Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

Kierunek: **Informatyka algorytmiczna (INA)**

PRACA DYPLOMOWA INŻYNIERSKA

Bot dla gry w Szachy

Bot for Chess game

Krzysztof Wiśniewski

Opiekun pracy
dr Maciej Gębala, prof. uczelni

Słowa kluczowe: szachy, bot, teoria gier

WROCŁAW, 2025

Streszczenie

Celem pracy było opracowanie silnika szachowego w języku Java. Program ten zaprojektowano tak, aby analizować i oceniać pozycję na szachownicy, a następnie sugerować graczowi najlepszy ruch, uwzględniając jego strategiczne i taktyczne aspekty. Interakcję z programem zaprojektowano dla wiersza poleceń, z wykorzystaniem Uniwersalnego Interfejsu Szachowego. Umożliwiło to łatwą integrację z innymi aplikacjami szachowymi oraz pozwoliło na przeprowadzanie symulacji i analiz działania silnika bez potrzeby aplikowania interfejsu graficznego.

Niniejsza praca inżynierska składa się z trzech części, w których omówiono kolejne etapy pracy nad opracowaniem silnika szachowego. W pierwszej części przedstawiono podstawową wersję programu, która obejmuje generowanie możliwych ruchów zgodnie z zasadami gry w szachy, algorytm wyszukiwania optymalnego ruchu oraz implementację naiwnej heurystyki. W części drugiej opisano ulepszenia algorytmów wyszukiwania i oceny pozycji, mające na celu zwiększenie efektywności i precyzji silnika. Ostatnią część pracy poświęcono zagadnieniom związanym z testowaniem siły programu. Przeprowadzono analizę wydajności w odniesieniu do różnych jego wersji oraz innych silników, uwzględniając testy porównawcze oraz metodologię oceny skuteczności.

Efektem prac jest silnik, którego ranking szachowy można szacować na zakres pomiędzy 1800 a 2000 ELO, co plasuje go na poziomie porównywalnym z graczem posiadającym kategorię kandydata na mistrza krajowego.

Słowa kluczowe: szachy, bot, teoria gier

Abstract

The aim of this thesis is to develop a chess engine in Java. This program is designed to analyze and evaluate position on the chessboard and subsequently suggest the best move for the player, considering its both strategic and tactical aspects. Interaction with the program is conducted via the command line, using the Universal Chess Interface. This allows for easy integration with other chess applications and facilitates the simulation and analysis of the engine's performance without the need to create graphical interface.

This engineering thesis consists of three parts, which discuss the successive stages of developing the chess engine. The first part presents the basic version of the program, which includes generating possible moves according to the rules of chess, the algorithm for searching for the optimal move, and the implementation of a naive heuristic. The second part describes the improvements to the search and position evaluation algorithms, aiming to increase the efficiency and precision of the engine. The final part of the thesis is dedicated to issues related with testing the engine's strength. An analysis of performance was conducted with respect to various versions of the program and other engines, including comparative tests and methodology for assessing effectiveness.

The result of the work is an engine, whose chess ranking can be estimated in the range between 1800 and 2000 ELO, which places it at a level comparable to a player with the title of Candidate for National Master.

Keywords: chess, bot, game theory

Spis treści

Spis rysunków

Spis tabel

Spis listingów

Skróty

UCI (ang. *Universal Chess Interface*) Uniwersalny Interfejs Szachowy
FEN (ang. *Forsyth–Edwards Notation*) Notacja Forsytha-Edwardsa
LAN (ang. *Long Algebraic Notation*) Pełna Algebraiczna Notacja Szachowa
Perft (ang. *Performance Test*) Test Wydajności

Rozdział 1

Wstęp

1.1. Wprowadzenie

Szachy, powszechnie nazywane grą królewską, są jedną z najstarszych, a zarazem najpopularniejszych form intelektualnej rozrywki w dziejach ludzkości. Swoją niekwestionowaną reputację, tak wśród profesjonalistów, jak i amatorów, zawdzięczają połączeniu prostoty zasad ze złożoności strategicznych wyzwań. Ich historia, sięgająca VI wieku p.n.e., obejmuje stale ponawiane próby udoskonalania reguł i odkrywania nowych, coraz bardziej zaawansowanych, taktyk mających zagwarantować zwycięstwo nad oponentem. Wprowadzenie roszady, czy ruchu en-passant to najbardziej spektakularne przykłady zmian, świadczących o nieograniczonej kreatywności kolejnych pokoleń graczy.

Jednak największą zmianą w swojej ponad tysiąc pięćsetletniej historii, szachy zawdzięczają postępowi technologicznemu w połowie XX wieku. Rozwój zaawansowanych maszyn liczących otworzył możliwość zautomatyzowania procesu analizy partii szachowych na niespotykaną dotąd skalę.

Za pioniera w tej dziedzinie uważa się amerykańskiego matematyka Claude Shannon, który w roku 1950 opublikował pracę o teoretycznych aspektach programowania silników szachowych, opartych o ocenę heurystyczną oraz algorytm min-max. Istotny wkład w rozwój szachowej sztucznej inteligencji miał także ojciec informatyki — Alan Turing, który rok po publikacji pracy Shannona zaprojektował pierwszy program komputerowy, w pełni zdolny do rozegrania partii szachowej. Ograniczenia techniczne tamtych czasów nie pozwoliły jednak na przetestowanie programu na maszynie. Rozegrano niewielką liczbę partii szachowych, w których każdy ruch był obliczany analogowo.

Najstarszy program uruchomiony na komputerze, który pozwalał na przeprowadzenie pełnej rozgrywki, powstał w 1958 roku. Od tamtego momentu wiele silników szachowych biło rekordy swoich poprzedników. Do kluczowego przełomu doszło zimą 1997 roku, kiedy to silnik szachowy DeepBlue wygrał pojedynek $3\frac{1}{2} - 2\frac{1}{2}$ z ówczesnym mistrzem świata, Garrym Kasparovem.

Po tym wydarzeniu świat wkroczył w erę super silników. Szachy stały się nie tylko areną dla ludzkiego intelektu, ale także polem testowym zaawansowanych technologii. Współcześnie, wykorzystanie komputerów stanowi nieodłączny element analizy partii szachowych. Zastosowanie najnowocześniejszych rozwiązań, takich jak uczenie maszynowe i sieci neuronowe, zrewolucjonizowało sposób, w jaki rozumiemy tą gre, oraz pokazało, jak wiele jeszcze można w tej dziedzinie osiągnąć.

1.2. Cel i zakres

Zasadniczym celem pracy było stworzenie silnika szachowego, zdolnego do oceny heurystycznej pozycji, oraz proponowania graczowi ruchów z uwzględnieniem ich strategicznych aspektów.

Zakres pracy objął następujące zagadnienia:

- Przegląd literatury na temat technik oraz algorytmów wykorzystywanych przy tworzeniu nowoczesnych silników szachowych.
- Zapoznanie się z powszechnie obowiązującymi zasadami turniejowej gry w szachy, opublikowanymi przez Międzynarodową Federację Szachową.
- Stworzenie silnika szachowego w języku programowania Java 22, z celowym pominięciem dodatkowych rozwiązań open-source.
- Wykorzystanie Uniwersalnego Interfejsu Szachowego do komunikacji z systemem.
- Zintegrowanie silnika z wybranym interfejs graficzny.
- Testowanie poprawności stworzonego oprogramowania.
- Implementacja rozwiązań programistycznych przyspieszających przeszukiwanie drzewa decyzyjnego oraz ulepszających dokładność oceny heurystycznej.
- Przeprowadzenie analizy porównawczej pomiędzy wersjami systemu w celu oceny efektywności zastosowanych rozwiązań.
- Porównanie najlepszej wersji silnika z już istniejącymi rozwiązaniami w celu określenia poziomu gry.

1.3. Układ pracy

Niniejsza praca inżynierska składa się z trzech głównych części.

W pierwszej z nich opisano elementy oprogramowania konieczne do stworzenia podstawowej wersji silnika szachowego. Przedstawiono metody komunikacji z interfejsem, techniki reprezentacji pozycji i generowania dostępnych ruchów, algorytm minimalizowania start i maksymalizowania zysków wraz z oceną heurystyczną oraz metodologią zarządzania czasem gry. Przetestowano aplikację w celu sprawdzenia poprawności działania.

Następny rozdział poświęcono pracy nad ulepszeniem systemu. Przedstawiono zaimplementowane rozwiązania mające na celu poprawę poziomu gry systemu. Skupiono się na dwóch kierunkach: poprawy prędkości przeszukiwania stanów oraz na poprawie precyzji heurystycznej oceny pozycji. Jako że dla słabych silników o wiele większe znaczenie w poprawie jego gry ma pierwszy z tych aspektów, omówiono go w pierwszej kolejności. [5] Z uwagi na mnogość opisanych w literaturze możliwych rozwiązań danego problemu algorytmicznego, w niektórych miejscach przedstawiono także techniki, które ostatecznie nie zostały zaimplementowane w silniku, a także wytłumaczono, czym kierował się autor przy wyborze danego rozwiązania.

Ostatnią część pracy poświęcono testom wydajnościowym oraz jakościowym. Przetestowano program pomiędzy różnymi jego wersjami. W dalszej części przedstawiono wyniki gry systemu przeciwko innym, publicznie dostępnym silnikom. Na końcu podsumowano rozgrywki przeprowadzone z graczami.

W dodatku do pracy zamieszczono instrukcję wdrożenia aplikacji w celu odpalenia silnika we własnym środowisku. Choć autor zakłada, że czytelnik zna zasady gry w szachy, w pracy zawarto także omówienie niektórych, bardziej skomplikowanych bądź mniej znanych, aspektów tej gry. Wiele z opisanych w niniejszej pracy rozwiązań algorytmicznych można znaleźć jedynie w literaturze anglojęzycznej, a ich przetłumaczenie nie jest dokładne.

Rozdział 2

Implementacja silnika szachowego

2.1. Komunikacja z systemem

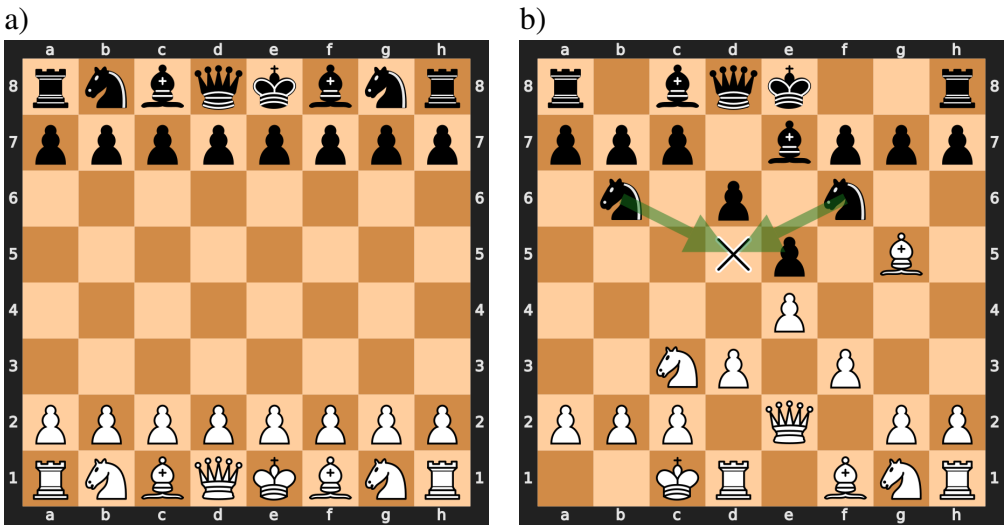
2.1.1. Notacja Forsytha-Edwardsa

W 1950 roku amerykański matematyk Claude Shanon na łamach ”Philosophical Magazine” opublikował pracę naukową zatytułowaną ”Programowanie komputera do gry w szachy”. [4] Stała się ona teoretyczną podstawą dla dalszego rozwoju silników szachowych. Zawarte w niej zostało między innymi oszacowanie, co do ilości możliwych pozycji szachowych, wynoszące 10^{43}. Oznacza to, że liczba legalnych ułożeń planszy przewyższa o rzędy wielkości liczbę gwiazd w widzialnym wszechświecie.

Aby umożliwić użytkownikowi efektywne wprowadzenie danych oraz komunikację z programem, należało w pierwszej kolejności sprecyzować format, w jakim zostaną dostarczone informacje dotyczące aktualnej pozycji. Standardem, wykorzystywanym nie tylko w większości silników, ale także w pojedynkach rozgrywanych online, jest Notacja Forsytha-Edwardsa (ang. *Forsyth–Edwards Notation*, w skrócie FEN). Została stworzona przez dziennikarza Davida Forsytha, a następnie dalej dostosowywana do potrzeb komputerów przez Stevena Edwardsa.

Notacja FEN jest linią znaków ASCII, która pozwala na precyzyjne określenie aktualnego stanu gry. Wielkimi literami kodowane są bierki białe, małymi natomiast bierki czarne. Każda z nich opisana jest skrótem pochodzącym od ich angielskich nazw.

Pełna specyfikacja FEN dostępna jest w dokumentacji ”Portable Game Notation”. [1]



Rys. 2.1: Przykładowe pozycje szachowe: a) startowa, b) po paru ruchach

2.1 a) `rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1`
2.1 b) `r1bqk2r/ppp1bppp/1n1p1n2/4p1B1/4P3/2NP1P2/PPP1Q1PP/2KR1BNR b kq - 7 7`

2.1.2. Szachowa Notacja Algebraiczna

Kluczowym aspektem, z perspektywy komunikacji z systemem, jest także określenie formatu zapisu ruchów. W aplikacji wykorzystano Szachową Notację Algebraiczną.

Notacja ta, w swojej krótkiej formie, jest powszechnie stosowana w literaturze oraz podczas oficjalnych zawodów. Stanowi jedyną formę zapisu posunięć uznawaną przez Międzynarodową Federację Szachową. Zawiera informacje o rodzaju ruszanej bierki oraz o jej polu docelowym. Zapis ten z punktu widzenia komputerów zawiera jednak wadę. W sytuacji, w której dwie bierki tego samego rodzaju mogą poruszyć się na jedno pole, występuje dwuznaczność zapisu 2.1. Choć w takiej sytuacji dodaje się do ruchu kolumnę bądź wiersz startowy różniący obie bierki, jest to rozwiązanie wymagające implementacji dodatkowej logiki oraz wiedzy o stanie całej planszy.

Znacznie bardziej intuicyjne dla komputerów jest zastosowanie długiej wersji szachowej notacji algebraicznej. Zawarte są w niej informacje o polu startowym oraz polu docelowym ruchu, usuwając tym samym ryzyko dwuznaczności. Roszady oznaczano przez pola ruchu króla, natomiast do ruchów z promocją dopisano literę określającą rodzaj podmienionej figury.

2.1.3. Uniwersalny Interfejs Szachowy

Uniwersalny Interfejs Szachowy (ang. *Universal Chess Interface*, w skrócie UCI) [2] jest ustandaryzowanym protokołem tekstowym, służącym do wymiany informacji pomiędzy różnymi programami szachowymi. Jego implementacja pozwoliła na komunikację silnika szachowego z wybranymi interfejsami graficznymi oraz środowiskami testowymi.

UCI jest protokołem rozbudowanym, pozwalającym między innymi na rozgrywki innych wersji szachów niż europejskie, dla przykładu Chess960. W silniku zaimplementowano jedynie te z komend, które konieczne były do rozegrania podstawowej partii mierzonej czasowo.

Metodę połączenia z dowolnym programem obsługującym UCI przedstawiono w dodatku A. Opisy komend oraz przykład wymiany informacji pomiędzy aplikacją a GUI zaprezentowano w dodatku B.

2.2. Reprezentacja pozycji

2.2.1. Reprezentacja szachownicy

Istotnym aspektem, z punktu widzenia prędkości działania silnika, było zastosowanie odpowiedniego typu reprezentacji położenia bierek na 64-polowej planszy szachowej. Po zapoznaniu się z proponowanymi w literaturze rozwiązaniami, w projekcie zdecydowano się zaimplementować dwie redundantne techniki.

Obie charakteryzują się odmiennymi właściwościami, znajdując zastosowanie dla innych algorytmów wewnątrz programu. Różnią się one pod względem gęstości zawartych informacji, szybkości dostępu do danych oraz łatwości modyfikacji. Jedna z nich skupia się na każdym z pól szachownicy (ang. *Square Centric*), druga natomiast bierze pod uwagę położenie konkretnych rodzajów bierek (ang. *Piece Centric*).

Tablica pól szachowych

Naturalnym podejściem do reprezentacji szachownicy wydało się zastosowanie 64-elementowej tablicy, w której każde pole odpowiada konkretnemu miejscu na planszy. W tej implementacji poszczególne bierki zostały zakodowane liczbami od 1 do 6, natomiast cyfry 0 i 8 odpowiednio reprezentują biały oraz czarny kolor. W ten sposób, za pomocą pojedynczych bajtów, można określić zarówno typ figury, jak i jej kolor na danym polu.

Taka struktura danych jest szczególnie użyteczna, gdy konieczna jest szybka odpowiedź na pytanie, czy na danym kwadracie znajduje się figura, a jeśli tak, to jaka. Dzięki prostemu indeksowaniu tablicy dostęp do informacji o stanie pojedynczego pola jest bardzo efektywny.

Wada tej techniki pojawia się w momencie, gdy wymagane jest odnalezienie wszystkich pól zawierających konkretny typ figury. W takim przypadku konieczna staje się iteracja całej tablicy, w celu zidentyfikowania odpowiednich miejsc. Operacja ta, szczególnie przy wielokrotnym wywołaniu, może znacząco obniżyć wydajność implementowanego algorytmu.

Tablice bitowe bierek

W celu zaadresowania powyższych spowolnień zastosowana została technika reprezentacji szachownicy za pomocą tablic bitowych. Informacja o położeniu bierek na planszy przechowywana jest w postaci tablicy liczb typu Long.

Implementacja ta wykorzystuje fakt, że tak jak szachowa plansza posiada 64 pola, tak większość współczesnych komputerów posiada architekturę 64-bitową. Oznaczając figurę na danym polu za pomocą jedynki, a pozostałe pola jako zera, można w prosty sposób zakodować całą planszę w pojedynczym słowie.

Dodatkowym atutem jest także szybkość manipulacji danymi przez operatory bitowe takie jak negacja, koniunkcja, alternatywa wykluczająca czy przesunięcie bitowe. Okazało się to być szczególnie użyteczne w przypadku algorytmów generowania możliwych posunięć.

Listing 2.1: Implementacja reprezentacji szachownicy tablicami bitowymi bierek

```
public void addPieceOnSquare(byte square, byte color, byte piece) {
    bitBoards[piece | color] |= Long.rotateLeft(1L, square-1);
    bitBoards[color] |= Long.rotateLeft(1L, square-1);
}

public void deletePieceOnSquare(byte square, byte color, byte piece) {
    bitBoards[piece | color] &= ~Long.rotateLeft(1L, square-1);
    bitBoards[color] &= ~Long.rotateLeft(1L, square-1);
}
```

2.2.2. Reprezentacja stanu

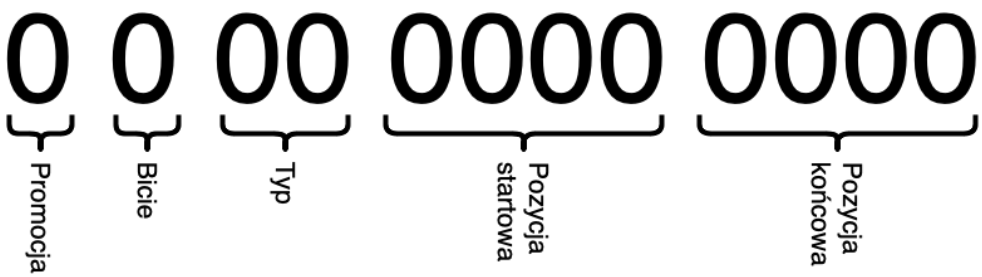
Reprezentacja stanu zawiera resztę informacji koniecznych do przedstawienia pozycji szachowej. Znajdują się w niej dane o możliwości roszad, polu en-passant, liczbie posunięć od ostatniego bicia, czy informacji, do którego gracza należy następny ruch.

W tym miejscu zawarto także strukturę HashMap zawierającą informacje o tym, jak często dane pozycja wystąpiły już w grze. Pozwoliło to na wykrywanie trzykrotnego powtórzenia stanu gry (niekoniecznie następującego bezpośrednio po sobie), które skutkuje automatycznym remisem.

2.2.3. Reprezentacja ruchu

Podobnie jak przy długiej notacji szachowej, informacjami wystarczającymi do zakodowania ruchu, jest jego pole startowe, pole docelowe, oraz w niektórych przypadkach typ promocji. Ruch szachowy jest jednak podstawową strukturą danych, na której silnik musi operować.

Z tego względu słusznym wydało się zastosowanie bardziej deskryptywnej reprezentacji, aby obliczenia wykonać przy inicjalizacji zmiennej, unikając ich powielania na późniejszym etapie wyszukiwania i oceny pozycji. Okazało to się szczególnie przydatne przy sortowaniu ruchów.



Rys. 2.2: Kodowanie ruchu szachowego

2.3. Generowanie ruchów

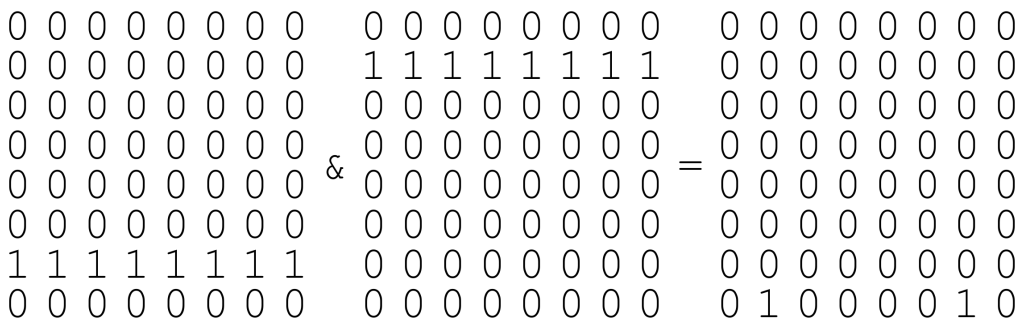
Generowanie możliwych ruchów w danej pozycji jest jednym z podstawowych, ale też kluczowych elementów każdego silnika szachowego. Jego efektywna implementacja ma znaczący wpływ na wydajność całego systemu. Opisane w poprzednim podrozdziale struktury danych reprezentacji szachownicy w dużym stopniu determinują techniki, które zostały wykorzystane w generatorze.

Podstawowym rozróżnieniem zastosowanych rozwiązań jest podział na generowanie ruchów pseudolegalnych oraz ruchów legalnych. Ruch pseudolegalny to taki, który nie narusza zasad ruchów poszczególnych bierek, natomiast istnieje możliwość, że po jego wykonaniu własny król znajdzie się w szachu. Takie rozwiązanie jest możliwe do zaimplementowania, z uwagi na pozostawienie odpowiedzialności za sprawdzenie legalności ruchu funkcji ten ruch wykonującej. Główną zaletą tego rozwiązania jest znacznie łatwiejsza implementacja.

2.3.1. Generowanie ruchów pseudolegalnych

Generowanie ruchów piona

Pionek na szachownicy ma do dyspozycji parę możliwych ruchów, które zależą od aktualnej pozycji na planszy: ruch o pole do przodu, początkowy ruch o dwa pola do przodu, bicie na skos w lewo bądź prawo, bicie en-passant. Powyższe ruchy zostały zaimplementowane przez operacje bitowe i bitowe przesunięcia na masce bitowej reprezentującej pionki. Dla przykładu technika generowania ruchów piona o jedno pole do przodu:



Rys. 2.3: Kodowanie ruchu szachowego

Generowanie ruchów hetmana, wieży i gońca

Generowanie ruchów króla i skoczka

2.3.2. Generowanie ruchów legalnych

Technika usuwania ruchów pseudo-legalnych

Perft test

Z uwagi na złożoność powyższego problemu, konieczne było wykonanie testów jednostkowych, które pozwolą sprawdzić poprawność otrzymywanych tablic ruchów.

2.4. Algorytm wyszukiwania

Z punktu widzenia teorii gier, szachy można zakwalifikować jako dwuosobową grę o sumie zerowej z pełną informacją. Oznacza to, że każdy z graczy wykonujących ruch ma dostęp do pełnej wiedzy na temat aktualnego stanu gry oraz jej historii. Co więcej, po wykonaniu ruchu, zysk gracza jest równy stratom poniesionym przez jego oponenta.

W związku ze średnią ilością możliwych posunięć w danej pozycji niemożliwe jest wprowadzenie optymalnej analizy statycznej proponującej następne ruchy. Z tego względu konieczne jest zastosowanie algorytmów dynamicznie przechodzących przez drzewo gry, w celu znalezienia najlepszego posunięcia.

2.4.1. Algorytm min-max

Wywodzący się z teorii gier algorytm min-max jest metodą minimalizacji maksymalnych strat. Znajduje on szerokie zastosowanie w grach strategicznych o sumie zerowej.

2.4.2. Iteracyjne pogłębianie wyszukiwania

2.4.3. Zarządzanie czasem

2.5. Ocena heurystyczna

2.5.1. Heurystyka stanu gry

Gdyby założyć nieskończoną moc obliczeniową komputera oraz odpowiednio dużą moc obliczeniową, jedyną heurystyką godną implementacji byłaby informacja o stanie gry, czyli wygranej, przegranej, bądź remisie. W praktyce jednak, z uwagi na ograniczenia sprzętowe i limitowaną głębokość algorytmu minimax, konieczne jest zastosowanie uproszczeń, które nie prowadzą bezpośrednio do wygranej, ale obiektywnie starają się do niej zbliżyć.

2.5.2. Heurystyka liczebności bierek

Rozdział 3

Ulepszenia dla silnika szachowego

Po opisanych w rozdziale drugim krokach silnik stał się zdolny do samodzielnej gry. W poniższej części opisane zostały zagadnienia związane z poprawą szybkości i precyzji działania systemu. W pierwszej kolejności skupiono się na ulepszeniu przeszukiwania ruchów w celu znalezienia najlepszego z posunięć. Następnie opisano zmiany co do heurystyki planszy, pozwalające silnikowi na lepszą ocenę bieżącej sytuacji gry.

3.1. Ulepszenia dla wyszukiwania

3.1.1. Biblioteka otwarć

Opis zagadnienia

Głównym problemem, który należało zaadresować, była przewidywalność i powtarzalność zaimplementowanego rozwiązania. W tym celu zdecydowano się na zaimplementowanie biblioteki otwarć.

Implementacja

Zdecydowano się na zaimplementowanie biblioteki otwarć we własnym formacie,

Rezultat

Wynikiem implementacji stał się silnik, który nie tylko rozpoczyna gry na różne sposoby, ale także o wiele szybciej je wykonuje, co pozwala zaoszczędzić czas obliczeniowy na środkowy moment gry.

3.1.2. Alfa-Beta cięcie

Zaimplementowane, pozostało wytłumaczyć o co cmon. Powołać się na Papadimitru

3.1.3. Ewaluacja cichych stanów

Po zakończeniu zwykłego min-max należy doprowadzić do stanu ”cichego” to znaczy takiego, gdzie nie ma żadnych dostępnych bić ani roszad. Inaczej może to zaburzyć poprawną interpretację pozycji przez heurystykę.

3.1.4. Sortowanie ruchów

Sortujemy ruchy zaczynając od roszad i bić w celu rozważenia ich na początku. Umożliwi to szybsze działanie alfa-beta cięcia i przez to rozważanie mniejszego drzewa decyzyjnego.

3.1.5. Tabela transpozycji

Pozycje już policzone sa haszowane Zobrist hashing oraz zapisywane. Gdy ponownie (na danym poziomie!?!) natrafimy na ten sam hash, to zwracamy wartość, nie przeszukując niżej drzewa. (Czy można tego użyć przy move ordering także?!?)

3.1.6. Okno estymacji

Z tego co rozumiem, zakłada to, że znajdziemy minimum ruch o danej jakości i przez to odcinamy alfa-beta szybciej. Jest jednak ryzyko, że takiego nie znajdziemy i będziemy musieli szukać od zera w większym oknie

3.1.7. Rozszerzanie wyszukiwania

Wydłużenie o maksymalnie dwa przeszukiwania na danej głębokości o ile jest to ruch szczególny.

3.2. Ulepszenia dla oceny heurystycznej

3.2.1. Tablice figur

Tablice dla każdej ze stron i każdej z figur w celu przypisania punktacji.

3.2.2. Ochrona króla

Sprawdzanie, czy król jest chroniony, na przykład przez piony

3.2.3. Struktura pionów

Czy piony są w rządku, czy chronią siebie wzajemnie. Najłatwiej to chyba przez bit-boardy sprawdzać

3.2.4. Moment gry

Początek - środek - koniec. Różne wartości, szczególnie dla króla w zależności od momentu gry.

3.2.5. Mobilność

Możliwość ruchów konkretnych figur, szczególnie gońców i skoczków.

Rozdział 4

Ocena siły silnika

Zastosowane w poprzednim rozdziale ulepszenia w postaci heurystyk oraz algorytmów przeszukiwania drzewa w empirycznym odczuciu autora przynosiły zamierzone rezultaty. W celu potwierdzenia tych przypuszczeń konieczne było przeprowadzenie bardziej formalnych, to jest miarodajnych i obiektywnych, testów.

4.1. Porównanie wersji silnika

4.1.1. Metodologia badawcza

Sequential Probability Ratio Test

4.1.2. Przeprowadzenie testów

4.1.3. Omówienie wyników

4.2. Porównanie z innymi silnikami

4.2.1. Metodologia badawcza

4.2.2. Omówienie wyników

4.3. Porównanie z graczami

4.3.1. Gra z autorem pracy

4.3.2. Gra z graczem na poziomie 1800 ELO

4.3.3. Gra z krajowym mistrzem szachowym

Rozdział 5

Zakończenie

5.1. Podsumowanie pracy

5.2. Możliwości dalszego rozwoju aplikacji

Bibliografia

[1] S. J. Edwards. *Standard: Portable Game Notation Specification and Implementation Guide*. 1994. URL: https://www.thechessdrum.net/PGN_Reference.txt (term. wiz. 04. 09. 2024).

[2] R. Huber i S.-M. Kahlen. *Description of the universal chess interface (UCI)*. 2006. URL: https://backscattering.de/chess/uci/ (term. wiz. 05. 09. 2024).

[3] *Lichess-bot*. URL: https://github.com/lichess-bot-devs/lichess-bot (term. wiz. 02. 10. 2024).

[4] C. E. Shannon i B. Telephone. "XXII. Programming a Computer for Playing Chess 1". W: *Philosophical Magazine Series 1* 41 (1950), s. 256–275. URL: https://api.semanticscholar.org/CorpusID:12908589.

[5] S. Vrzina. "Piece By Piece: Building a Strong Chess Engine." Bachelor's Thesis. Vrije Universiteit Amsterdam, 2023. URL: https://www.cs.vu.nl/~wanf/theses/vrzina-bscthesis.pdf.

Dodatek A

Instrukcja wdrożenia

A.1. Rozgrywka na platformie Lichess

Najłatwiejszym i najszybszym sposobem na zmierzenie się z silnikiem jest rozgrywka na platformie lichess.com, do czego autor niniejszej pracy serdecznie zaprasza. Należy odszukać silnik po nazwie użytkownika *KobayashiMaruPL* i gdy ten będzie aktywny zaprosić go do pojedynku.

W celu implementacji tego rozwiązania została wykorzystana biblioteka lichess-bot, która stanowi pośrednik pomiędzy silnikami wykorzystującymi UCI a API platformy Lichess. [3]

A.2. Kompilacja kodu i połączenie z GUI

Dodatek B

Protokół UCI

B.1. Wykorzystane komendy

Tab. B.1: UCI - komunikacja GUI do silnika

Komenda	Opis działania
uci	Wytłumaczenie działania komendy
debug	Wytłumaczenie działania komendy
isready	Wytłumaczenie działania komendy
setoption	Wytłumaczenie działania komendy

Tab. B.2: UCI - komunikacja silnika do GUI

Komenda	Opis działania
id	Wytłumaczenie działania komendy
uciok	Wytłumaczenie działania komendy
readyok	Wytłumaczenie działania komendy

B.2. Przykład użycia