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 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 na platformie szachowej Lichess wynosi pomiędzy 1600 a 1650 ELO, co mieści go wśród 40% najlepszych graczy.

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, 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 on Lichess platform can be estimated in the range between 1600 and 1650 ELO, which places it at the top 40% of players.

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
SPRT (ang. *Sequential Probability Ratio Test*) Sekwencyjny Test Probabilistyczny
MVV-LVA (ang. *Most Valuable Victim – Least Valuable Attacker*) Najbardziej Wartościowa Ofiara – Najmniej Wartościowy Atakujący
DFS (ang. *Depth First Search*, w skrócie DFS) Przeszukiwanie w głąb
LLM (ang. *Large Language Model*, w skrócie LLM) Duży Model Językowy
ACPL (ang. *Average Centipawn Loss*, ACPL) Średnia Strata w Centypionach

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 złożonością 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 połowy 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 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 następny ruch był obliczany przez człowieka.

Najstarszy program uruchomiony na komputerze, który pozwalał na przeprowadzenie rozgrywki, powstał w 1957 roku [11]. 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, Garrim 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 tę grę 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 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 interfejsem graficznym.
- 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 Rozdziale 2 opisano elementy oprogramowania konieczne do stworzenia podstawowej wersji silnika szachowego. Przedstawiono metody komunikacji z interfejsem, techniki reprezentacji pozycji i generowania dostępnych ruchów. Zaprezentowano algorytm minimalizowania strat i maksymalizowania zysków wraz z oceną heurystyczną oraz metodologię zarządzania czasem gry. Omówiono niektóre z testów jednostkowych stworzonych w celu sprawdzenia poprawności działania aplikacji.

Rozdział 3 poświęcono pracom nad ulepszeniem systemu. Przedstawiono zaimplementowane rozwiązania mające na celu poprawę poziomu gry. Skupiono się na dwóch kierunkach: poprawie prędkości przeszukiwania stanów oraz na udoskonaleniu precyzji heurystycznej oceny pozycji. Z uwagi na to, że dla słabych silników o wiele większe znaczenie w poprawie poziomu gry ma pierwszy z tych aspektów, omówiono go w pierwszej kolejności[21]. W literaturze można znaleźć więcej niż jedno możliwe podejście do niektórych z opisanych problemów algorytmicznych. Ze względu na ich mnogość, w pracy przedstawiono parę możliwych rozwiązań, a następnie wskazano, czym kierował się autor przy wyborze jednego z nich.

Rozdział 4 poświęcono testom wydajnościowym oraz jakościowym. Przetestowano program pomiędzy różnymi jego wersjami. Przedstawiono wyniki gry systemu przeciwko innym, publicznie dostępnym silnikom. W dodatkach do pracy zamieszczono instrukcję wdrożenia aplikacji w celu uruchomienia 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" [20]. 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.

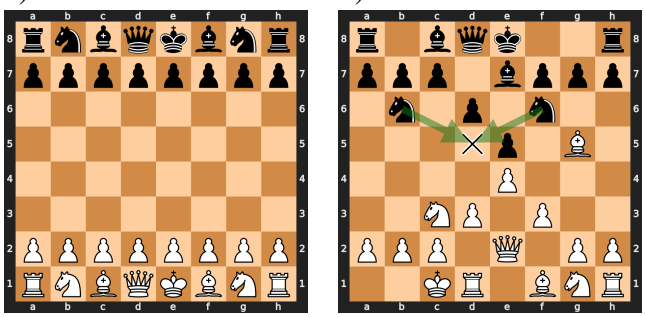
Aby umożliwić użytkownikowi efektywną komunikację z programem, należało w pierwszej kolejności sprecyzować format, w jakim zostaną dostarczone informacje dotyczące aktualnej pozycji. Standardem wykorzystywanym nie tylko w większości silników, ale także w pojedynkach rozgrywanych online, jest Notacja Forsytha-Edwardsa (ang. *Forsyth–Edwards Notation*, w skrócie FEN).

Notacja FEN jest sześciopolową linią znaków ASCII, która pozwala na precyzyjne określenie aktualnego stanu gry. Na pierwszym polu zakodowano pozycję bierek z perspektywy białego gracza, gdzie wielkimi literami oznaczono bierki białe, małymi natomiast bierki czarne. Każdą z nich opisano skrótem pochodzącym od ich angielskich nazw. Na następnych polach zawarto kolejno informację o: kolorze następnego ruchu, możliwości krótkich i długich roszad obydwu graczy, polu bicia w przelocie oraz liczbie reprezentującej posunięcia od ostatniego bicia bądź ruchu pionem i liczbie pełnych ruchów.

Pełna specyfikacja FEN dostępna jest w dokumentacji „Portable Game Notation" [9].

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`

a) b)



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

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 to samo pole, występuje dwuznaczność zapisu, jak pokazano strzałkami na rysunku 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 w niej informacje o polu startowym oraz polu docelowym ruchu usuwają 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) [12] 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, tak by uwzględniał 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 i znajdują zastosowanie dla innych algorytmów wewnątrz programu. Różnią się one pod względem gęstości zawartych informacji, szybkości dostępu do danych oraz łatwości modyfikacji. Pierwsza 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 tym samym do wprowadzenia trudnych do odnalezienia 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 liczby 0 i 8 odpowiednio reprezentowały biały oraz czarny kolor. W ten sposób, za pomocą pojedynczego bajtu, można było określić zarówno typ figury, jak i jej kolor na danym polu.

Taka struktura danych jest szczególnie użyteczna, gdy oczekiwana jest szybka odpowiedź na pytanie, czy na konkretnym 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 ujawnia się w momencie, gdy wymagane jest odnalezienie wszystkich pól zawierających konkretny typ figury. W takim przypadku konieczną 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 rozwiązania wyżej opisanych problemów zastosowana została technika reprezentacji szachownicy za pomocą tablic bitowych, powszechnie znanych w środowisku programistów szachowych jako ang. *Bitboards*. Informacja o położeniach bierek konkretnego typu 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 obecność figury na danym polu za pomocą jedynki, a jej brak jako zero, można w prosty sposób zawrzeć informacje o planszy w dwunastu pojedynczych liczbach typu `Long`. Pozycje bierek nie są natomiast jedyną informacją, którą można zakodować ten sposób. Zainicjowane w konstruktorze maski: możliwych ruchów i atakowanych pól, to przykłady 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 szybkość manipulacji danymi przez operatory bitowe takie jak: negacja, koniunkcja, alternatywa wykluczająca i przesunięcie bitowe. Z reguły, w szczególności na starszych procesorach, operacje bitowe są szybsze, niż ich odpowiedniki w postaci operacji arytmetycznych.

2.2.2. Reprezentacja stanu

Reprezentacja stanu zawiera pozostałe informacje konieczne do przedstawienia pozycji. Znajduje się w niej logika dotycząca możliwych roszad, pola en-passant, czy liczby posunięć od ostatniego bicia. Stan gry posiada również dane o ruchu, który do określonej 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 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 matem nie została wprowadzona. Wynikało to z faktu, że po wykonaniu ruchu należałoby sprawdzić możliwe odpowiedzi rywala, co okazałoby się kosztowne. W praktyce sytuacja ta nie występuje zbyt często i w najgorszym wypadku skutkuje remisem, natomiast brak implementacji tej reguły znacząco przyspiesza obliczenia.

Haszowanie Zobrist

Struktura `HashMap` zawierająca informacje o wcześniejszych pozycjach w grze w pierwotnej wersji została zaimplementowana z kluczami reprezentującymi pozycję FEN. Z perspektywy czasu okazało się, że każdorazowe generowanie klucza w postaci zmiennej typu `String` było niezmiernie kosztowne.

Postanowiono zatem jako klucza użyć wartości zwracanej przez haszowanie Zobrist. Polega ono na przypisaniu kolorowi czarnemu oraz każdej figurze na każdym możliwym polu planszy losowej liczby typu `long`. Korzystając ze standardowego generatora liczb pseudolosowych, dostępnego w języku Java, otrzymano 769 liczb.

Wartość haszowania dla danej pozycji obliczona została jako alternatywa wykluczająca wszystkich wartości dla figur na planszy oraz koloru aktualnego gracza. Rozwiązanie to byłoby równie złożone, gdyby nie fakt, że wartością odwrotną dla `XOR` jest ona sama. W trakcie gry nie jest konieczne ponowne obliczanie wartości haszowania dla całej planszy, a jedynie zamiana wartości dla figury dodawanej i usuwanej.

Niestety, zaimplementowany algorytm jest funkcją nieiniekcyjną. Liczba możliwych pozycji w szachach znacząco przewyższa liczbę dostępnych wartości haszowania. Z tego względu istnieje ryzyko kolizji, którego przy tym rozwiązaniu nie da się pozbyć. Jest ono natomiast na tyle małe, że zysk czasowy znacząco przewyższył możliwe komplikacje z nim związane. W celu otrzymania większej powtarzalności wyników zdecydowano się na zastosowanie tego samego ziarna (ang. *seed*) dla generatora liczb pseudolosowych. Przeprowadzono testy jednostkowe, gdzie dla różnych pozycji na planszy przechodzono przez drzewo gry, a następnie sprawdzano, czy wartości haszowania powtórzą się dla różnych pozycji. Kolizji nie zaobserwowano ani razu. Powyższe rozwiązanie zostało także wykorzystane w dalszych etapach pracy przy implementacji tabeli transpozycji.

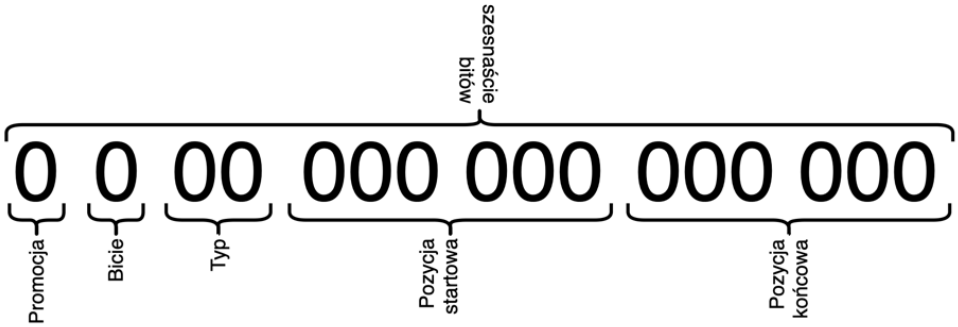
csquotes

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 podwójnym ruchem pionka, czy może jest biciem klasycznym, bądź en-passant. Okazało to się szczególnie przydatne przy sortowaniu ruchów.

Rysunek wzorowany na modelu z pracy S. Vrzina [21].



Rys. 2.2: Kodowanie ruchu szachowego

2.3. Generowanie ruchów

Generowanie możliwych ruchów w danej pozycji jest jednym z kluczowych elementów każdego silnika szachowego. Jego efektywna implementacja to taka, w której silnik spędza jak najmniej czasu, pozostawiając możliwości obliczeniowe na przeszukiwanie drzewa decyzyjnego i obliczanie oceny heurystycznej.

Podstawowym rozróżnieniem stosowanych rozwiązań jest podział na generowanie ruchów pseudolegalnych oraz ruchów legalnych [4]. 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 przekazanie odpowiedzialności za sprawdzenie legalności ruchu do 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 ewaluację stanów cichych.

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 liczbę przez większość czasu gry.

Pion ma do dyspozycji kilka 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 oddzielnych metodach. 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 ang. *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

```
addMovesFromMask(movesMask, moveType, 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 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 na jednej prostej, a tym samym uniknięcia czasochłonnej i nieczytelnej logiki iteracyjnej. Poniższy przykład został zaczerpnięty ze źródła [2].

$o = 11010101$	$o' = 10101011$	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 liczbę 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 pionów, 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, w subiektywnym odczuciu autora, łatwiejsze.

Po wykonaniu konkretnego ruchu, w miejscu, gdzie znajduje się król, stawiane są kolejne figury 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 zakończenia programu.

Ocena poprawności metodą przeprowadzania rozgrywek z silnikiem szachowym jest rozwiązaniem czasochłonnym. Odwiedzenie dużej liczby węzłów drzewa gry, w celu sprawdzenia poprawności wykonywania ruchów, jest praktycznie niemożliwe. Błędy w ten sposób powstałe są trudne do zlokalizowania.

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 na wykorzystaniu algorytmu przeszukiwania w głąb (ang. *Depth First Search*, w skrócie 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 przyjął 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. Są one 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.

Funkcja zerowa, która zwraca MAX w przypadku mata króla przeciwnika, MIN w przypadku mata własnego króla, oraz -5000 w przypadku remisu wynikającego z trzykrotnego powtórzenia pozycji bądź reguły 50 ruchów. Celem było demotywowanie silnika do wykonywania ruchów prowadzących do remisu.

Funkcja pierwsza, która opiera się na intuicyjnym założeniu, że korzystniejsza pozycja to taka, w której gracz ma więcej bierek na planszy niż przeciwnik. Różnica między liczbą bierek

konkretnego typu dla gracza oraz jego oponenta jest przemnażana przez jej wagę. 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:

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

gdzie,

H_0 – heurystyka końca gry
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” [18].

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 w głąb drzewa gry. Z uwagi na mnogość możliwych decyzji, proces ogranicza się do eksploracji do określonej głębokości. Liściom drzewa przypisywane są obliczone wartości heurystyczne. Następnie idąc wzwyż drzewa, na kolejnych poziomach, nadawane są wartości maksymalne dla protagonisty i minimalne dla antagonisty. Ruch, który prowadzi z korzenia do wierzchołka 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: Uproszczony kod implementacji algorytmu negaMax

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

    bestMoveValue = MINIMUM;
    for(Move move : moves) {
        board.makeMove(move);
        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 oznacza automatyczną wygraną oponenta. Koniecznym było zatem zaimplementowanie mechanizmu zarządzania, który pozwoliłby na podział pozostałego czasu, na wykonanie kolejnych ruchów. Pozostały czas otrzymany przez UCI w formie `go wtime <wtime> btime <btime> winc <winc> binc <binc>` pozwolił na implementacje rozwiązania opartego na wzorze:

$$\text{Est pozostałych ruchów} = \max(40 - \text{wykonane ruchy}, 10)$$
$$\text{Est pozostałego czasu} = \text{time}_{\text{color}} + (inc_{color} * \text{Est pozostałych ruchów})$$
$$\text{Propozycja czasu} = \frac{\text{Est pozostałych ruchów}}{\text{Est pozostałego czasu}}$$

Założono średnią liczbę 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 na kolejnych głębokościach. W momencie upływu czasu na obliczenia zwracana jest wartość otrzymana z najgłębszego, w pełni ukończonego przeszukiwania.

Takie podejście mogłoby wydawać się 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 [3].

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 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 rozwiązać, były otwarcia partii rozgrywanych przez silnik. Gry szachowe zawsze rozpoczynają się z tego samego położenia, co oznacza, że system musi znaleźć się w pozycjach bezpośrednio wynikających z pierwszych posunięć. Program poświęcał na początkowe obliczania zbyt 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 [14].

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 i 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 liczbie wariantów, a także umila rozgrywkę użytkownikowi.

3.1.2. Alfa-Beta cięcie

Opis zagadnienia

Poprawne wykonanie algorytmu negaMax nie gwarantuje jego skuteczności. Wraz z pogłębianiem wyszukiwania rośnie również liczba wierzchołków, które należy odwiedzić. Uśredniając, liczba dzieci, to jest pozycji wynikających z wykonania ruchu, dla danego rodzica w szachach wynosi 35 [10]. Własność ta nazywa się współczynnikiem rozgałęzienia (ang. *branching factor*).

Do minimalizacji liczby odwiedzonych węzłów wykorzystuje się alfa-beta cięcia. Jest to przykład metody podziału i ograniczeń, a jej skuteczność w dużym stopniu zależy od kolejności przeszukiwanych ruchów. Zakładając ułożenie posunięć od najlepszego, algorytm wykona najwięcej cięć, co w efekcie skutkować będzie zmniejszeniem liczby odwiedzonych wierzchołków dla głębokości według wzoru z tabeli:

Tab. 3.1: Teoretyczny limit alfa-beta cięcia dla współczynnika rozgałęzienia $b_f = 35$

Głębokość n	$(b_f)^n$	$b_f^{\lceil \frac{n}{2} \rceil} + b_f^{\lfloor \frac{n}{2} \rfloor} - 1$
1	35	35
2	1 225	69
3	42 875	1 259
⋮	⋮	⋮
10	$\simeq 2,76 * 10^{15}$	$\simeq 1,05 * 10^{8}$

Posortowanie ruchów od najgorszych, spowoduje brak możliwych cięć, a tym samym wynik będzie identyczny, jak przy algorytmie bez implementacji alfa-beta cięcia.

Implementacja

Algorytm alfa-beta cięcia jest edycją kodu funkcji negaMax 2.2. α oraz β zostają dodane jako argumenty. W sytuacji, gdy `score` $>$ `bestMoveValue` oraz $\alpha >$ `bestMoveValue`, α zostaje zaktualizowana na `bestMoveValue`. Jeśli `score` $>= \beta$ następuje odcięcie i zwracana jest wartość `bestMoveValue`. W wywołaniach funkcji na większej głębokości $\alpha = -\beta$ oraz $\beta = -\alpha$.

Rezultat

Implementacja powyższego rozwiązania pozwoliła na zmniejszenie liczby odwiedzonych wierzchołków podczas przeszukiwania, a tym samym pozwoliła iteratywnemu pogłębianiu na osiągnięcie lepszych rezultatów. Realna wartość tego ulepszenia jest różna w zależności od aktualnej pozycji na szachownicy.

3.1.3. Ewaluacja stanów cichych

Opis zagadnienia

Silnik znacznie przyspieszył znajdowanie ruchów dla danej głębokości. Zdarzały się jednak sytuacje, że zwracane wyniki znacznie odbiegały od optymalnych. Dla przykładu mogła zdarzyć się sytuacja, że liściem w drzewie było bicie hetmanem skoczka. Silnik zwracał wynik heurystyki jako bardzo wysoki i wykonywał powyższy ruch. Program nie brał jednak pod uwagę faktu, że na głębokości o jednej większej, hetman ten mógłby zostać zbity przez wrogiego pionka.

Taką sytuację nazywa się efektem horyzontu [8]. Rozwiązaniem wykonywania nieoptymalnych ruchów, było wykonanie dodatkowego wyszukiwania po osiągnięciu głębokości końcowej. Z liścia drzewa, które poprzednio zostało oceniane heurystycznie, wyprowadzono kolejne wyszukiwanie, w którym generowane ruchy są tylko biciami. Pozwoliło to na ocenę tak zwanych stanów cichych, czyli pozycji, gdzie nie ma żadnego dostępnego ruchu, który prowadziłby do bicia.

Implementacja

Algorytm rozwiązujący efekt horyzontu w literaturze nosi nazwę Quiescence Search.

Listing 3.1: Implementacja algorytmu Quiescence Search

```
public search(depth, alpha, beta) {
    standPat = evaluator.evaluate();
    if (depth == 0) return standPat;

    if (standPat >= beta) return beta;
    if (alpha < standPat) alpha = standPat;

    moves = generator.generateCaptureMoves();
    for(Move move : moves) {
        board.makeMove(move);
        score = -search(depth-1, -beta, -alpha);
        board.unmakeMove();

        if(score >= beta) return beta;
        if(score > alpha) alpha = score;
    }
    return alpha;
}
```

Niektóre wersje tego algorytmu, oprócz ruchów prowadzących do bicia, korzystają także z roszad, czy podwójnych ruchów piona, z uwagi na ich szczególny charakter.

Rezultat

W efekcie implementacji algorytmu osiągnięto silnik, który jest odporny na efekt horyzontu. Z reguły, dodatkowe wyszukiwanie prowadzi do odwiedzenia większej liczby węzłów niż w przypadku zwykłego wyszukiwania. Co warte zaznaczenia, w niektórych przypadkach, chociażby wskazanych w dodatku C, algorytm Quiescence Search w połączeniu z alfa beta cięciem skutkował zmniejszeniem liczby odwiedzonych węzłów. Ostatecznie, czy korzystniejszym jest wyszukiwanie dla większych głębokości, czy korzystanie wyłącznie ze stanów cichych, zależy od konkretnej sytuacji na planszy.

3.1.4. Statyczne sortowanie ruchów

Opis zagadnienia

W czasie testów zauważono, że podczas przeszukiwania drzewa nie następuje taka ilość alfa-beta cięć, która pozwalałaby na znaczne przyspieszenie obliczeń. Rozpoczynanie przeszukiwania węzłów gry od najlepszych posunięć pozwoliłoby na osiągnięcie teoretycznego limitu wskazanego w tabeli 3.1. W żaden sposób nie ma jednak możliwości wskazania, który z dostępnych ruchów prowadzi do najlepszej pozycji, przed wykonaniem tych ruchów. Istnieją jednak posunięcia, które można rozważyć w pierwszej kolejności, ze względu na ich szczególny charakter, a tym samym przyspieszyć cięcie. Jeśli sortowanie jest niezależne od wcześniejszych obliczeń, to jest to sortowanie statyczne.

Implementacja

Sortowanie wykonano dzięki implementacji interfejsu `Comparator` w klasie `Move`. W celu porównania pomiędzy sobą dwóch dowolnych ruchów wykorzystano następujące założenia:

- Ruch, który umożliwia bicie, jest prawdopodobnie lepszy od ruchów, które nie umożliwiają bicia.
- Wszystkie ruchy specjalne, takie jak roszady, podwójne ruchy pionem czy promocje, są lepsze od ruchów zwykłych.
- Promocje do wieży i gońca należy rozważyć na końcu, gdyż są mniej wartościowe od promocji do hetmana, jednocześnie dając dostęp do tych samych ruchów.
- Zastosowano technikę Najbardziej wartościowa ofiara – Najmniej wartościowy atakujący (ang. *Most Valuable Victim – Least Valuable Attacker*, w skrócie MVV-LVA). Sortuje ona bicia zakładając, że korzystniejsze są posunięcia, w których różnica pomiędzy wartością reprezentowaną przez bierkę bijącą a bierką zbitą, jest jak największa.

Rezultat

Dzięki zastosowaniu sortowania statycznego, udało się zwiększyć ilość alfa-beta cięć, co skutkowało możliwością przeszukiwania drzew gry do większych głębokości. Szczególnie zauważalne było to ulepszenie w procesie wyszukiwania stanów cichych, gdzie każdy z dostępnych ruchów można było zakwalifikować za pomocą MVV–LVA. Dodatek C zawiera przykładowe pozycje, w których porównano wyniki perft dla wersji klasycznej, z alfa-beta cięciem, oraz z sortowaniem ruchów.

3.1.5. Tabela transpozycji

Opis zagadnienia

Silnik szachowy wielokrotnie znajduje się w tej samej pozycji planszy, musząc wyliczać wartości heurystyczne od nowa. Dzieje się tak, zarówno w przypadku dotarcia do pozycji w wyniku odmiennych sekwencji ruchów, jak i w wyniku szukania kolejnego ruchu po ruchu przeciwnika. Rozwiązaniem tego problemu jest zastosowanie tabeli transpozycji, która przechowuje wyniki obliczeń dla pozycji, mając jednocześnie na uwadze głębokość, dla której wynik został obliczony. W sytuacji, gdy silnik ponownie napotka na tę samą pozycję, w pierwszej kolejności sprawdzi, czy w tabeli nie znajduje się już obliczony wynik dla tej samej, bądź większej głębokości.

Implementacja

Do implementacji tabeli transpozycji zastosowano strukturę `LinkedHashMap`, dla której kluczami są wartości `Zobrist Hasz` planszy, a wartościami struktura zawierająca wynik obliczeń dla danej pozycji oraz głębokość. W trakcie działania programu silnik przechodzi przez miliony możliwych pozycji, natomiast możliwości przechowywania wyników są ograniczone. W przypadku przekroczenia z góry ustalonego limitu wpisów usuwane są te wartości, które zostały użyte najdawniej. Pozwala to na pozbycie się poddrzew, w których silnik nie będzie miał okazji się ponownie znaleźć, przy jednoczesnym zachowaniu prostoty implementacji.

W algorytmie negaMax z zaimplementowanym alfa-beta cięciem, istotna jest nie tylko informacja o dokładnej wartości pozycji, ale także o górnej i dolnej granicy tej wartości. Taka wersja została zaimplementowana w kodzie, wzorując się na pracy Dennisa Breukera [1].

Rezultat

Po zastosowaniu ulepszenia zaobserwowano zmniejszenie liczby odwiedzonych węzłów w trakcie przeszukiwania drzewa gry. Było to szczególnie widoczne przy mniejszych głębokościach, gdzie w dużej mierze pozycje zostały już wyliczone w trakcie poprzednich posunięć.

3.1.6. Okno estymacji

Opis zagadnienia

Poprzednie wersje silnika rozpoczynając przeszukiwanie węzłów gry, dla każdego poziomu głębokości, zakładały wartości alfa i beta jako odpowiednio $-\infty$ i $+\infty$. Można jednak przypuszczać, że wartości zwrócone dla większych głębokości, nie będą w bardzo dużym stopniu odbiegać, od wartości zwróconych dla głębokości mniejszych. Rozpoczynając wyszukiwanie algorytmem miniMax z wartościami $alpha = wz_{n-1} - \delta$ oraz $beta = wz_{n-1} + \delta$, gdzie wz_{n-1} to wartość zwrócona dla poprzedniej głębokości, a δ to okno estymacji, istnieje szansa wystąpienia większej ilości alfa-cięć oraz beta-cięć. Pomimo oczywistych korzyści wynikających z zastosowania powyższej techniki, istnieje także ryzyko, że w przypadku, gdy wartość wykracza poza okno estymacji, konieczne jest rozpoczęcie ponownego wykonania algorytm wyszukiwania. Ponowne uruchomienie odbywa się z szerszym oknem estymacji.

Implementacja

Kluczowe w implementacji algorytm było odpowiednie ustalenie wartości δ. Zbyt wąskie okno skutkowałoby zbyt częstymi ponownymi przeszukiwaniami drzewa. Zbyt szerokie natomiast, prowadziłoby do braku efektów zastosowanego usprawnienia. Większość silników ustawia wartość δ na $1/3$ wartości pionka [8]. Taką też wartość zastosowano w programie. W przypadku wyjścia poza zakres, okno estymacji poszerzone zostaje do $-\infty$ i $+\infty$, co w efekcie gwarantuje poprawne działanie.

Dodatkowym utrudnieniem, stał się fakt, że przejście algorytmu, którego efektem będzie zwrócenie wartości spoza okna estymacji, wywoływało zapis nieprawdziwych wartości do tabeli transpozycji. Przy próbie jednoczesnego zastosowania obu rozwiązań, należałoby utworzyć bufor, który zapisywałby wartości z danego przeszukania jedynie w przypadku, gdy wartość końcowa mieściła się w oknie estymacji.

Rezultat

W programie zdecydowano się nie wykorzystywać jednocześnie dwóch rozwiązań, to jest tabeli transpozycji i okna estymacji. Porównano rezultaty oraz wybrano to ulepszenie, które przyczyniło się do największego wzrostu siły silnika. Przeprowadzono także testy, które miały na celu sprawdzenie liczby wystąpień wyjścia poza zakres okna estymacji. Spośród przetestowanych opcji wybrano tę wartość δ, która pozwalała na istotne polepszenie alfa-beta cięć, przy jednoczesnej próbie zachowania małej liczby powtórzeń.

3.1.7. Rozszerzanie wyszukiwania

Opis zagadnienia

Rozszerzenie wyszukiwania to technika, która zakłada, że niektóre z dostępnych posunięć wymagają dodatkowego zbadania. W przypadku, gdy taki ruch nastąpi, algorytm przeszukiwania przedłuży przeszukiwanie poddrzewa gry o jeden poziom. W odróżnieniu od Quiescence Search rozszerzenie wyszukiwania przedłuża się na wszystkie z dostępnych ruchów z pozycji, a nie tylko na te, które prowadzą do bicia.

Implementacja

Zdecydowano się na implementację dwóch rodzajów przedłużeń:

- Jeśli ruch jest atakiem powodującym wystąpienie szacha, to przeszukiwanie zostaje przedłużone o jeden poziom.
- W przypadku, gdy istnieje jedynie jedno dostępne posunięcie, również przeszukiwanie zostaje przedłużone o jeden poziom.

W celu uniknięcia zbytniego rozgałęziania się drzewa, przedłużenie wyszukiwania może nastąpić w danym poddrzewie gry tylko raz.

Rezultat

W toku gry, na skutek rozszerzenia wyszukiwania, nie zaobserwowano znacznego wzrostu bądź spadku siły silnika.

3.2. Ulepszenia dla oceny heurystycznej

3.2.1. Tablice figur

Opis zagadnienia

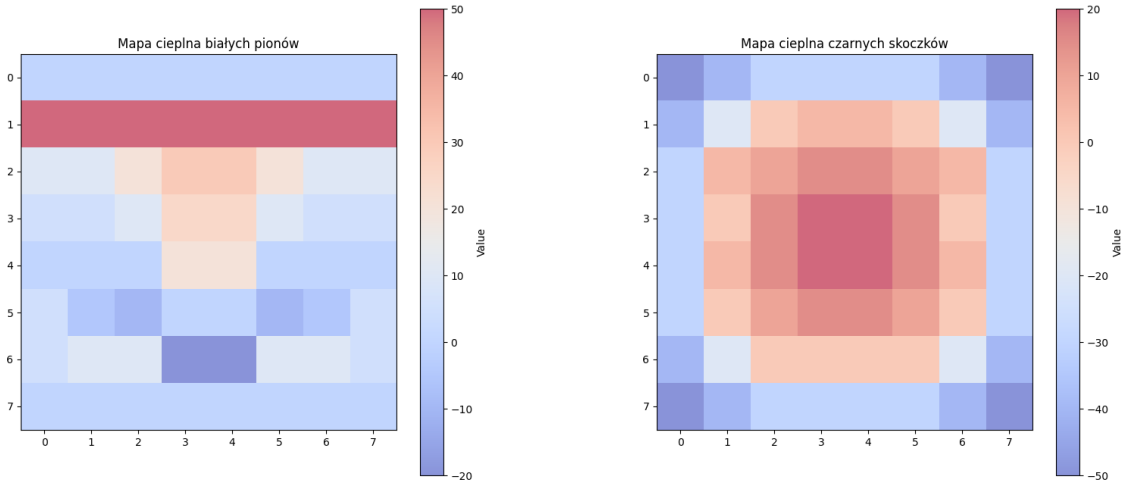
Istotnym z punktu widzenia heurystyki silnika jest prawidłowa ocena informacji na planszy. Ograniczona wiedza, jedynie co do liczby figur na planszy, była niewystarczająca, aby móc prowadzić rozgrywkę na odpowiednio wysokim poziomie.

Wraz z rozwojem strategii szachowych gracze wypracowali szereg schematów, które ułatwiają im podejmowanie decyzji, niezależnie od konkretnej sytuacji na planszy. Są nimi między innymi: zajęcie centralnych pól przez pionki, nierozwijanie skoczka na skrajne pola planszy, czy pozycjonowanie wież na otwartych liniach i wierszach pionów przeciwnika.

Implementacja

Implementacja każdego z powyższych, jak i wielu innych reguł mogłaby okazać się czasochłonna i złożona obliczeniowo. Zdecydowano się zatem na zastosowanie tablic figur, które można rozumieć jako mapy cieplne, reprezentujące korzystność umieszczenia figury na danym polu. Dla każdego typu bierki $(2 * 6)$ stworzono tablicę 64 elementową określającą wartości, odnośnie do tego, jak korzystne jest umieszczenie figury na danym polu. Zdecydowano się na nie tworzenie własnych tablic, a zastosowanie gotowych, dostępnych na forum tworzenia silników szachowych [5].

a) b)



Rys. 3.1: Przykładowe tablice cieplne bierek: a) białych pionów, b) czarnych skoczków

Rezultatem zastosowania tablic figur stał się silnik, który w ocenie autora znacznie lepiej „rozumiał” pozycję na planszy.

3.2.2. Ochrona króla

W celu zapobieżenia sytuacjom, w których król jest zagrożony, zaimplementowano mechanizm, który sprawdza wszystkie bierki w najbliższej okolicy króla. Pionki i figury tego samego koloru są traktowane korzystnie, przeciwnego natomiast bardzo niekorzystnie. Istotny jest również

typ figury, która stanowi zagrożenie. Miało to w efekcie ograniczyć ryzyko zakładania mata, szczególnie na środkowym etapie gry. Jest to ulepszenie, którego skuteczność jest trudna do zweryfikowania w czasie gry, z uwagi na fakt, że jego efekt jest widoczny dopiero w sytuacji zagrożenia króla.

3.2.3. Struktura pionów

Opis zagadnienia

Po implementacji ulepszenia 3.2.1 silnik uzyskał zdolność oceny pozycyjnej. Problemem było natomiast, że bierki oceniane były pojedynczo, to jest bez uwzględnienia ich wzajemnych relacji, oraz relacji w stosunku do bierek przeciwnika.

Termin „struktura pionów" jest szeroko znany w literaturze szachowej [6] i odnosi się do technik mających na celu skoordynowaną obronę własnej części szachownicy oraz przeprowadzania ataków z wykorzystaniem pionów. Skuteczne pozycjonowanie pionków na planszy może prowadzić do uzyskania przewagi w grze.

Implementacja

Zasady rozumiane przez graczy szachowych w sposób intuicyjny należało przekształcić na zbiór reguł możliwych do zrozumienia przez program, a prowadzących do skutecznej oceny pozycji.

- Pionki, które posiadają z tyłu po swojej prawej lub lewej stronie innego pionka, są uważane za dobrze chronione.
- Pionki nieposiadające za sobą innych pionków na sąsiednich kolumnach są traktowane jako izolowane i trudne do obrony.
- Dwa pionki znajdujące się w jednej linii pionowej (tzw. „zdublowane") są uważane za małowartościowe, gdyż odsłaniają inne linie na potencjalne ataki.
- Pionki, dla których istnieje możliwość promocji, spowodowana brakiem pionów przeciwnika na sąsiednich kolumnach, są uznawane za bardzo wartościowe.

Powyższe techniki zostały w dużej mierze zaimplementowane z wykorzystaniem uprzednio zainicjowanych tablic bitowych bierek i operacjach na nich. Odbyło się to w sposób analogiczny do generowania dostępnych ruchów.

Rezultat

Rezultatem stał się silnik, który w sposób bardziej złożony oraz koherentny był w stanie ocenić swoją sytuację na planszy. Istniało natomiast ryzyko, że czas konieczny na przeprowadzanie obliczeń znacząco przewyższy oferowane korzyści.

3.2.4. Moment gry

Opis zagadnienia

Szachy to gra dynamiczna, w której techniki stosowane przez szachistów w dużym stopniu zależą od pozycji na planszy. Jednym z podstawowych wskaźników wpływających na obierane strategie jest moment gry. Partia szachowa dzieli się na trzy etapy: **otwarcie** – najczęściej trwające do 10 ruchów, rozgrywane z zapamiętanej teorii szachowej, **grę środkową** – najdłuższy etap, w którym obie strony walczą o kontrolę nad planszą, oraz **grę końcową** – etap, w którym liczba bierek po obu stronach jest niska, a gracze próbują znaleźć techniki zamatowania przeciwnika.

Implementacja

Pierwszy z nich zaimplementowany został dzięki Bibliotece Otwarć 3.1.1. W celu odróżnienia gry środkowej od końcowej zastosowano kryterium, mówiące o pozostaniu po każdej stronie maksymalnie jednej figury wysokiej (hetmana, wieży) bądź dwóch figur niskich (gońca, skoczka). W przypadku spełnienia tego warunku silnik przechodzi do fazy końcowej, cechującej się odmienną oceną.

W grze końcowej między innymi zmienia się tablica dla figury króla. W fazie środkowej król zazwyczaj unikał walki bezpiecznie schroniony po roszadzie w rogu planszy. Natomiast w fazie końcowej, z uwagi na mniejszą liczbę bierek jego możliwości ofensywne zyskują na znaczeniu. Powinien przesunąć się do centrum planszy, pomagając w ochronie pionków oraz znajdowaniu matów na przeciwniku. Co więcej, na znaczeniu zyskały same pionki, gdzie ich promocja na hetmana może przesądzić o losie partii.

Rezultat

Zastosowane ulepszenie przyniosło zakładane rezultaty. W połączeniu ze zwiększeniem głębokości wyszukiwania, wynikającej z mniejszej liczby bierek, a co za tym idzie niższym współczynnikiem rozgałęzienia drzewa, doprowadziło do efektywnej gry końcowej. Wyszukiwanie ruchów było na tyle dobre, że podjęto decyzję o zaniechaniu implementacji siatek matowych, gdyż silnik radził sobie bardzo dobrze ze znajdowaniem rozwiązań.

Rozdział 4

Ocena siły silnika

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

4.1. Porównanie wersji silnika

W pierwszej kolejności porównano wersje silnika z odmiennymi poziomami usprawnień. Pozwoliło to na potwierdzenie poprawności zaimplementowanych rozwiązań oraz na wyłonienie najsilniejszej wersji programu.

Najbardziej oczywistą, a zarazem najłatwiejszą do wykonania metodą porównawczą było rozegranie określonej liczby pojedynków pomiędzy oponentami, a następnie sprawdzenie współczynnika wygranych. Do przeprowadzenia testów wykorzystano program komputerowy Cutechess [7]. Posłużył on nie tylko jako szachowy interfejs graficzny, ale także jako zarządca turniejowy dla silników implementujących protokół UCI.

Pomiędzy wybranymi parami silników rozegrano po 25 gier, każda trwająca 1 minutę na partię plus $0,6$ sekundy na ruch. Aby zmniejszyć powtarzalność, gry rozpoczynały się nie od pozycji startowej, ale od losowych otwarć szachowych z biblioteki [17]. Pozycje te były wyrównane, aby uniknąć przewagi początkowej któregokolwiek z silników.

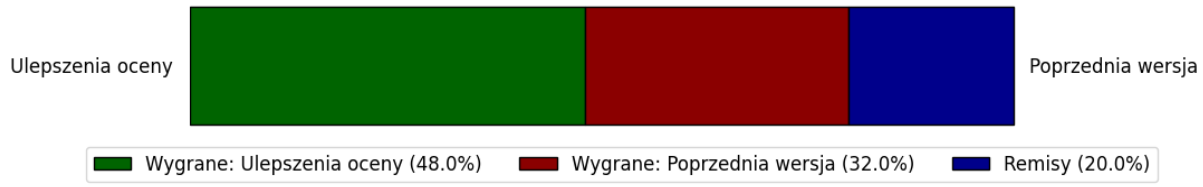
4.1.1. Wybrane wyniki

Tablica figur

Rys. 4.1: Wyniki rozgrywek z tablicą figur

Silnik z zaimplementowaną tablicą figur uzyskał znaczną przewagę nad swoim rywalem. Nie tylko nie przegrał ani jednej partii, ale wygrał aż 80% z nich. Świadczy to o zwiększeniu dokładności oceny heurystycznej pozycji, co przełożyło się na możliwość podejmowania lepszych decyzji w trakcie gry.

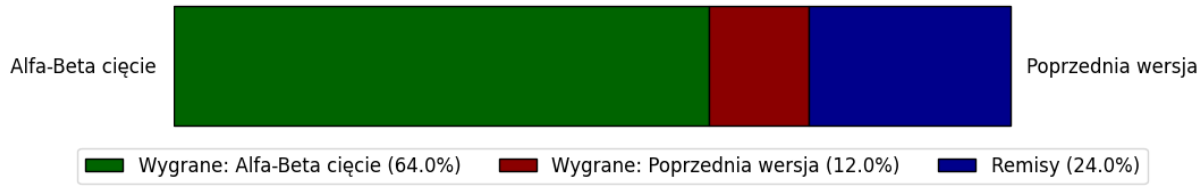
Struktura pionów i bezpieczeństwo króla



Rys. 4.2: Wyniki rozgrywek ze strukturą pionów i bezpieczeństwem króla

Poprzednią wersję implementującą tablicę figur porównano z wersją, która dodatkowo posiadała ulepszenia w postaci struktury pionów oraz oceny bezpieczeństwa króla. Wyniki były bardziej zbliżone niż w poprzednim pojedynku, co sugeruje ich mniejszy wpływ na siłę silnika. Niemniej, były one na tyle znaczące, aby zagwarantować wygraną na poziomie 48%. Z analizy pojedynczych partii wynikało, że silnik z ulepszeniami zyskiwał przewagę już w pierwszych ruchach gry, co pozwalało mu na kontrolę nad planszą w późniejszych etapach.

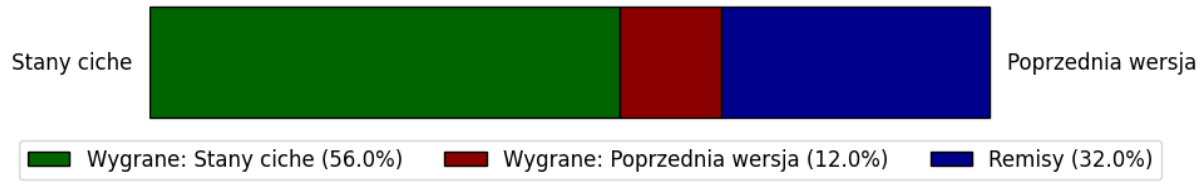
Alfa-Beta cięcie



Rys. 4.3: Wyniki rozgrywek z alfa beta cięciem

Pierwsze sprawdzone ulepszenie co do algorytmów przeszukiwania drzewa gry przyniosło oczekiwane rezultaty. Zmniejszenie liczby odwiedzonych węzłów pozwoliło na zwiększenie głębokości przeszukiwań, a co za tym idzie, pozwoliło silnikowi na lepsze ocenianie pozycji. Wynik 64% wygranych oraz 24% remisów utwierdza w przekonaniu o poprawności i skuteczności implementacji algorytmu alfa-beta.

Ewaluacja stanów cichych



Rys. 4.4: Wyniki rozgrywek z ewaluacją stanów cichych

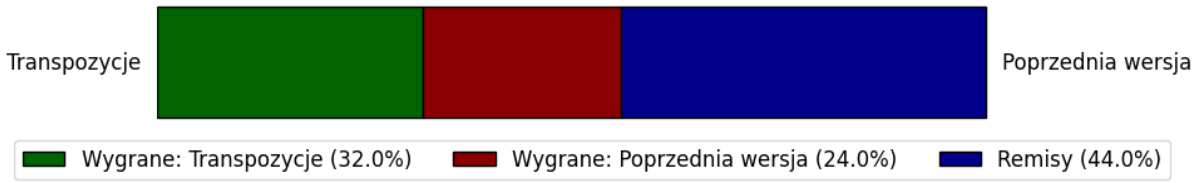
Wykonanie testów dla tego ulepszenia było szczególne istotne, z tego względu, że dodatkowa ewaluacja stanów cichych ma także tendencje do zwiększenia liczby odwiedzonych węzłów drzewa. Należało potwierdzić, że zysk związany z uniknięciem efektu horyzontu przewyższa koszty związane z dodatkowymi obliczeniami. Jak widać na rysunku 4.4, silnik z ulepszeniem wygrał 56% i zremisował 32% partii, co sugeruje, że dodatkowe obliczenia były warte podjęcia.

Statyczne sortowanie ruchów

Rys. 4.5: Wyniki rozgrywek ze statycznym sortowaniem ruchów

Wyniki pojedynku pomiędzy poprzednią wersją silnika a wersją z zaimplementowanym statycznym sortowaniem ruchów były zaskakująco korzystne. Silnik z ulepszeniem wygrał 68% partii, przy jednoczesnym braku jakiejkolwiek porażki. Jeszcze raz potwierdza to znaczenie ułożenia odwiedzanych wierzchołków dla algorytmu alfa-beta.

Tabela transpozycji



Rys. 4.6: Wyniki rozgrywek z tabelą transpozycji

Otrzymane rezultaty dla ulepszenia związanego z tabelą transpozycji wskazywały na błąd w implementacji. Nowy silnik remisował większość pojedynków. Co zaskakujące, analiza poszczególnych rozgrywek wskazywała, że silnik z ulepszeniem w większości zremisowanych gier obejmował zauważalne prowadzenie. W opinii autora jednym z możliwych powodów mógło być nierozróżnianie oceny jednej pozycji planszy, od drugiej, takiej samej, ale prowadzącej do niekorzystnego remisu. Dla przykładu ocena heurystyczna trzykrotnie powtarzającej się pozycji powinna być różna. W efekcie silnik zapętlał się w pozycji z jego perspektywy korzystnej, prowadząc do remisu. Poprawa implementacji wymagałaby stworzenia oddzielnego haszowania rozróżniającego te dwie, na pozór identyczne, pozycje.

4.1.2. Podsumowanie

Powyżej przedstawiono te z wyników, które miały największy wpływ na siłę silnika, bądź przedstawiały ciekawe zależności. Biblioteka otwarć, choć istotna, nie została porównana, gdyż rozgrywki rozpoczynały się od wyrównanych pozycji w grze środkowej, a więc biblioteka otwarć nie miałaby wpływu na wynik.

W pozostałych ulepszeniach wynik był znacznie bardziej wyrównany, zahaczający o błąd statystyczny. Aby przeprowadzić analizę tych usprawnień, należałoby przeprowadzić więcej rozgrywek pomiędzy silnikami, bądź skorzystać z bardziej wyrafinowanych metod statystycznych. Jedną z takich technik, powszechnie stosowanych w silnikach szachowych, jest Sekwencyjny Test Probabilistyczny (ang. *Sequential Probability Ratio Test*, SPRT). Pozwala on na zwiększenie wiarygodności wyników, przy jednoczesnym zredukowaniu do minimum liczby przeprowadzanych rozgrywek [19]. Choć program Cutechess posiada również możliwość przeprowadzenia SPRT, wykraczały one poza zakres niniejszej pracy.

4.2. Porównanie z innymi silnikami

Przedstawione powyżej rozwiązania są w stanie z dużą dozą pewności wskazać najsilniejszą wersję silnika szachowego. Są to jednak wyniki subiektywne, odnoszące się jedynie do samego programu. W celu obliczenia relatywnej siły silnika konieczne jest przetestowanie zaimplementowanych rozwiązań przeciwko innym programom szachowym.

Lichess jest darmową platformą internetową, umożliwiającą prowadzenie rozgrywek online. Jedną z jej funkcji, jest możliwość wykorzystania udostępnionego przez dewelopera API, w celu podłączenia konta do lokalnie uruchomionego programu z silnikiem wykonującym ruchy.

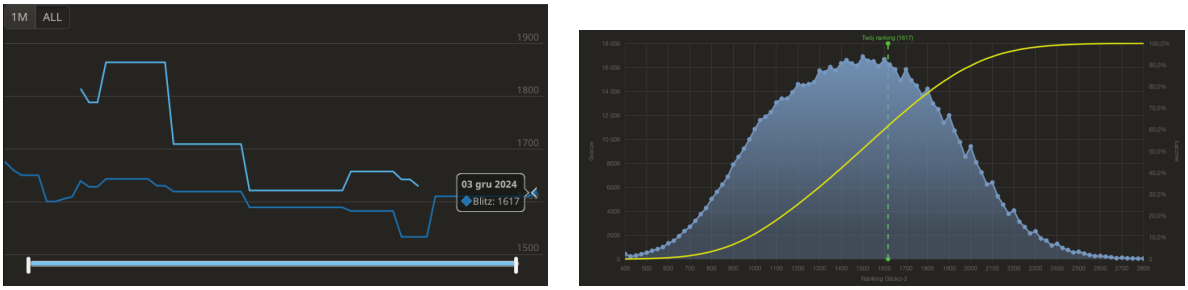
Jako pośrednika pomiędzy platformą a silnikiem, wykorzystano program lichess-bot [16]. Umożliwił on komunikację z serwerem Lichess, automatyczne dobieranie przeciwników oraz przekazywanie ruchów.

Do testów wybrano rozgrywki typu `bullet` (poniżej trzech minut na gracza) oraz `blitz` (powyżej trzech, a poniżej dziesięciu minut na gracza), które charakteryzują się krótkim czasem na partię.

W trakcie działania programu przeprowadzono 210 gier. System lichess-bot został skonfigurowany w taki sposób, aby akceptować wyzwania jedynie dla gier typu `bullet` oraz `blitz`, oraz dla oponentów z rankingiem odstającym o ± 300 od rankingu silnika.

W momencie zakończenia prac nad silnikiem osiągnął on ranking 1629 ELO dla gier typu `bullet` oraz 1617 ELO dla gier typu `blitz`.

a) b)



Rys. 4.7: Platforma lichess: a) ranking na platformie, b) rozkład rankingów

Platforma Lichess zapewnia również dostęp do narzędzi umożliwiających bardziej szczegółową analizę gier. Do ich stworzenia wykorzystywany jest jeden z najsilniejszych silników szachowych – Stockfish. Poniżej przedstawiono niektóre z dostępnych statystyk dla gier `blitz`, wszystkie są wartościami średnimi:

1. **Dokładność gry** – 84.6%
2. **Ruchów na grę** – 43.41
3. **Gier wygranych** – 41.1%
4. **Czas ruchu** – 5.65 sekundy
5. **Gier zremisowanych** – 16.9%
6. **Ranking przeciwnika** – 1 625.85ELO
7. **Strata względem ruchu optymalnego** – 45.29 ACPL

Średnia Strata w Centypionach (ang. *Average Centipawn Loss*, w skrócie ACPL) jest miarą oceniającą jakość ruchów gracza. Wartość ta jest obliczana jako średnia różnica (w setnych częściach wagi piona) pomiędzy ruchem wykonanym przez gracza a ruchem optymalnym [15]. ACPL na poziomie 0 oznaczałaby grę idealną, ACPL na poziomie 100 oznaczaby średnią stratę na ruch o wartości jednego pionka. Dla człowieka wartość ACPL równa 20 jest grą niemal idealną, na poziomie mistrzowskim [22].

Rozdział 5

Zakończenie

5.1. Możliwości dalszego rozwoju silnika

Jak pokazano w rozdziale 4, stworzony program reprezentuje wysoki poziom gry oraz może stanowić wyzwanie dla niejednego gracza. Silnik powstał w taki sposób, aby umożliwić łatwą implementację dodatkowych usprawnień, które w dalszym stopniu mogłyby zwiększyć jego siłę. Można zdefiniować kilka ścieżek dalszego rozwoju oprogramowania:

Pierwszym z nich jest poprawa funkcji haszującej wykorzystywanej w tabeli transpozycji. Jej poprawa skutkowałaby zmniejszoną liczbą remisów, a co za tym idzie zwiększeniem współczynnika wygranych. Następnie należałoby zintegrować poprawioną wersję algorytmu z oknem estymacji.

Kolejnym z aspektów wartych poprawy jest zarządzanie czasem gry. W obecnej konfiguracji silnik oblicza czas ruchu na podstawie pozostałego czasu, nie biorąc pod uwagę takich aspektów, jak przewaga nad przeciwnikiem, czy też pozycja na planszy. Niektóre ruchy, które wydają się oczywiste, gdyż znacznie poprawiają pozycję, mogłyby być wykonywane szybciej, pozostawiając cenny czas na obliczenia w bardziej złożonych, czy równiejszych sytuacjach. Implementacja polegałaby na obserwacji zmiany proponowanych ruchów oraz oceny heurystycznej dla różnych głębokości, a następnie na tej podstawie decydowała o szybszym zakończeniu przeszukiwania.

Kolejnym polem do rozwoju jest sortowanie ruchów. Silnik przed wykonaniem ruchu stosuje statyczne sortowanie ruchu, tak jak opisano w rozdziale 3.1.4. Możnaby przypuszczać, że istotnym ulepszeniem byłoby zastosowanie dynamicznego sortowania ruchów, czyli takiego, w którym ułożenie ruchów zależy od wyników wyszukiwań na niższych głębokościach. Dla przykładu możnaby rozpocząć od ruchów, które poprzednio doprowadziły do jak największego alfa beta cięcia.

Naturalnym przedłużeniem prac nad silnikiem szachowych byłoby zaimplementowanie algorytmów z dziedziny sztucznej inteligencji i uczenia maszynowego. W kategorii heurystyk oceny pozycji możnaby zastosować implementację algorytmu genetycznego, w celu dostrojenia wag poszczególnych funkcji oceny, tak aby były one zbliżone do optymalnych. Natomiast w kategorii wyszukiwania ruchów, można by zaimplementować sieć neuronową, która proponowałaby obiecujące ruchy, na podstawie wyuczonej wiedzy z zestawu partii szachowych.

Aktualna architektura systemu oraz automatyzacja procesu oceny skuteczności pozwala na eksperymentowanie z mniej oczywistymi, acz z punktu widzenia informatyki ciekawymi rozwiązaniami. Dla przykładu możliwym usprawnieniem jest komunikacja systemu z dużym modelem językowym (ang. *Large Language Model*, w skrócie LLM), który zwracałby propozycje ruchów na podstawie przekazanych mu informacji o stanie gry. Takie ruchy byłyby sprawdzane w pierwszej kolejności, przyspieszając (bądź spowalniając) alfa-beta cięcia.

5.2. Podsumowanie otrzymanych wyników

W ocenie autora, cel pracy inżynierskiej został zrealizowany. Przegląd literatury oraz zapoznanie się z wykorzystywanymi algorytmami pozwoliło na zrozumienie zasad działania silników szachowych. Zdobyta wiedza pozwoliła na implementację programu komputerowego w języku Java 22, który analizuje i ocenia pozycję na szachownicy, a następnie wykonuje legalny i optymalny ruch. Architektura programu pozwoliła na łatwą implementację dodatkowych usprawnień, które zwiększyły jego skuteczność. Przeprowadzono analizę porównawczą w celu oceny jakości zaimplementowanych rozwiązań. Wyniki testów wskazały, że silnik osiąga na platformie Lichess ranking szachowy w zakresie pomiędzy 1600 a 1650 ELO. Plasuje go tym samym wśród 38 procent najlepszych graczy na stronie. Ponadto silnik wykazał się na tyle skuteczny, że z łatwością pokonuje jego autora. Programowi nadano nazwę Kobayashi Maru [13].

Bibliografia

[1] D. M. Breuker. „Memory versus Search in Games”. Ph.D. thesis. Maastricht University, 1998. URL: https://project.dke.maastrichtuniversity.nl/games/files/phd/Breuker_thesis.pdf.

[2] *Chess Programming Wiki: Hyperbola Quintessence*. URL: https://www.chessprogramming.org/Hyperbola_Quintessence (term. wiz. 02. 12. 2024).

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

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

[5] *Chess Programming Wiki: Simplified Evaluation Function*. URL: https://www.chessprogramming.org/Simplified_Evaluation_Function (term. wiz. 05. 11. 2024).

[6] *Chess Terms: Pawn Structure*. URL: https://www.chess.com/terms/pawn-structure (term. wiz. 11. 11. 2024).

[7] *Cute Chess*. URL: https://github.com/cutechess/cutechess (term. wiz. 02. 12. 2024).

[8] W. Duch. *Chess Programming Theory. Notatki do wykładów*. URL: http://fizyka.umk.pl/~duch/Wyklady/AI/Notki/Chess%20Programming%20Theory.htm (term. wiz. 11. 11. 2024).

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

[10] M. Franaszczuk. *Chess Programming Wiki: Evaluation*. URL: http://cs.cornell.edu/boom/2004sp/ProjectArch/Chess/chessmain.html (term. wiz. 10. 10. 2024).

[11] D. Heath i D. Allum. „The Historical Development of Computer Chess and its Impact on Artificial Intelligence”. W: AAAIWS’97-04 (1997), s. 63–68.

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

[13] *Kobayashi Maru scenario*. URL: https://memory-alpha.fandom.com/wiki/Kobayashi_Maru_scenario (term. wiz. 02. 12. 2024).

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

[15] *Lichess - Najczęściej zadawane pytania*. URL: https://lichess.org/faq#acpl (term. wiz. 04. 12. 2024).

[16] *Lichess Bot - documentation*. URL: https://github.com/lichess-bot-devs/lichess-bot (term. wiz. 01. 12. 2024).

[17] official-stockfish. *Chess books used to develop Stockfish*. URL: https://github.com/official-stockfish/books/tree/master (term. wiz. 05. 11. 2024).

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

[19] *Sequential Probability Ratio Test*. URL: https://www.chessprogramming.org/Sequential_Probability_Ratio_Test (term. wiz. 01. 12. 2024).

[20] 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: http://api.semanticscholar.org/CorpusID:12908589.

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

[22] K. Wojcicki. *How many mistakes do Grandmaster chess players make?* URL: https://kwojcicki.github.io/blog/CHESS-BLUNDERS#:~:text=Average%20Centipawn%20Loss%20(ACPL)%20is,or%20purposefully%20losing%20the%20game. (term. wiz. 04. 12. 2024).

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.

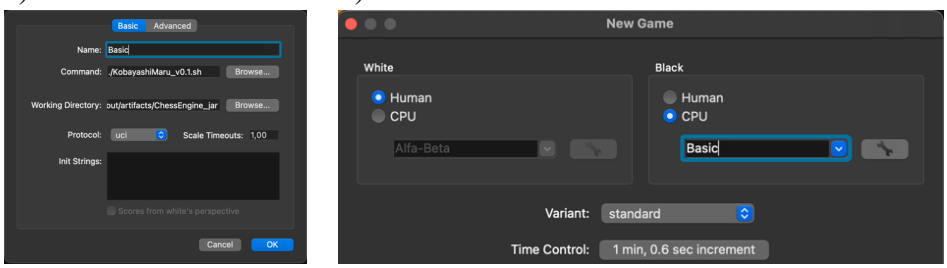
W celu implementacji tego rozwiązania została wykorzystana biblioteka lichess-bot, która stanowi pośrednik pomiędzy silnikami wykorzystującymi UCI a API platformy Lichess. [16]

A.2. Kompilacja kodu i połączenie z GUI

Aby własnoręcznie skompilować kod źródłowy silnika, należy wykonać poniższe kroki:

1. Pobrać kod źródłowy z publicznie dostępnego repozytorium `https://github.com/KaWis17/KobayashiMaru`.
2. Stworzyć plik `.jar` z kodem źródłowym, korzystając z dowolnego narzędzia do budowania projektów w języku Java.
3. Stworzyć skrypt `.sh` zawierający komendę uruchamiającą silnik, np.: `java -jar KobayashiMaru.jar`.
4. Sprawdzić poprawność przez odpalenie silnika w konsoli. Po wpisaniu na standardowe wejście `uci` silnik powinien odpowiedzieć zgodnie z protokołem.
5. Połączyć z interfejsem graficznym według instrukcji dostarczonej przez producenta oprogramowania. Dla przykładu w programie CuteChess należy wejść w preferencje -> silniki -> dodaj silnik -> wybrać skrypt `.sh` jako komendę.
6. Rozpocząć grę z silnikiem przez stworzenie nowego meczu.

a) b)



Rys. A.1: Interfejs graficzny CuteChess: a) dodawanie silnika, b) tworzenie rozgrywki

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

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

Start pos	Stockfish	Wersja podstawowa	Alfa-beta	Quiescence	Move ordering	Estimation	Transposition
1.	20	20	20	20	20	20	20
2.	400	400	186	194	214	214	214
3.	8 902	8 902	2 262	2 279	2 360	2 741	2 360
4.	197 281	197 281	20 596	23 119	20 428	23 597	17 481
5.	4 865 609	4 865 609	223 840	225 836	173 183	199 062	123 575
6.	119 060 324	119 060 324	3 349 766	1 606 833	1 019 119	1 245 427	615 267
7.	3 195 901 860	xxx	20 668 442	19 449 096	7 934 005	9 078 322	3 923 917
8.	84 998 978 956	xxx	275 274 306	183 000 753	57 778 837	70 097 202	23 360 242

Tab. C.1: Wyniki porównań dla pozycji startowej

FEN: TODO!!!

Start pos	Stockfish	Wersja podstawowa	Alfa-beta	Quiescence	Move ordering	Estimation	Transposition
1.	20	20	20	20	20	20	20
2.	400	400	186	194	214	214	214
3.	8 902	8 902	2 262	2 279	2 360	2 741	2 360
4.	197 281	197 281	20 596	23 119	20 428	23 597	17 481
5.	4 865 609	4 865 609	223 840	225 836	173 183	199 062	123 575
6.	119 060 324	119 060 324	3 349 766	1 606 833	1 019 119	1 245 427	615 267
7.	3 195 901 860	xxx	20 668 442	19 449 096	7 934 005	9 078 322	3 923 917
8.	84 998 978 956	xxx	275 274 306	183 000 753	57 778 837	70 097 202	23 360 242

Tab. C.2: Wyniki porównań dla pozycji w grze środkowej

FEN: TODO!!!

Start pos	Stockfish	Wersja podstawowa	Alfa-beta	Quiescence	Move ordering	Estimation	Transposition
1.	20	20	20	20	20	20	20
2.	400	400	186	194	214	214	214
3.	8 902	8 902	2 262	2 279	2 360	2 741	2 360
4.	197 281	197 281	20 596	23 119	20 428	23 597	17 481
5.	4 865 609	4 865 609	223 840	225 836	173 183	199 062	123 575
6.	119 060 324	119 060 324	3 349 766	1 606 833	1 019 119	1 245 427	615 267
7.	3 195 901 860	xxx	20 668 442	19 449 096	7 934 005	9 078 322	3 923 917
8.	84 998 978 956	xxx	275 274 306	183 000 753	57 778 837	70 097 202	23 360 242

Tab. C.3: Wyniki porównań dla pozycji w grze końcowej