflate în zona de atac la dreapta a pionilor.

Pentru atacul pionilor la stânga raționamentul este același, doar că este folosită negația măștii de bits precalculate pentru prima coloană în loc de ultima, iar shift-area este în continuare la dreapta, dar de 9 bits în loc de 7.

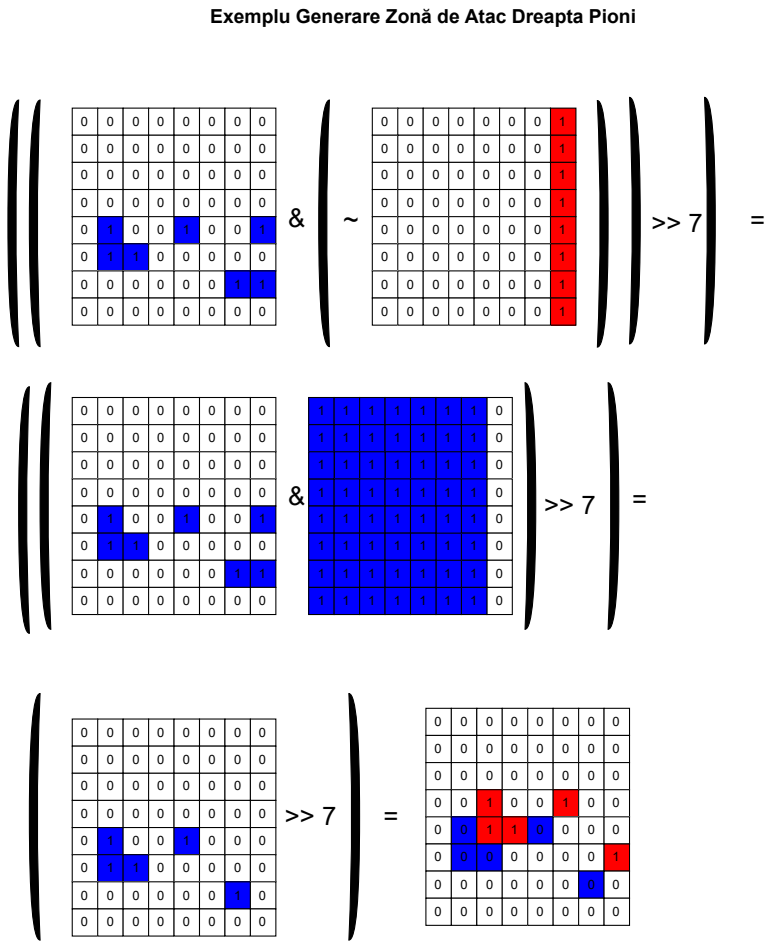



Figura 2.2: Exemplu Zonă de Atac Pioni, idee inspirată din [24]

Pentru toate precalculările menționate nu este importantă complexitatea obținerii lor. Acestea sunt calculate o singură dată la început și folosite pe toată durata unui meci și chiar și între meciuri diferite. Important este că acestea oferă acces la înformație în timp constant și cu overhead cât mai redus.

# Capitolul 3

# Rezultate Obținute

Codul sursă pentru toate rezultatele prezentate în acest capitol este disponibil în întregimea sa online. [1]

## 3.1 Performanță

Aplicația a jucat câteva meciuri impotriva agenților gazduiți de site-urile chess.com [3] și lichess.org [4].

S-a ajung la concluzia că bot-ul implementat are un rating de aproximativ 2100-2200, fiind capabil să câștige meciuri împotriva unor adversari de 1600, 1800, 2000 și respectiv 2200 rating.

În figura 3.1 se poate observa distribuția rating-urilor tuturor jucătorilor de pe chess.com în cazul meciurilor cu durată de maximum 15 minute. Putem trage concluzia că dacă agentul ar fi o persoană, acesta teoretic s-ar clasa în primii 1% jucători globali și ar avea un titlu apropiat de Expert, Candidate Master sau National Master. [2]

Următoarele tabele ilustrează statistica performanței agentului dupa mai multe meciuri jucate împotriva unor agenți găzduiți de Chess.com [3] și respectiv Lichess.org [4].

| Rating Oponent | Victorii | Înfrângeri | Remize | Total | Rată Victorie |
|---|---|---|---|---|---|
| 1600 (Chess.com) | 7 | 1 | 2 | 10 | 70% |
| 1800 (Chess.com) | 7 | 2 | 1 | 10 | 70% |
| 2000 (Chess.com) | 3 | 1 | 1 | 5 | 60% |
| 2200 (Chess.com) | 2 | 3 | 0 | 5 | 40% |
| 2300 (Chess.com) | 0 | 2 | 1 | 3 | 0% |

[1] Repository-ul de GitHub al proiectului, accesibil la https://github.com/Razvan48/Proiect-Licenta-FMI-UNIBUC

[2] Sistemul de Rating în Competițiile de Șah, detalii accesibile la https://en.wikipedia.org/wiki/Chess_rating_system

| Oponent | Victorii | Înfrângeri | Remize | Total | Rată Victorie |
|---|---|---|---|---|---|
| Stockfish nivel 5/8 (Lichess.org) | 5 | 1 | 1 | 7 | 71.4% |
| Stockfish nivel 6/8 (Lichess.org) | 0 | 4 | 1 | 5 | 0% |

Un dezavantaj pe care îl are implementarea curentă este dat de faptul că aflarea mutării optime durează câteva secunde, ceea ce nu permite ca agentul să fie folosit în cadrul meciurilor cu o limită mai scurtă de timp, cum ar fi Bullet și Blitz, care durează 3, respectiv 5 minute. Aplicația dă rezultate bune în cazul meciului de tip Rapid, unde fiecare jucător are 15 minute pentru toate mutările sale.

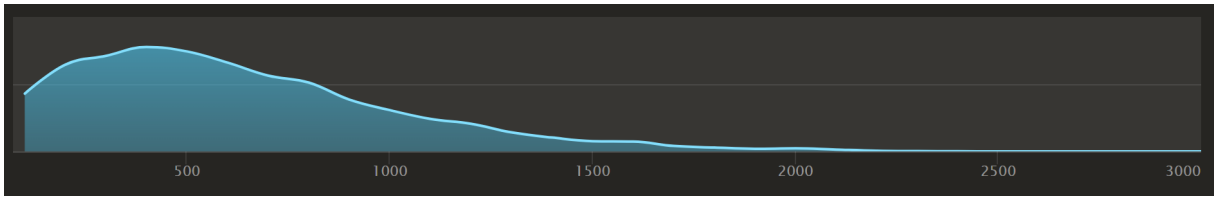



Figura 3.1: Distribuția Rating-urilor pentru meciuri cu limită de 15 minute (chess.com) [3]

Demo-urile în care se observă performanța agentului pot fi vizualizate aici [25, 26, 27, 28].

## 3.2 Studiu asupra Avantajului de un Tempo

În domeniul șahului există o problemă deschisă de decizie, care își pune întrebarea dacă avantajul de o mutare pe care jucătorul alb îl are este suficient pentru a forța o victorie în fața unui adversar ce joacă perfect, plecându-se de la o configurație inițială și egală material pentru ambii participanți. [3] Această problemă este calculabilă, dar puterea necesară este enormă, chiar dacă tabla jocului este de doar 8x8 celule.

Aplicația dezvoltată în această lucrare este suficient de eficientă pentru a răspunde la întrebări de această formă pentru table de șah de dimensiuni mai reduse, precum 4x4 sau 6x2 celule.

Astfel, în urma unor modificări efectuate asupra logicii aplicației, agentului i se filtrează mutările generate în mod obișnuit astfel încât să rămână în zona constrânsă a tablei, configurația de start fiind modificată pentru a include doar piese aflate în această zonă. De asemenea, toată logica ce determină mutarea optimă pentru un jucător poate fi adaptată pentru a determina un scor atribuit configurației curente în felul următor: dacă se consideră că albul are un scenariu în care câștigă indiferent de cât de bine joacă adversarul,

[3] Avantajul Albului într-un Meci de Șah, detalii accesibile la https://en.wikipedia.org/wiki/First-move_advantage_in_chess

atunci scorul asociat este $+\infty$. Dacă negrul câștigă indiferent de mutările albului atunci scorul asociat este $-\infty$. Pentru configurațiile unde este o remiză prin repetiție scorul va fi exact *0*, rămânând ca pentru orice altă stare valoarea asociată să fie $v \in (-\infty, +\infty) \setminus \{0\}$.

Scopul acum rămâne să mărim treptat adâncimea maximă și numărul de noduri vizitate în cadrul căutării exhaustive Minimax până când valoarea asociată stării de început nu mai este o valoare din $(-\infty, +\infty) \setminus \{0\}$, ci devine $-\infty$, 0 sau $+\infty$. În acest moment știm că starea curentă este o înfrângere absolută, o remiză, respectiv o victorie absolută, indiferent de mutările celor doi jucători. În acest moment a fost parcurs întregul arbore Minimax, iar mărirea în continuare a căutării realizate nu va mai avea niciun efect. Astfel de căutări complete pot dura considerabil, rularea fiind de câteva minute doar pentru table de dimensiuni reduse, precum 6x2.

Explicit în cadrul acestei căutări ce atribuie o valoare stării se poate aplica o optimizare adițională, considerând că o stare este remiză prin repetiție dacă aceasta a fost vizitată cel puțin încă o dată în drumul de la nodul curent către rădăcina arborelui Minimax. Acest lucru este o abatere de la regula șahului, care stipulează că acea configurație trebuie să fi fost vizitată anterior de cel puțin două ori pentru a fi remiză prin repetiție, dar în cazul calculului valorii asociate stării acest lucru nu modifică rezultatul final și aduce în schimb o îmbunătățire a timpului de rulare, dimensiunea arborelui reducându-se.

În cazul unei table de 4x4 apare un scenariu interesant. Piesele celor doi adversari sunt atât de apropiate încât avantajul de un tempo al albului este transformat într-un lanț de șahuri repetate împotriva jucătorului negru, acesta fiind nevoit să își sacrifice piesele pentru a salva regele. Victoria aparține cert albului.

În cazul unei table de 6x2 nu mai poate fi exploatată apropierea pieselor, rezultatul fiind acela că are loc o remiză în cazul unui meci jucat perfect de către ambele tabere.

## 3.3 Automatizarea Evaluării

În meciurile jucate dintre aplicație și site-uri precum chess.com și lichess.org este nevoie de o persoană care să funcționeze drept intermediar între cei doi agenți și care să transmită mutările realizate de la un bot la celalalt. Pentru a scăpa de această ineficiență s-a implementat un program care automatizează interacțiunea dintre aplicație și site-uri.

Acest script a fost dezvoltat în Python 3.9.13 și folosește următoarele module:

- NumPy, versiunea 2.0.2
- PyAutoGUI, versiunea 0.9.54
- Pillow (PIL), versiunea 11.1.0, incluzând submodulele Image și ImageDraw

Comunicarea inter-proces dintre aplicație și script a fost realizată prin socket-uri.

Programul în Python realizează periodic capturi de ecran folosind modulul Pillow, după care identifică un minimal bounding box pentru tabla de șah de pe site. Tabla este apoi împărțită în cele 64 de pătrate ale sale, fiecare celulă fiind apoi comparată cu echivalenta ei din captura de ecran anterioară. Se calculează o histogramă a valorilor RGB din imagini și se identifică ce pătrate s-au schimbat, afându-se astfel mutarea realizată.

Această mutare este apoi trimisă prin intermediului modulului socket către aplicația de șah, ce o va aplica pe propria tablă, urmând să-și calculeze în mod obișnuit răspunsul. Mutarea generată, împreună cu noua configurație a tablei de șah, sunt apoi trimise către script. Acesta va folosi modulul PyAutoGUI pentru a manipula mouse-ul și a aplica mutarea primită pe tabla de șah de pe site.

Din acest moment procedeul se reia, programul scris în Python continuând să realizeze capturi de ecran periodice pentru a identifica următoarea mutare primită de la site.

Un demo pentru automatizarea jocului dintre aplicație și site poate fi accesat aici [29].

# Capitolul 4

# Concluzii

Lucrarea a avut ca prim scop să evidențieze cum înțelegerea aprofundată a unei probleme ce trebuie rezolvată, luarea unor decizii arhitecturale balansate și menținerea unui echilibru între memoria utilizată și timpul de execuție pot aduce rezultate notabile.

Rezultatul întregului proiect a fost obținerea unei aplicații integrale și complet funcționale care permite utilizatorului să joace șah, atât împotriva altor jucători prin intermediul modului multiplayer și a arhitecturii Client-Server implementate, cât și contra unui agent în modul singleplayer, acesta folosind diverse tehnici de optimizare la nivel de bits și structuri de date eficiente. Proiectul îmbină astfel subdomenii variate ale Informaticii, de la grafică și principii ale programării orientate pe obiecte, până la comunicare inter-proces.

Aplicația este în stare să învingă oponenți competitivi, respectând în același timp limitările hardware la care este supusă. Agentul a câștigat 2 meciuri din cele 5 jucate împotriva unui adversar de rating 2200 și 5 meciuri din 7 contra Stockfish nivel 5/8 găzduit de Lichess.org [4]. Aceste rezultate clasează aplicația la nivelul campionilor locali și naționali din cadrul competițiilor de șah.

## 4.1 Posibile Îmbunătățiri

Asupra proiectului pot fi aplicate diverse îmbunătățiri.

De exemplu, o schimbare notabilă asupra agentului ar fi înlocuirea evaluării statice care are loc în fiecare nod frunză al arborelui Minimax cu o inferență la un model de învățare automată, cum ar fi o rețea neuronală convoluțională.

Acest lucru ar putea îmbunătății modelul, deoarece funcția euristică a unei stări ar fi învățată.

În schimb, o problemă care ar apărea este modul cum algoritmul ar apela inferența modelului, luând în considerare faptul că arborele Minimax este calculat pe mai multe fire de execuție. Utilizarea unei variabile de tip mutex poate îngreuna considerabil algoritmul, așa cum s-a concluzionat în timpul implementării sistemului de memoizare pentru stările

deja întâlnite. O metodă ar fi duplicarea modelului pentru ca fiecare fir de execuție să poată realiza o inferență fără să existe probleme de concurentă.

Mai apare problema obținerii unui set de date pentru antrenare. O posibilă soluție ar fi construirea acestuia în timpul parcurgerilor Minimax ale aplicației, într-o manieră similară cu algoritmii de Reinforcement Learning.

Pentru construirea și antrenarea modelului s-a constatat faptul că nu există multe biblioteci în `C++` care să faciliteze acest lucru. Astfel, o abordare ar fi construirea arhitecturii modelelor de la zero, incluzând metoda de feed-forward și cea de back-propagation a gradientului. Altă abordare ar fi utilizarea modulelor existente în Python, cum ar fi TensorFlow sau PyTorch. Problema în această situație este implicarea unei comunicări inter-proces, algoritmul Minimax fiind nevoit să apeleze alt program pentru a afla rezultatul unei inferențe, ceea ce ar îngreuna enorm generarea arborelui, unde viteza este esențială.

O altă îmbunătățire ar fi posibila integrare a unui mecanism prin care algoritmul Minimax să se oprească dacă rularea lui depășește un anumit timp prestabilit. Momentan, aplicația își generează mutarea în câteva secunde, ceea ce este acceptabil în cazul unui meci ce durează de exemplu 15 minute, cum ar fi cele de tip Rapid. În schimb, pentru cele de tip Bullet și Blitz, care durează 3, respectiv 5 minute, aplicația nu mai poate fi utilizată în versiunea curentă.

# Bibliografie


[1] *Engine de Șah Stockfish*, Ultima accesare: iunie 2025, URL: https://stockfishchess.org/.

[2] *Engine de Șah Leela Chess Zero*, Ultima accesare: iunie 2025, URL: https://lczero.org/.

[3] *Chess.com*, Ultima accesare: iunie 2025, URL: https://www.chess.com/.

[4] *Lichess.org*, Ultima accesare: iunie 2025, URL: https://lichess.org/.

[5] *Microsoft Visual Studio 2022*, Ultima accesare: iunie 2025, URL: https://visualstudio.microsoft.com/downloads/.

[6] *Bibliotecă GLFW*, Ultima accesare: iunie 2025, URL: https://www.glfw.org/download.html.

[7] *Bibliotecă GLEW*, Ultima accesare: iunie 2025, URL: https://glew.sourceforge.net.

[8] Sean Barrett, *Header stb_image.h*, Ultima accesare: iunie 2025, URL: https://github.com/nothings/stb/blob/master/stb_image.h.

[9] *Bibliotecă GLM*, Ultima accesare: iunie 2025, URL: https://github.com/g-truc/glm.

[10] *Bibliotecă FreeType*, Ultima accesare: iunie 2025, URL: https://freetype.org/download.html.

[11] *Bibliotecă ENet*, Ultima accesare: iunie 2025, URL: http://enet.bespin.org/Downloads.html.

[12] *Bibliotecă IrrKlang*, Ultima accesare: octombrie 2024, URL: https://www.ambiera.com/irrklang/downloads.html.

[13] *Aplicație LogMeIn Hamachi*, Ultima accesare: iunie 2025, URL: https://vpn.net/.

[14] Colin M.L. Burnett, *Texturile Pieselor de Șah*, Licență: CC BY-SA 3.0. Ultima accesare: iunie 2025, URL: https://commons.wikimedia.org/wiki/Category:PNG_chess_pieces/Standard_transparent.

[15] *Efecte Sonore Folosite*, Licență: Pixabay Content. Ultima accesare: iunie 2025, URL: https://pixabay.com/sound-effects/.

[16] *Fontul de text Open Sans Regular*, Ultima accesare: iunie 2025, URL: https://fonts.google.com/specimen/Open+Sans.

[17] *Repository Proiect (GitHub)*, Ultima accesare: iunie 2025, URL: https://github.com/Razvan48/Proiect-Licenta-FMI-UNIBUC.

[18] *Demo Multiplayer (YouTube)*, Ultima accesare: iunie 2025, URL: https://www.youtube.com/watch?v=b2uM6Mkn8E0.

[19] Rashi Sahay, *Comparative analysis of minmax algorithm with alpha-beta pruning optimization for chess engine*, februarie 2023, Ultima accesare: iunie 2025, URL: https://ijaem.net/issue_dcp/Comparative%20analysis%20of%20minmax%20algorithm%20with%20alpha%20beta%20pruning%20optimization%20for%20chess%20engine.pdf.

[20] April Walker, *Anatomia unui Bot de Șah*, Ultima accesare: iunie 2025, URL: https://medium.com/@SereneBiologist/the-anatomy-of-a-chess-ai-2087d0d565.

[21] *Zobrist Hashing*, Ultima accesare: iunie 2025, URL: https://www.chessprogramming.org/Zobrist_Hashing.

[22] Albert L. Zobrist, *A New Hashing Method with Application for Game Playing*, aprilie 1970, Ultima accesare: iunie 2025, URL: https://research.cs.wisc.edu/techreports/1970/TR88.pdf.

[23] Dennis Breuker și Jos W. H. M. Uiterwijk, *Transposition Tables in Computer Chess*, februarie 1970, Ultima accesare: iunie 2025, URL: https://www.researchgate.net/publication/2686280_Transposition_Tables_in_Computer_Chess.

[24] Rhys Rustad-Elliott, *Generarea Eficientă a Mutărilor*, Ultima accesare: iunie 2025, URL: https://rhysre.net/fast-chess-move-generation-with-magic-bitboards.html.

[25] *Demo Singleplayer (win vs. Bot, 1600 rating) (YouTube)*, Ultima accesare: iunie 2025, URL: https://www.youtube.com/watch?v=Fa7U4N-Ah20.

[26] *Demo Singleplayer (win vs. Bot, 1800 rating) (YouTube)*, Ultima accesare: iunie 2025, URL: https://www.youtube.com/watch?v=6eb28QZ9re0.

[27] *Demo Singleplayer (win vs. Bot, 2000 rating) (YouTube)*, Ultima accesare: iunie 2025, URL: https://www.youtube.com/watch?v=ktVuOuYLKAk.

[28] *Demo Singleplayer (win vs. Bot, 2200 rating) (YouTube)*, Ultima accesare: iunie 2025, URL: https://www.youtube.com/watch?v=UbMmwjcPJKc.

[29] *Demo Automatizare Gameplay (YouTube)*, Ultima accesare: iunie 2025, URL: https://www.youtube.com/watch?v=4EzIWjo99Eg.

# Anexa A

# Diagrame ce ilustrează comunicarea Client-Server

În figura A.1 se poate observa ordinea în care au loc transmisiile de informație. Ordinea diferă în funcție de clientul implicat. Instanța de server folosită este mereu cea care aparține utilizatorului creator al sesiunii de joc.

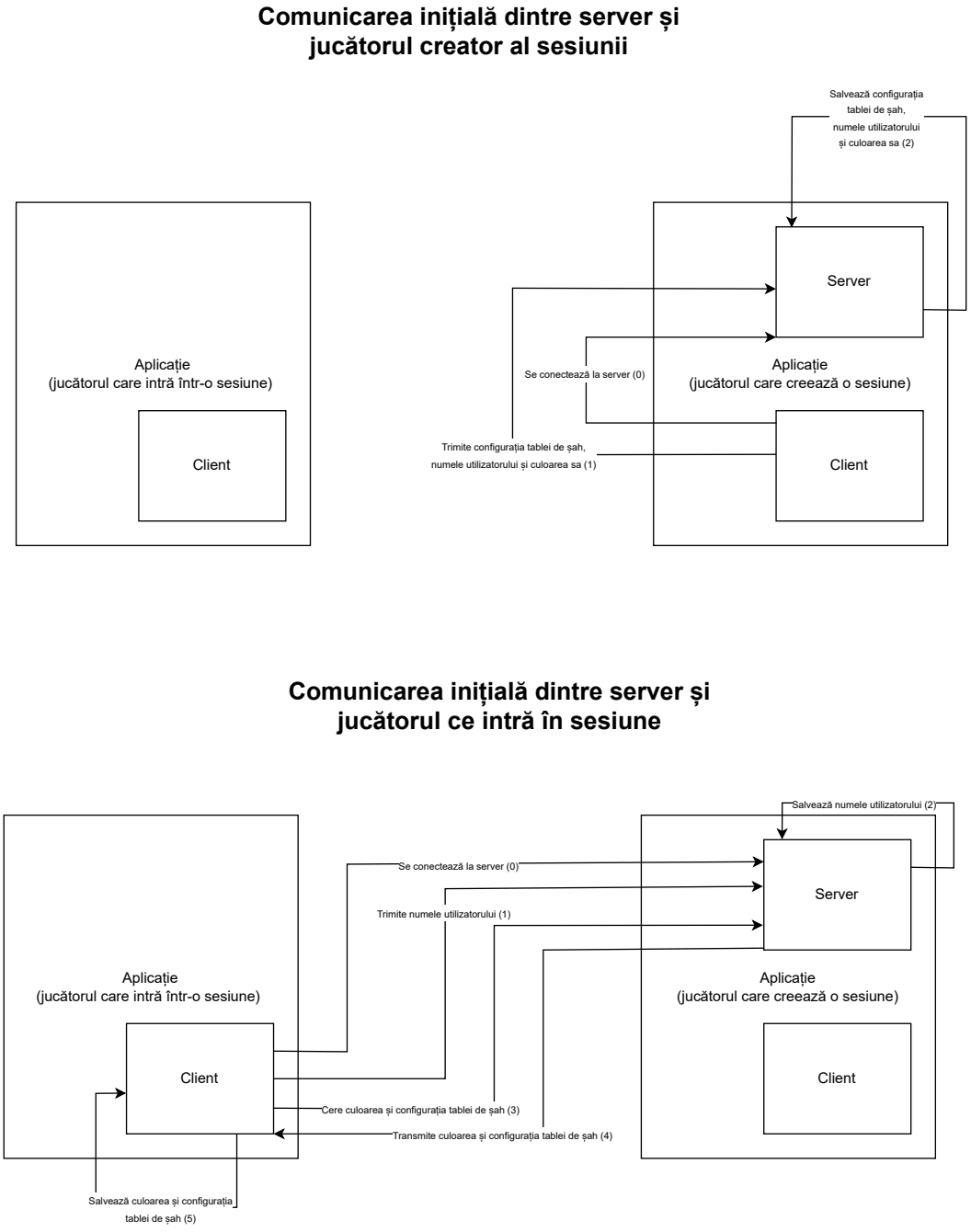
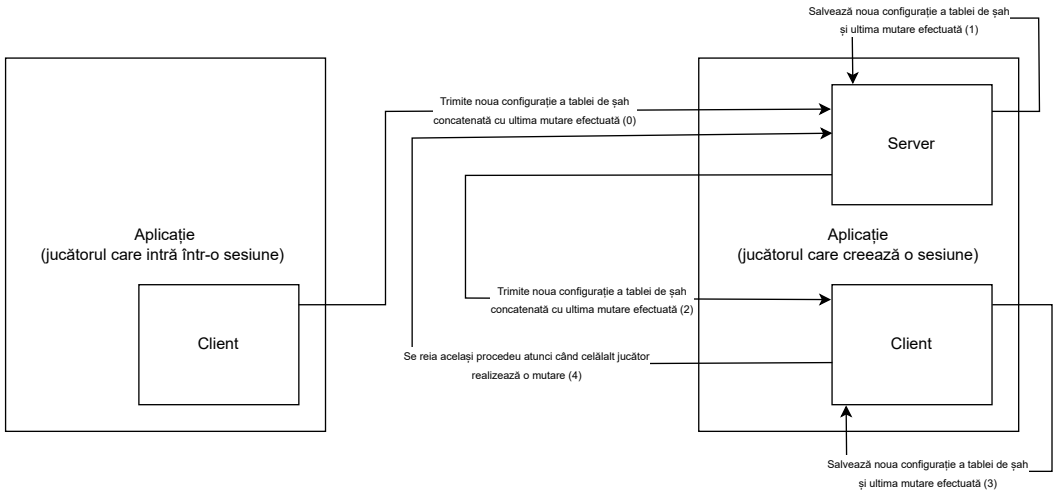



Figura A.1: Comunicare Inițială Client-Server

**Comunicarea dintre server și clienți**
**în timpul jocului**




**Sistemul de pinging**
**dintre server și clienți**

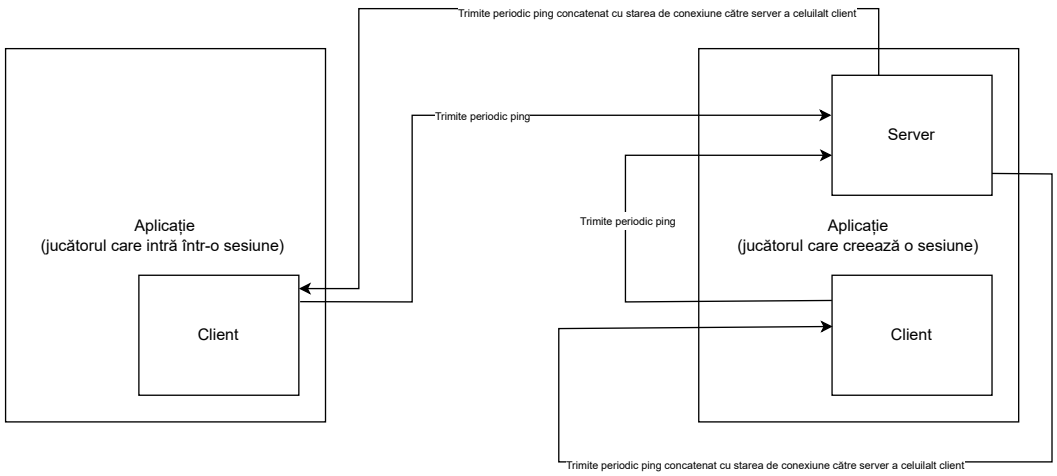



Figura A.2: Comunicare Client-Server în joc

# Anexa B

# Vizualizări ale tehnicilor de optimizare folosite

În figura B.1 sunt ilustrate câte două exemple de zone de atac pentru rege și pentru cal. Albastrul semnifică poziția curentă a piesei pe tablă, iar roșu este zona de atac. Această tabla de bits este stocată într-o variabilă de tip unsigned long long.

**Exemple Zone de Atac pentru Rege**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Exemple Zone de Atac pentru Cal**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Figura B.1: Exemple Zone de Atac, idee inspirată din [24]

# Anexa C

# Exemplificări ale performanței agentului obținut

În figurile C.1, C.2, C.3, C.4 sunt ilustrate ultimele configurații ale tablei de șah înainte ca agentul implementat să forțeze matul, alături de mutarea câștigătoare.

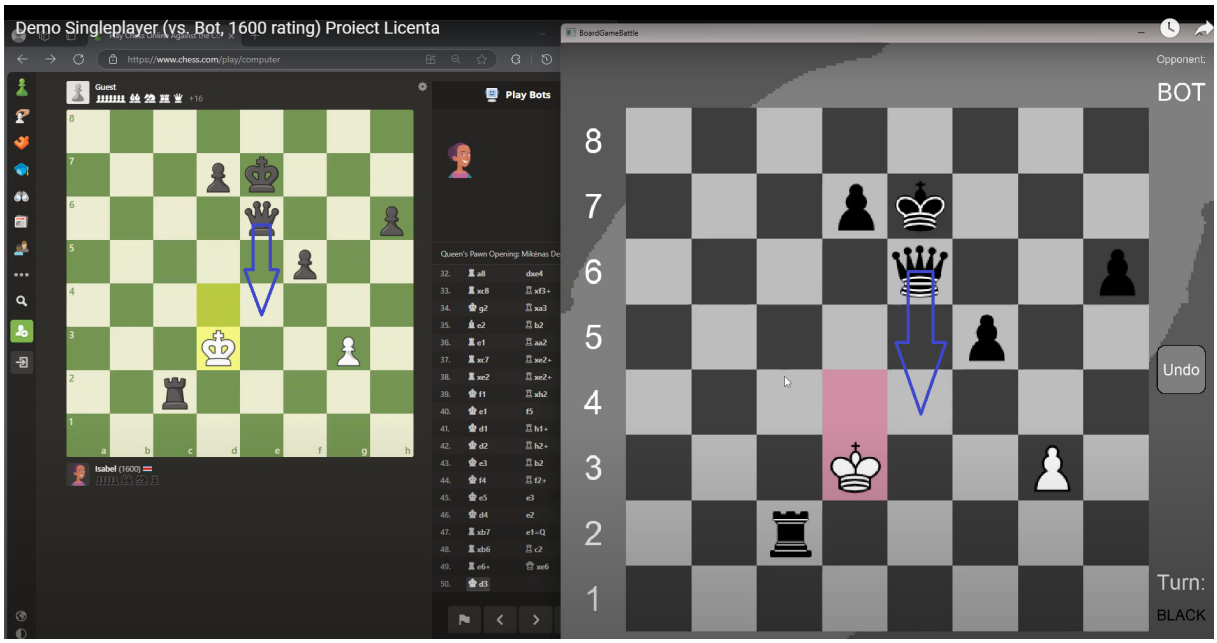



Figura C.1: Victorie împotriva bot-ului Isabel(1600) (chess.com) [3], imagine preluată din demo-ul accesibil la [25]

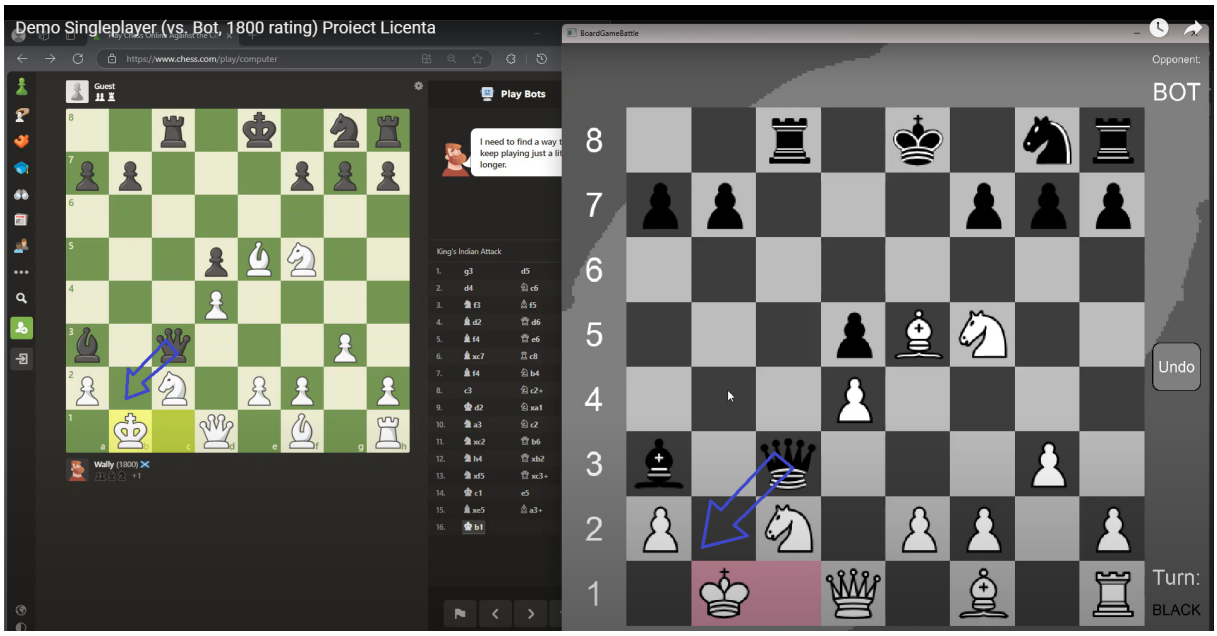



Figura C.2: Victorie împotriva bot-ului Wally(1800) (chess.com) [3], imagine preluată din demo-ul accesibil la [26]

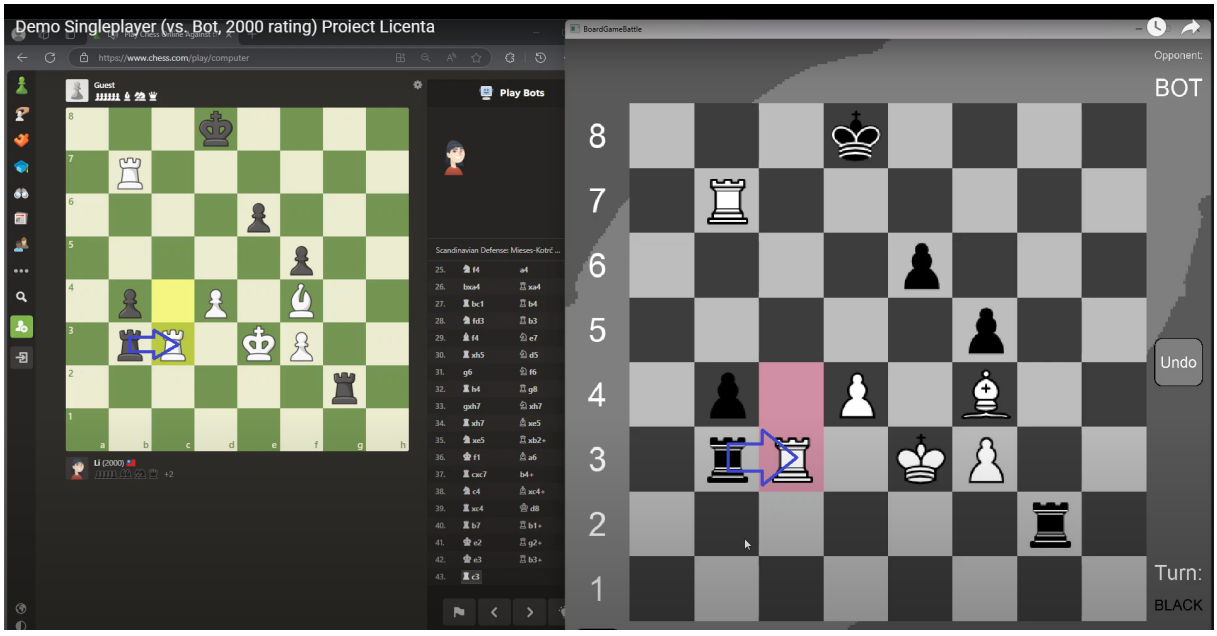



Figura C.3: Victorie împotriva bot-ului Li(2000) (chess.com) [3], imagine preluată din demo-ul accesibil la [27]

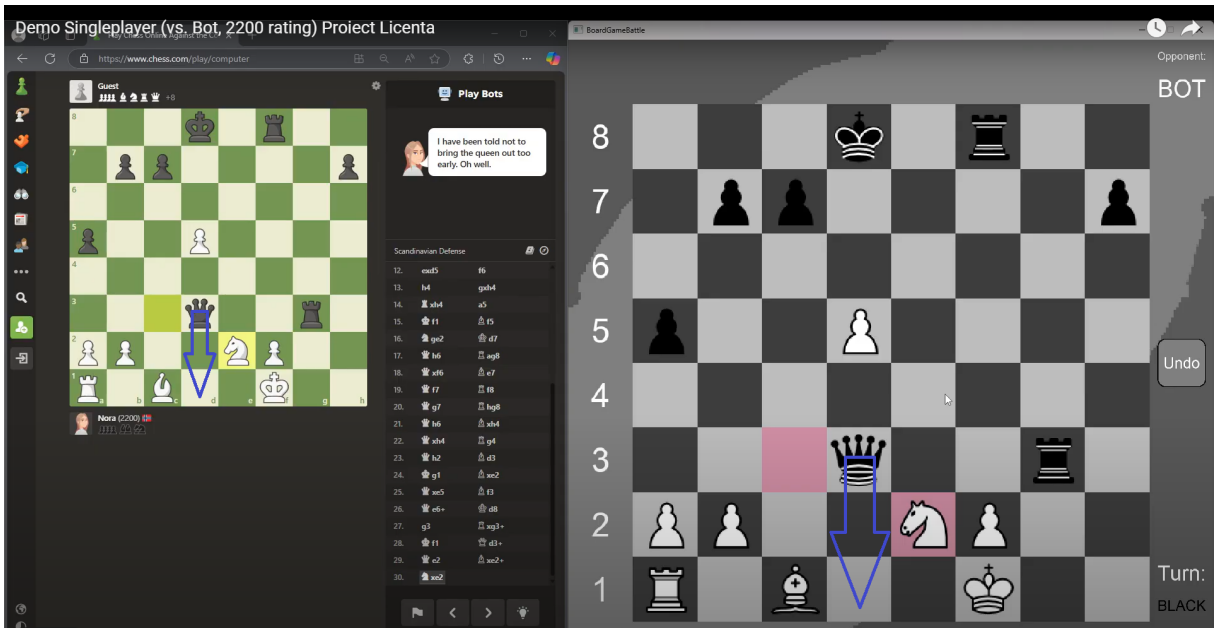



Figura C.4: Victorie împotriva bot-ului Nora(2200) (chess.com) [3], imagine preluată din demo-ul accesibil la [28]

# Anexa D

# Vizualizări ale studiului realizat asupra avantajului de un tempo

În figurile D.1 și D.2 se observă 2 configurații de start diferite și rezultatul unui joc pornind din acele stări.

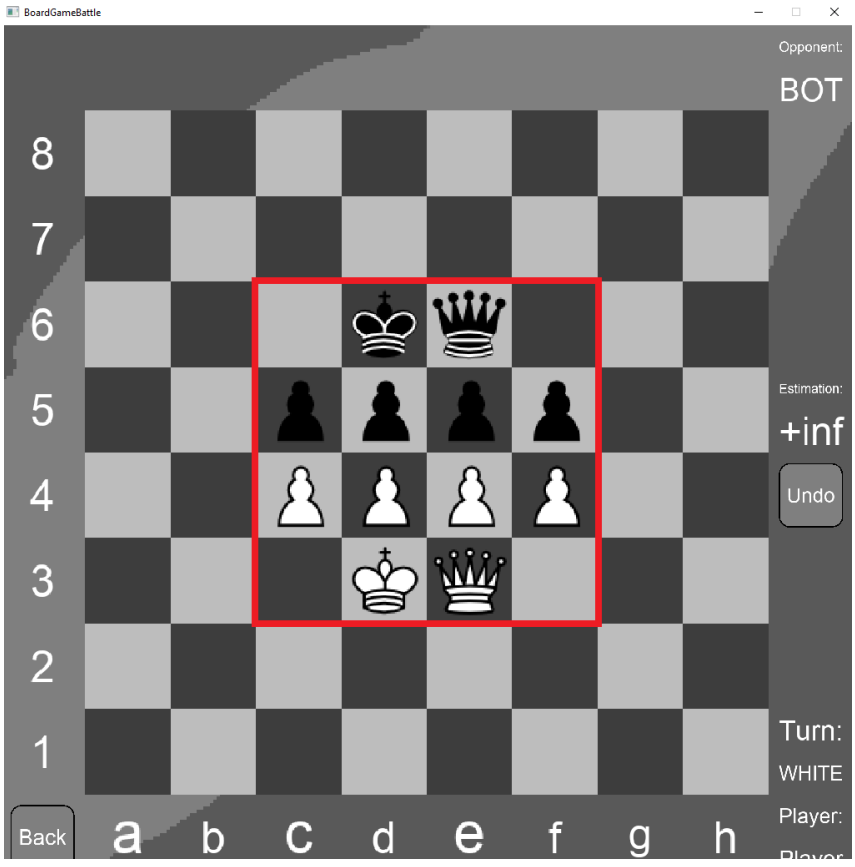



Figura D.1: Victorie pentru alb pe o tablă de 4x4, chenarul roșu reprezentând zona restrânsă a jocului. Dimensiunea arborelui parcurs a fost de 20,168,282 noduri.

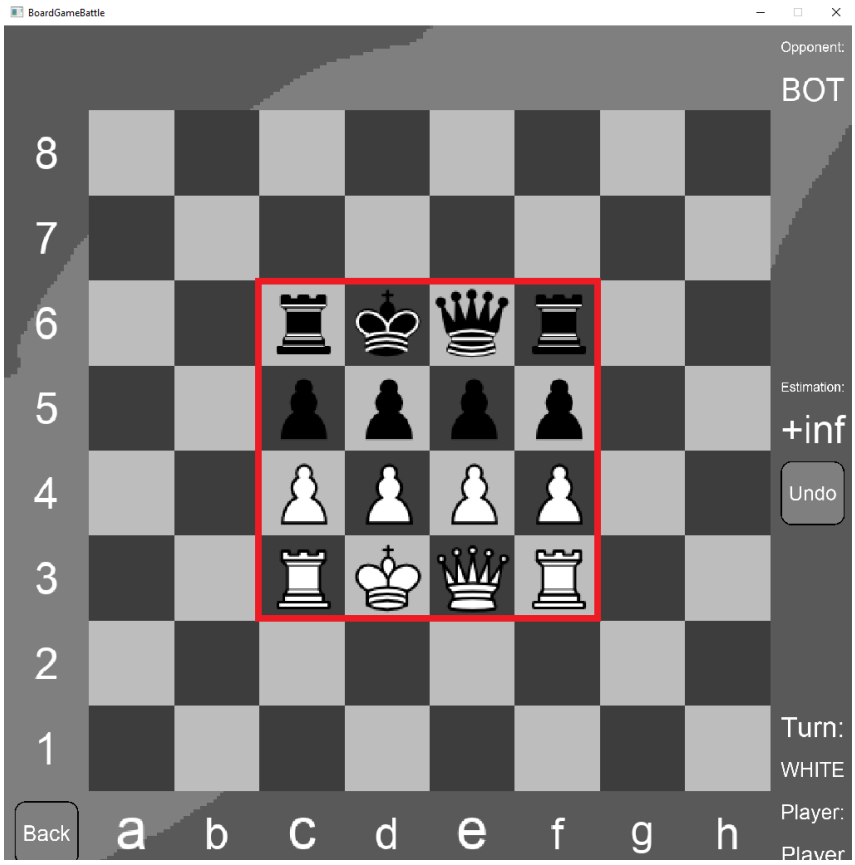



Figura D.2: Victorie pentru alb pe o tablă de 4x4, chenarul roșu reprezentând zona restrânsă a jocului. Dimensiunea arborelui parcurs a fost de 29,566,406 noduri.

În figura D.3 se observă remiza ce are loc în cazul unui meci jucat perfect de către ambele tabere pe o tabla de dimensiune 6x2.

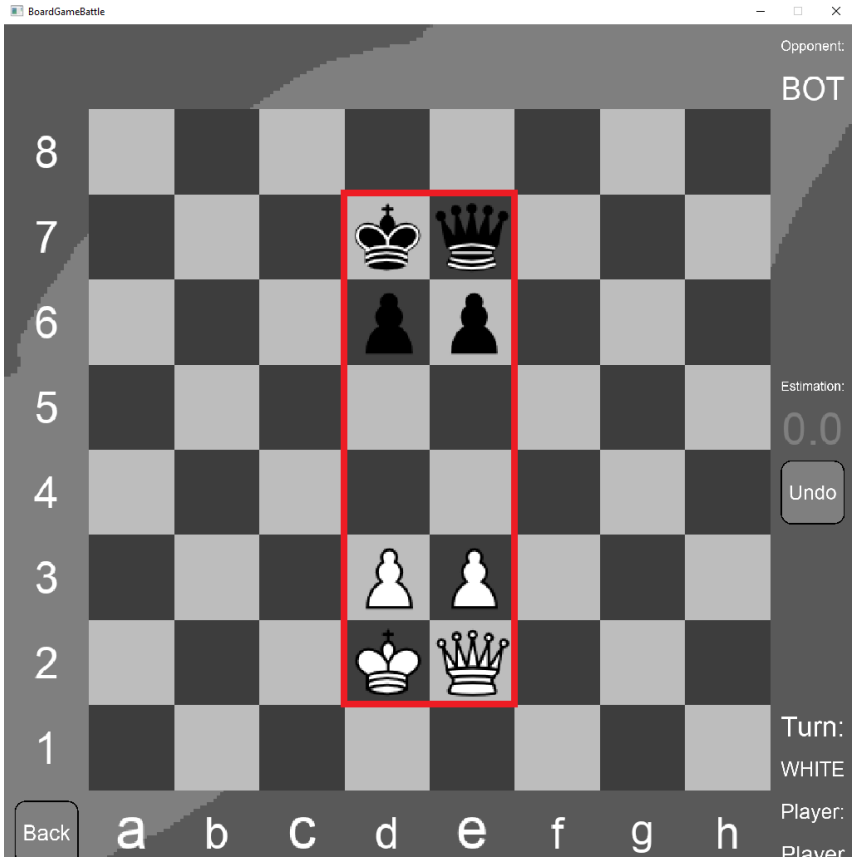



Figura D.3: Remiză totală pe o tablă de 6x2, chenarul roșu reprezentând zona restrânsă a jocului. Dimensiunea arborelui parcurs a fost de 18,777,324 noduri.

# Anexa E

# Diagramă ce descrie comunicarea dintre script-ul de automatizare și aplicație

În figura E.1 se poate observa ordinea în care au loc interacțiunile dintre site, script-ul de automatizare al evaluării și aplicația dezvoltată. Script-ul a preluat funcția de intermediar.

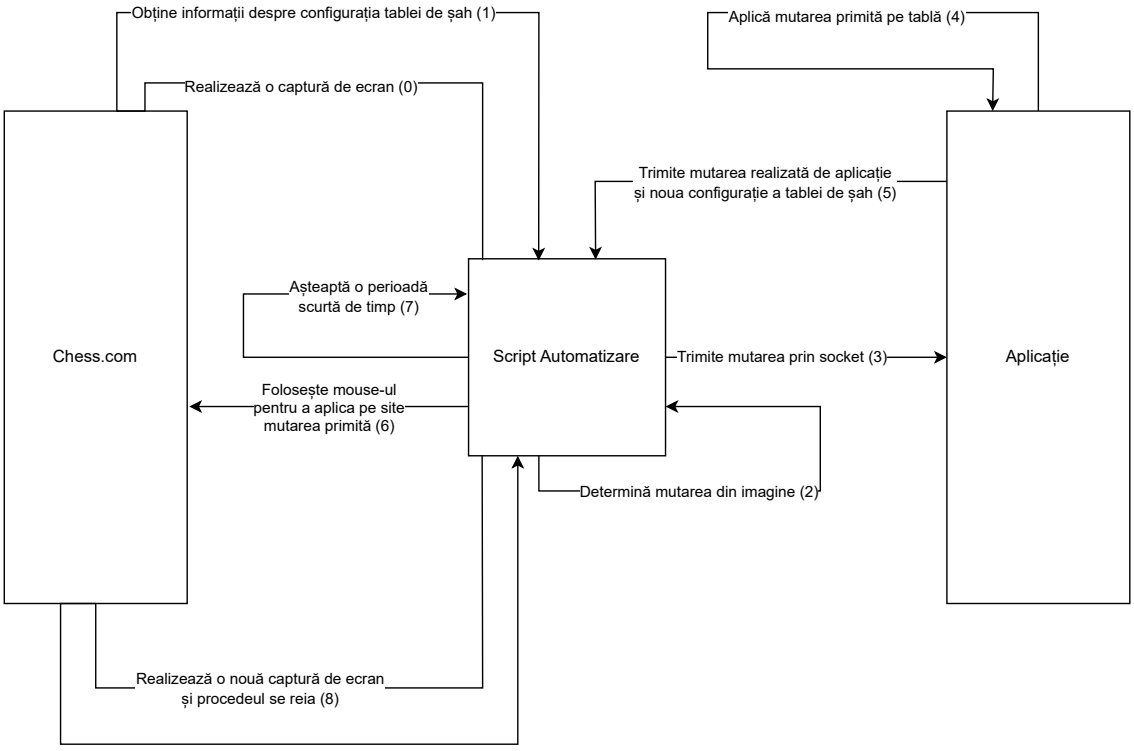



Figura E.1: Comunicare Script Automatizare