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
FIDE (fr. *Fédération Internationale des Échecs*) Międzynarodowa Federacja 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 zmianę w swojej ponad dwu i pół tysiącletniej 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 Shannona, 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 metodologię 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: poprawie prędkości przeszukiwania stanów oraz na udoskonaleniu precyzji heurystycznej oceny pozycji. Jako że dla słabych silników o wiele większe znaczenie w poprawie poziomu jego gry ma pierwszy z tych aspektów, omówiono go najpierw. [9] 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. 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, jej aspektów. Wiele z opisanych rozwiązań algorytmicznych można znaleźć jedynie w literaturze anglojęzycznej. W miejscach, gdzie tłumaczenie uznano za niewystarczające, podano także oryginalne nazewnictwo.

Rozdział 2

Implementacja silnika szachowego

2.1. Komunikacja z systemem

2.1.1. Notacja Forsytha-Edwardsa

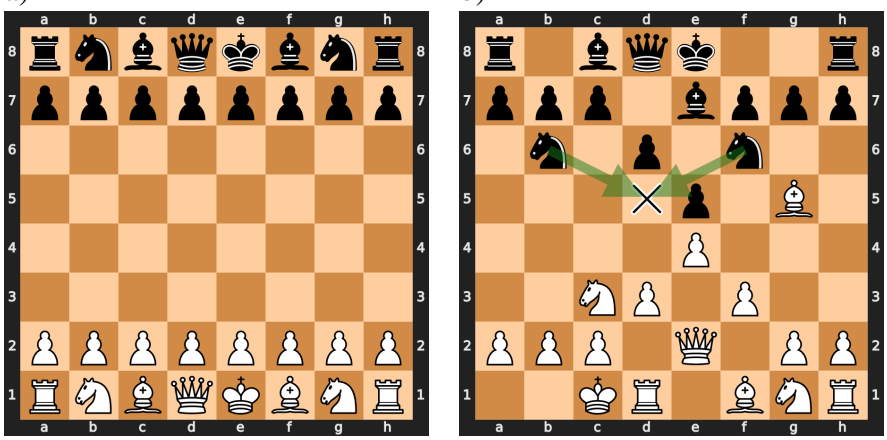
W 1950 roku amerykański matematyk Claude Shanon na łamach ”Philosophical Magazine” opublikował pracę naukową zatytułowaną ”Programowanie komputera do gry w szachy”. [8] Stała się ona teoretyczną podstawą dla dalszego rozwoju silników szachowych. Zawarte w niej zostało między innymi oszacowanie, co do ilości możliwych pozycji szachowych, wynoszące 10^{43}. Oznacza to, że liczba legalnych ułożeń planszy o rzędy wielkości przewyższa 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).

Notacja FEN jest sześciopolową 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”. [3]

a) b)



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. 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) [4] 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

Struktury danych, wybrane do reprezentacji szachownicy, w dużym stopniu determinują implementację funkcji generowania dostępnych ruchów, a co za tym idzie, bezpośrednio wpływają na wydajność całego silnika. Z tego względu ich dobór musiał zostać odpowiednio zaplanowany, uwzględniając ogół architektury programu. Po zapoznaniu się z proponowanymi w literaturze rozwiązaniami, w projekcie zdecydowano się zastosować dwie redundantne techniki reprezentacji 64 pól szachowych.

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*).

Przy procesie implementacji omawianego fragmentu należało zachować szczególną ostrożność, aby ciągłe aktualizowanie dwóch niezależnych struktur danych nie doprowadziło do ich rozspójnienia, a co za tym idzie wprowadzenia trudnych do zdebugowania błędów. Ryzyko ich pojawienia zostało zminimalizowane dzięki wprowadzeniu testów jednostkowych regularnie kontrolujących poprawność operacji podczas enumeracji drzewa gry.

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 objawia 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, powszechnie znanych w środowisku programistów szachowych jako ang. *Bitboards*. Informacja o położeniu konkretnego typu bierki na planszy przechowywana jest w postaci tablicy liczb o rozmiarze 8 bajtów.

Implementacja ta wykorzystuje kodowanie 64 polowej tablicy szachowej w 64 pojedynczych bitach zmiennej. Oznaczając figurę na danym polu za pomocą jedynki, a pozostałe pola jako zera, można w prosty sposób zawrzeć informacje o planszy w dwunastu pojedynczych słowach. Pozycje bierek nie są natomiast jedyną informacją, którą można zakodować w tablicach bitowych. Uprzednio policzone maski możliwych ruchów czy atakowanych pól, to tylko niektóre z możliwych zastosowań, które na późniejszych etapach implementacji znacznie przyspieszały obliczenia.

Powszechne użycie 64-bitowej architektury sprawiło, że operacje na danych takiej wielkości są bardzo efektywne, gdyż nie wymagają rozbicia instrukcji na mniejsze części. Dodatkowym atutem jest także szybkość manipulacji danymi przez operatory bitowe takie jak negacja, koniunkcja, alternatywa wykluczająca czy przesunięcie bitowe. Z reguły, a w szczególności na starszych procesorach, operacje bitowe są szybsze, niż ich odpowiedniki w postaci operacji arytmetycznych.

2.2.2. Reprezentacja stanu

Reprezentacja stanu zawiera resztę informacji koniecznych do przedstawienia pozycji. Znajduje się w niej logika dotycząca możliwych roszadach, pola en-passant, czy liczby posunięć od ostatniego bicia. Stan gry posiada również wiedzę o ruchu, który do danej pozycji doprowadził, aby ułatwić jego cofanie przez inne komponenty systemu. W tej klasie zawarto także strukturę HashMap z informacjami, jak często dana pozycja wystąpiła już w grze.

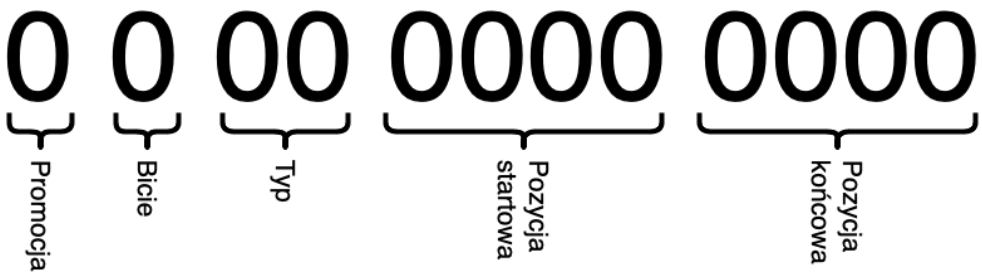
Pozwoliło to na wykrywanie wielokrotnego powtórzenia pozycji, które niekoniecznie następują bezpośrednio po sobie. Choć według oficjalnych zasad automatyczny remis następuje dopiero po pięciokrotnym powtórzeniu, większość platform do rozgrywek online uznaje trzykrotne powtórzenie za obligatoryjny remis. Taka też wersja została zaimplementowana.

Reguła remisu dyktowanego pięćdziesięcioma ruchami bez bicia lub ruchu pionem nie została zaimplementowana. Ze statystyk wynikło, że takie sytuacje zdarzały się niezwykle rzadko, a gdy się pojawiały, pozycje obydwu graczy można było uznać za równą. Remis w takiej sytuacji uznano za korzystny dla obydwu graczy.

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 wykonywane były przy inicjalizacji zmiennej, unikając ich powielania na późniejszym etapie wyszukiwania i oceny pozycji. Zawarto także informacje o tym, czy ruch jest roszadą, czy może podwójnym ruchem pionka, czy może jest biciem klasycznym, bądź en-passant. 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 także kluczowych elementów każdego silnika szachowego. Jego efektywna implementacja to taka, w której silnik spędza jak najmniej czasu, pozostawiając możliwości obliczeniowe na przeszukiwanie drzewa decyzyjnego.

Podstawowym rozróżnieniem stosowanych rozwiązań jest podział na generowanie ruchów pseudolegalnych oraz ruchów legalnych. [2] Ruch pseudolegalny to taki, który nie narusza zasad ruchów poszczególnych bierek, natomiast istnieje ryzyko, że po jego wykonaniu własny król znajdzie się w szachu. Takie rozwiązanie jest możliwe do zastosowania, z uwagi na pozostawienie odpowiedzialności za sprawdzenie legalności ruchu funkcji, która ten ruch wykonuje. Główną zaletą tej metody jest znacznie łatwiejsza implementacja.

Kolejnym aspektem wartym zaznaczenia, jest wydzielenie oddzielnego generatora dla ruchów zbijających. Z perspektywy dalszej implementacji znacznie ułatwiło to rozwój silnika o ewaluacje stanów cichych oraz sortowanie ruchów, o czym wspomniano w następnym roździale.

2.3.1. Operacje na mapach bitowych

Jak już wcześniej wspomniano, iteracyjne generowanie ruchów każdej z bierek z osobna mogłoby okazać się zbyt czasochłonne. Z tego względu większość dostępnych posunięć tworzona jest dzięki przekształcaniu map bitowych reprezentujących konkretny typ figury. Optymalizacja przynosi szczególny efekt przy generowaniu ruchów pionów, ze względu na ich znaczną ilość przez większość czasu gry.

Pion ma do dyspozycji parę możliwych ruchów, które należało zaimplementować: ruch o jedno pole do przodu, ruch o dwa pola do przodu, bicie w lewo i bicie w prawo.

Maski dla każdego z tych ruchów zostały wygenerowane w oddzielnej metodzie. Przykładowo, dla ruchu o dwa pola do przodu wzór wygląda następująco:

$moves = empty$	pole docelowe musi być puste
$moves = moves \wedge (pionki_w \ll 16)$	biały pion musi być dwa wiersze niżej
$moves = moves \wedge (empty \ll 8)$	wiersz niżej musi być pusty
$moves = moves \wedge rank4$	pole docelowe musi być w czwartym wierszu

W taki sposób stworzono maskę bitową końcowych pól, na które piony mogą się przesuwać, skacząc o dwa pola. Na otrzymanym wyniku należy przeprowadzić serializację, to jest przekształcić go na listę dostępnych ruchów. Aby nie iterować przez wszystkie 64 bity, zastosowano technikę zwaną Bit Scan, która zwraca indeks najbardziej istotnego bitu na masce, dodaje ruch do listy, a następnie usuwa ten bit z maski. Operacja jest wykonywana do momentu, aż maska pozostanie pusta.

Listing 2.1: Metoda dodająca ruchy z maski wraz z przykładowym wywołaniem

```
void addMovesFromMask(long movesMask, byte moveType, byte offset) {
   while(movesMask != 0L) {
       index = (64 - Long.numberOfLeadingZeros(movesMask));
       possibleMoves.add(new Move(index+offset, index, moveType));
       movesMask &= ~(1L << (index - 1));
   }
}

addMovesFromMask(allDoublePushMask, DOUBLE_PAWN_PUSH, -16);
```

Legalne ruchy króla i skoczka generowane są w sposób analogiczny, z tą jednak różnicą, że maski dostępnych ruchów tworzone są nie przez przesunięcia bitowe, ale przy inicjalizacji silnika generowana jest tablica dla każdego z pól startowych.

2.3.2. Hyperbola Quintessence

Generowanie ruchów hetmana, wieży i gońca odbywa się w sposób odmienny. Wynika to z faktu, że figury te poruszają się o dowolną liczbę pól w danym kierunku, aż do momentu napotkania innej bierki na swojej drodze. W przypadku bierki przeciwnika możliwe jest bicie, w przypadku bierki własnej, należy zatrzymać się o pole wcześniej. Choć są to trzy różne figury, to mają do dyspozycji dwa możliwe ruchy, ruch w linii prostej, jak wieża, oraz ruch po przekątnej, jak goniec. Ruchy hetmana są natomiast sumą dwóch powyższych generatorów.

Większość silników szachowych korzystających z masek bitowych implementuje funkcję, która w literaturze zwana jest jako ang. *Hyperbola Quintessence*. Pozwala ona na wygenerowanie dostępnych ruchów w jednej prostej oraz na uniknięcie zawiłej logiki iteracyjnej.

$o = 11010101$	$o' = 10101011$	pola zajęte przez bierki
$r = 00010000$	$r' = 00001000$	pole figury dla której generujemy ruchy
$o - r = 11000101$	$o' - r' = 10100011$	pola zajęte minus pole figury
$\alpha = o - 2r = 10110101$	$\beta = o' - 2r' = 10011011$	pola zajęte dwukrotnie minus pole figury
$(\alpha \oplus \beta') \wedge \neg\gamma$	01101100	maska legalnych ruchów

$\oplus$ — alternatywa wykluczająca
$\wedge$ — koniunkcja
x' — odwrócenie bitów
γ — maska pól zajętych przez własne bierki

2.3.3. Ruchy specjalne

Pozostałymi ruchami do zaimplementowania było bicie w przelocie oraz cztery roszady. Z uwagi na znikomą ilość takich posunięć w danej pozycji oraz na złożoną logikę tych ruchów, zostały one wygenerowane explicite z zasad gry.

Oba typy ruchów wymagały dodatkowej weryfikacji z danymi dostępnymi w reprezentacji stanu gry. Pierwszy z nich - en-passant - został wygenerowany przez nałożenie na siebie maski pola bicia w przelocie oraz maski pinów, odpowiednio przesuniętych o ± 7 oraz ± 9 kratek.

Aby uzyskać prawo roszady, należy spełnić następujące warunki:

1. Ani król, ani wieża biorąca udział w roszadzie nie mogły wykonać wcześniej ruchu.
2. Pola między królem a wieżą muszą być puste.
3. Król nie może znajdować się w szachu.
4. Król nie może przechodzić bądź kończyć ruch na polach, atakowanych przez bierki przeciwnika.

2.3.4. Generowanie ruchów legalnych

Technika usuwania ruchów pseudo-legalnych

Po zaimplementowaniu logiki opisanej powyżej silnik był zdolny do generowania posunięć pseudolegalnych. Natomiast ruch, po którym własny król znajduje się w szachu, nie tylko jest ruchem nieoptymalnym, ale również z punktu widzenia reguł FIDE nielegalnym.

W literaturze można znaleźć kilka podejść do problemu odfiltrowywania ruchów pseudolegalnych. Niektóre z nich korzystają z dodatkowych masek bitowych, reprezentujących pola atakowane przez bierki danej strony. W niniejszej implementacji zastosowano jednak rozwiązanie subiektywnie dla autora łatwiejsze.

Po wykonaniu konkretnego ruchu, w miejscu, gdzie znajduje się król, stawiana jest każda figura z kolei, oraz generowane są dostępne bicia. Jeśli wśród bić znajduje się bierka przeciwnika, tego samego typu, co aktualnie podstawiona, oznacza to, że król znajduje się w szachu, a więc posunięcie nie należy do kategorii ruchów legalnych.

Test wydajności

Generatory ruchów szachowych posiadają skomplikowaną logikę. Z tego względu bardzo łatwo o popełnienie błędu w implementacji. Dopuszczenie choćby jednego błędu, skutkować będzie jego propagacją na większych głębokościach, a w skrajnych przypadkach doprowadzi do błędu programu.

Ocena poprawności dzięki grze w szachowych posiada szereg wad dyskwalifikujących tę metodę. Przede wszystkim, wymaga ręcznego rozegrania wielu partii, w celu odwiedzenia wystarczającej liczby węzłów drzewa. Co więcej, konkretna sytuacja jest czasochłonne do zreplikowania, a pojawiające się błędy trudne do zalogowania.

Z tego względu zastosowano Test Wydajności (ang. *Performance Testing*, w skrócie Perft Test). Choć nazwa mogłaby wskazywać na testowanie prędkości generowanych ruchów, test ten można przeprowadzić także w celu kontroli poprawności. Metoda ta opiera się o wykonanie algorytmu DFS dla ograniczonej głębokości na drzewie gry, przy jednoczesnym zliczaniu odwiedzonych węzłów. Tak otrzymane wyniki można porównać z konsensusem osiągniętym przez twórców silników szachowych. Jako punkt odniesienia autor uznał wyniki generowane przez silnik Stockfish. Przeprowadzenie testów z różnych pozycji startowych oraz dla różnych głębokości pozwoliło na potwierdzenie poprawności implementowanego generatora, z prawdopodobieństwem graniczącym z pewnością. Przykładowe wyniki zaprezentowano w dodatku C.

2.4. Ocena heurystyczna

Funkcje heurystyczne są komponentami silnika szachowego, które odpowiadają za ocenę konkretnej pozycji szachowej, z punktu widzenia jednego z graczy. Jest to istotne ze względu na to, że umożliwiają silnikowi wybór i wykonywanie takich ruchów, które prowadzą do osiągnięcia najkorzystniejszej z punktu widzenia silnika pozycji.

Z czysto teoretycznego punktu widzenia, nieskończona moc obliczeniowa pozwalałaby na przeszukanie pełnego drzewa gry, a co za tym idzie, jedyną funkcją heurystyczną godną implementacji, byłaby funkcja zwracająca 1 w przypadku wygranej, oraz -1 w przypadku przeciwnym. W rzeczywistości natomiast mnogość możliwych ruchów z każdej pozycji nakłada limit co do głębokości przeszukiwania i konieczności oceny pozycji, które nie są liśćmi drzewa, to jest pozycjami, które nie kończą partii. Z tego względu konieczne było zaimplementowanie funkcji heurystycznych, które, choć nie prowadzą gracza bezpośrednio do wygranej, to przybliżają go do tego celu na określone sposoby.

W podstawowej wersji silnika zaimplementowano dwie funkcje heurystyczne.

Pierwsza z nich zwraca MAX w przypadku mata króla przeciwnika, MIN w przypadku mata własnego króla, oraz -5000 w przypadku pata, bądź trzykrotnego powtórzenia pozycji, w celu demotywowania silnika do dążenia do remisu. Druga natomiast opiera się na intuicyjnym założeniu, że korzystniejsza pozycja to taka, w które gracz ma więcej bierek na planszy niż przeciwnik. Różnica między liczbą bierek konkretnego typu dla gracza oraz jego oponenta jest przemnażana przez wagę poszczególnej bierki. W tym celu skorzystano ze skali zaproponowanej przez Claude Shannona, w której odpowiednio hetman, wieża, goniec, skoczek oraz pion mają wartości 900, 500, 300, 300 oraz 100.

W dalszej części pracy przedstawiono kolejne funkcje heurystyczne, mające na celu poprawę precyzji oceny pozycji. Klasa oceny heurystycznej silnika szachowego zwraca ocenę pozycji P dla koloru, który aktualnie wykonuje ruch według zasady:

$$Ocena(P) = \begin{cases} H_0(P) & \text{if } H_0(P) \neq 0 \\ \sum_{i=1}^{n}(c_i * H_i(P)) & \text{else} \end{cases} \tag{2.1}$$

H_0 – heurystyka win-loss-draw
H_i – kolejne funkcje heurystyczne
c_i – waga heurystyki i

Manipulowanie wagami konkretnych heurystyk pozwoliło na przeprowadzenie testów porównawczych i ich dostrojenie w celu uzyskania jak najdokładniejszej oceny.

2.5. Algorytm wyszukiwania

Z punktu widzenia teorii gier, szachy można klasyfikować na wiele różnych sposobów.

1. W wydaniu europejskim są one grą dwuosobową.
2. Jest to gra o sumie stałej, ponieważ w wyniku wykonywania dowolnej liczby ruchów proporcja zysków jednego gracza pozostaje niezmienna w stosunku do strat drugiego. Gry o sumie stałej można sprowadzić do gier o sumie zerowej.
3. Rozgrywka toczy się w postaci ekstensywnej, a więc ruchy wykonywane są na przemian.
4. Jest to gra skończona, z uwagi na regułę trzykrotnego powtórzenia.
5. Każdy z graczy posiada dostęp do doskonałej informacji.

Najbardziej istotne, z punktu tworzenia silnika szachowego, jest połączenie dwóch pierwszych z wymienionych cech. Takie gry nazywane są ściśle konkurencyjnymi. Innymi

słowy, aby uzyskać maksymalną wypłatę, gracz dąży do tego, by zminimalizować sumę wypłat przeciwnika. [7]

W algorytmice problemy tego typu można rozwiązać za pomocą algorytmów dążących do minimalizowania maksymalnych strat.

2.5.1. Algorytm minimalizowania maksymalnych strat

Algorytm minimax polega na przeszukiwaniu drzewa gry. Z uwagi na mnogość możliwych decyzji, proces ogranicza się do określonej głębokości. Liściom drzewa przypisywane są obliczone wartości heurystyczne. Idąc wzwyż grafu, na kolejnych poziomach, nadawane są wartości maksymalne dla protagonisty i minimalne dla antagonisty. Ruch, który prowadzi z korzenia do wierzchołka na głębokości jeden z największą wartością, jest ruchem proponowanym przez algorytm.

W silniku zastosowano skróconą wersję powyższego algorytmu, wykorzystując fakt, że $max(\alpha, \beta) = -min(-\alpha, -\beta)$.

Listing 2.2: Implementacja algorytmu negaMax

```
negaMax(int depth) {
    moves = moveGenerator.generateMoves();
    if(depth == 0 || moves.isEmpty())
            return evaluator.evaluateForCurrentColor();

    int bestMoveValue = MINIMUM;
    for(Move move : moves) {
        board.makeMove(move);
        int score = -negaMax(depth-1);
        if(score > bestMoveValue)
            bestMoveValue = score;
        board.unmakeMove();
    }
    return bestMoveValue;
}
```

2.5.2. Zarządzanie czasem

Estymacja dostępnego czasu

Większość pojedynków szachowych odbywa się w czasie rzeczywistym, z narzuconym ograniczeniem co do łącznego czasu na wykonanie ruchu. Przekroczenie tego limitu stanowiłoby automatyczną wygraną oponenta. Koniecznym było zatem zaimplementowanie mechanizmu zarządzania, który pozwoliłby na podział pozostałego czasu, na wykonanie poszczególnych ruchów. Pozostały czas otrzymany przez UCI w formie `go wtime <wtime> btime <btime> winc <winc> binc <binc>` pozwolił na implementacje rozwiązania opartego na wzorze:

$$\text{Est pozostałych ruchów} = \max(40 - \text{wykonaneRuchy}, 10)$$

$$\text{Est pozostałego czasu} = time_{color} + (inc_{color} * \text{Est pozostałych ruchów})$$

$$\text{Propozycja czasu} = \frac{\text{Est pozostałych ruchów}}{\text{Est pozostałego czasu}}$$

Założono średnią ilość ruchów na poziomie czterdziestu.

Iteratywne pogłębianie wyszukiwania

Prędkość wykonania algorytmu negaMax zależy od wielu czynników, takich jak głębokość wyszukiwania, aktualna pozycja, prędkość generowania ruchów czy ilość dostępnych

dla programu zasobów. Nie ma możliwości oszacowania, ile czasu zajmie jego wykonanie, a przerwanie w trakcie działania, może skutkować wybraniem skrajnie nieopłacalnych posunięć.

W tym celu zaimplementowano rozwiązanie iteracyjnego pogłębiania wyszukiwania, polegającego na szukaniu najlepszego ruchu dla kolejnych głębokościach. W momencie upływu czasu na obliczenia zwracana jest wartość otrzymana z najgłębszego, w pełni ukończonego przeszukiwania.

Takie podejście mogłoby wydawać się skrajnie nieoptymalne, gdyż wymaga wielokrotnego generowania drzewa gry. W praktyce jednak jest to rozwiązanie gwarantujące otrzymanie w miarę dobrego ruchu, a ulepszenia algorytmu takie jak dynamiczne sortowanie ruchów pozwalają na osiągnięcie szybszych rezultatów, niż przy przeszukaniu drzewa od razu dla danej głębokości. [1]

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, były otwarcia partii rozgrywanych przez silnik. Gry szachowe zawsze rozpoczynają się z tego samego położenia, więc wiadomo, że system znajdzie się w wielu pozycjach bezpośrednio wynikających z pierwszych posunięć. Program natomiast na obliczenia poświęcał wiele czasu, który lepiej byłoby spożytkować w środkowym etapie rozgrywki.

Profesjonalni gracze, korzystając z dorobku i doświadczenia wielu pokoleń szachistów, zapamiętują pierwsze posunięcia do wykonania. Tym samym oszczędzają czas, który pozostaje na bardziej złożone pozycje w środkowej fazie gry.

Implementacja

W celu zaimplementowania biblioteki otwarć, dostępne posunięcia czytane są z pliku, a następnie zapisywane przy inicjalizacji silnika do HashMapy <`String, []Moves`>. Podczas działania silnika, FEN aktualnej pozycji porównywalny jest z dostępnymi kluczami, a ruch losowo wybierany spośród dostępnych w tablicy. W przypadku nieznalezienia klucza program przechodzi do gry środkowej z wykorzystaniem algorytmu negaMax.

W literaturze znanych jest wiele formatów zapisu biblioteki otwarć, np. EPD, PGN czy Bin-format. System wykorzystuje bibliotekę wygenerowaną przez Sebastiana Lague i dostępną na licencji MIT w implementacji jego silnika. [5]

Rezultat

Efektem prac stał się silnik, który początkowe posunięcia wykonuje za pomocą podpiętej biblioteki otwarć. Dzięki temu pierwsze ruchy wykonywane są bardzo szybko, pozostawiając cenny czas obliczeniowy na dalszą grę, tym samym zwiększając precyzję.

We wcześniejszej wersji silnik był deterministyczny, to jest, dla konkretnej głębokości i pozycji, algorytm zwracał ten sam najlepszy ruch. Dodatkowym atutem stała się jego nieprzewidywalność. Początkowe posunięcia są zrandomizowane, co pozwala na jego testowanie w większej ilości wariantów, a także umila rozgrywkę użytkownikowi.

3.1.2. Alfa-Beta cięcie

Usprawnia przeszukiwania drzewa.

Opis zagadnienia

Implementacja

Rezultat

3.1.3. Ewaluacja cichych stanów

Usuwa efekt horyzontu.

Opis zagadnienia

Implementacja

Rezultat

3.1.4. Sortowanie ruchów

Przyspiesza pruning drzewa.

Opis zagadnienia

Implementacja

Rezultat

3.1.5. Tabela transpozycji

Minimalizuje czas obliczania heurystyki.

Opis zagadnienia

Implementacja

Rezultat

3.1.6. Okno estymacji

Przyspiesza alpha-beta cięcie.

Opis zagadnienia

Implementacja

Rezultat

3.1.7. Rozszerzanie wyszukiwania

Pogłębia ocenę niektórych pozycji.

Opis zagadnienia

Implementacja

Rezultat

3.2. Ulepszenia dla oceny heurystycznej

3.2.1. Tablice figur

Znacznie polepsza ocene stanu gry.

Opis zagadnienia

Implementacja

Rezultat

3.2.2. Ochrona króla

Zmniejsza ryzyko szacha.

Opis zagadnienia

Implementacja

Rezultat

3.2.3. Struktura pionów

Zachęca do struktury pionów i ich promocji.

Opis zagadnienia

Implementacja

Rezultat

3.2.4. Moment gry

Przyspiesza matowanie w późnej fazie rozgrywki.

Opis zagadnienia

Implementacja

Rezultat

3.2.5. Mobilność

Usprawnia pozycjonowanie hetmana, wieży i gońca.

Opis zagadnienia

Implementacja

Rezultat

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

Ustalona liczba rozgrywek

Porównywanie oceny pozycji

Porównywanie liczby odwiedzonych węzłów

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

Platforma lichess

Wybór oponentów

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] *Chess Programming Wiki: Iterative Deepening*. URL: https://www.chessprogramming.org/Iterative_Deepening (term. wiz. 11. 10. 2024).

[2] *Chess Programming Wiki: Move Generation*. URL: https://www.chessprogramming.org/Move_Generation (term. wiz. 08. 10. 2024).

[3] 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).

[4] 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).

[5] S. Lague. *Biblioteka otwarć*. URL: https://github.com/SebLague/Chess-Coding-Adventure/blob/Chess-V2-UCI/Chess-Coding-Adventure/resources/Book.txt (term. wiz. 10. 10. 2024).

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

[7] T. Płatkowski. *Matematyka stosowana, Wstęp do Teorii Gier*. URL: https://mst.mimuw.edu.pl/wyklady/wtg/wyklad.pdf (term. wiz. 09. 10. 2024).

[8] 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.

[9] 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. [6]

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`	Określenie używanego protokołu. Po otrzymaniu komendy silnik powinien odpowiedzieć `id` oraz listą dostępnych opcji.
`setoption name <name> value <value>`	Instrukcja zmiany wewnętrznego parametru silnika. Nazwa parametru jest podawana przez silnik. Wartość może być typu bool, int, String.
`position [startpos | <fen>] moves <moves>`	Ustawienie pozycji zaczynając od startowej bądź z podanego FEN. Możliwe wykonanie dalszych ruchów przekazanych w formacie LAN.
`isready`	Zapytanie, służące synchronizacji GUI z silnikiem. Gdy silnik nie przetwarza poleceń powinien odpowiedzieć `uciok`.
`go wtime <wtime> btime <btime> winc <winc> binc <binc>`	Instrukcja rozpoczęcia szukania ruchu. Kolejne argumenty w milisekundach wyrażają: pozostały czas białego i czarnego, dodatkowy czas per ruch białego i czarnego.
`debug [on | off]`	Zmiana trybu diagnostycznego silnika.
`quit`	Kończy działanie silnika.

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

Komenda	**Opis działania**
`id [name | author] <value>`	Instrukcja służąca identyfikacji silnika przez GUI.
`option name <name> type <type>`	Wypisanie dostępnych opcji oraz typu ich działania.
`uciok`	Wysyłany po `id` oraz `option`, aby potwierdzić poprawne wykonanie komedny `uci`.
`readyok`	Odpowiedź na `isready`. Synchronizuje GUI z silnikiem.
`info <value>`	Przekazywanie GUI informacji nad aktualnym stanem obliczeń najlepszego ruchu.
`bestmove <move>`	Odpowiedź na instrukcję `go`. Zwraca ruch proponowany przez silnik.

B.2. Przykład użycia

Tab. B.3: Przykład użycia UCI

Output GUI	**Output silnika**
uci	
	id name KobayashiMaru id author Krzysztof Antoni Wiśniewski option name OwnBook type check uciok
setoption name OwnBook value false isready	
	readyok
position startpos go wtime 60000 btime 60000 winc 600 binc 600	
	info depth 1 nodes 21 pv g1f3 info depth 2 nodes 177 pv d2d4 g8f6 ... bestmove d2d4
position startpos moves d2d4 d7d5 go wtime 58000 btime 55000 winc 600 binc 600	
...	...
quit	

Dodatek C

Wyniki Perft