Politechnika Wrocławska
Wydział Elektroniki, Fotoniki i Mikrosystemów

KIERUNEK: Automatyka i robotyka

PRACA DYPLOMOWA INŻYNIERSKA

Tytuł pracy: Aplikacja do gry w szachy z wykorzystaniem klasycznego silnika szachowego oraz głębokich sieci neuronowych

AUTOR:
Jakub Jakubczak

PROMOTOR:
prof. dr hab. inż.
Artur Wymysłowski

WROCŁAW 2024

Spis treści

1. Wstęp

1.1. Cel i zakres wykonanej pracy

Celem pracy było zaprojektowanie i wykonanie aplikacji do gry w szachy, z wykorzystaniem języka programowania Python [1], biblioteki Tkinter [2] oraz sieci neuronowych, która pozwalałaby na grę z komputerem. W ramach wstępnie przyjętych założeń aplikacja miała zawierać:

- przyjazny dla użytkownika interfejs graficzny, tzw. GUI (z ang. Graphical User Interface) pozwalający zarówno na grę samodzielną, jak i na rywalizację z komputerem; ponadto, aplikacja miała zawierać menu umożliwiające wybór różnych opcji gry, takich jak rozpoczęcie nowej gry, wybór silnika szachowego oraz dostosowanie ustawień,
- reprezentację stanu gry prezentowaną w postaci graficznej, za pomocą szachownicy oraz bierek,
- logikę gry w szachy, której działanie jest zgodne z zasadami gry, zintegrowane z interfejsem graficznym oraz sztuczną inteligencją, np. silnik szachowy wykorzystujący algorytm Negamax oraz funkcje do oceny pozycji. Wstępnie planowano wykorzystanie głębokich sieci neuronowych do oceny pozycji, jednak z powodu ograniczeń czasowych oraz złożoności implementacji nie zostało to wykonane.

W ramach części dotyczącej logiki i zasad gry w szachy zaimplementowano:

- obsługę zasad ruchu figur,
- generowanie wszystkich możliwych ruchów,
- wykonywanie ruchów na szachownicy.

Stan gry jest prezentowany graficznie na interaktywnej szachownicy, na której figury można przesuwać za pomocą myszki. Dodatkowo, w menu gry wyświetlane są następujące informacje:

- liczba punktów,
- lista zbitych figur,
- ocena pozycji,
- historia gry w formacie LAN(z ang. Long Algebraic Notation).

Interfejs graficzny jest wyposażony w przyciski funkcjonalne:

- „Poddaj się", który umożliwia poddanie się.
- „Wróć do menu", który umożliwia użytkownikowi odpowiednio zakończenie gry i powrót do głównego ekranu aplikacji.

W ramach zrealizowanej pracy został zaprojektowany i zaimplementowany klasyczny silnik szachowy wykorzystujący zmodyfikowany algorytm Negamax. Analiza stanu gry oparta jest m.in. na ocenie stanu materialnego obu stron, który odnosi się do łącznej wartości bierek posiadanych przez obu graczy. Każda bierka ma przypisaną wartość liczbową, która odzwierciedla jej potencjalny wpływ na rozgrywkę. Wartości liczbowe bierek zostały przedstawione w podrozdziale 1.3.4. W ocenie stanu gry również są brane pod uwagę dodatkowe parametry pozycyjne. Algorytm Negamax został zoptymalizowany za pomocą cięć alfa-beta oraz różnych heurystyk, a analiza wyników

pozwala ocenić wpływ tych modyfikacji na wydajność oraz jakość gry silnika przy ustalonej głębokości przeszukiwania. Celem tych działań jest znalezienie najlepszej konfiguracji heurystyk oraz parametrów ewaluacji, które pozwalają silnikowi grać lepiej.

W przyszłości planuje się dodatkowo uzupełnienie aplikacji o implementację sieci neuronowej NNUE (z ang. Efficiently Updatable Neural Network) [3] do ewaluacji pozycji szachowych. Jej wydajność i jakość zostaną porównane z wynikami uzyskanymi za pomocą klasycznej metody oceny pozycji. Wyniki analizy umożliwią ocenę zalet i wad obu podejść w kontekście projektowania współczesnych silników szachowych.

1.2. Motywacja do podjęcia tematu

W założeniach praca inżynierska miała połączyć moje zainteresowania programowaniem komputerowym, grą w szachy oraz uczeniem maszynowym/sztuczną inteligencją. Tematy te są dla mnie niezwykle fascynujące, zwłaszcza w kontekście rozwoju współczesnych silników szachowych, które osiągnęły poziom daleko przewyższający możliwości graczy. Moim celem było zrozumienie algorytmów oraz technik, które pozwoliły na osiągnięcie takiego poziomu gry przez algorytmy uczenia maszynowego.

Współczesne silniki szachowe są projektowane z wykorzystaniem zaawansowanych algorytmów i modeli uczenia maszynowego. W ramach realizacji pracy chciałem zgłębić zarówno tradycyjne podejście, jak i nowoczesne rozwiązania, tj. połączenie matematyki, algorytmiki i uczenia maszynowego, które umożliwiają podejmowanie lepszych decyzji w grze. Dodatkowo, tematyka ta jest praktyczna i interdyscyplinarna, a realizacja projektu pozwoliła nie tylko na rozwiniecie umiejętności programistycznych, ale również zdobyć wiedzę z zakresu uczenia maszynowego/sztucznej inteligencji i teorii gier. Pomimo tego, że nie wszystkie wstępne założenia zostały wykonane to sam proces projektowanie aplikacji szachowej był wyzwaniem, które pozwoliło połączyć teoretyczne zagadnienia z ich praktycznym zastosowaniem.

1.3. Terminologia szachowa

Szachy to strategiczna gra planszowa rozgrywana przez dwóch graczy, która odbywa się na 64-polowej planszy. Jeden gracz posiada bierki koloru białego, a drugi czarnego. Poznanie jej zasad jest niezbędne w kontekście tworzenia silników szachowych czyli programów komputerowych, które analizują pozycje na szachownicy oraz próbują generować najlepsze ruchy.

1.3.1. Bierki szachowe

Każdy z graczy posiada 16 bierek, które mają specyficzne cechy i możliwości ruchu. Figury to wszystkie bierki z wyjątkiem pionów. Bierki w stanie początkowym planszy zostały przedstawione na rysunku 1. Rodzaje bierek to:

- król, który jest najważniejszą figurą w grze, która jest celem ataku przeciwnika. Król może poruszać się o jedno pole w dowolnym kierunku,
- hetman, który może przemieszczać się o dowolną liczbę pól w pionie, poziomie oraz po przekątnych,

- wieża, która porusza się o dowolną liczbę pól w pionie oraz w poziomie,
- goniec, który porusza się o dowolną liczbę pól po przekątnych,
- skoczek, który porusza się w kształcie litery "L", czyli o dwa pola w jednym kierunku oraz o jedno pole w drugim,
- pion, który przemieszcza się o jedno pole do przodu, z wyjątkiem pierwszego ruchu, gdzie może przesunąć się o dwa pola. Pion może zbić bierkę przeciwnika po przekątnej.

1.3.2. Zasady gry

Główne zasady gry w szachy to:

- szach - sytuacja, w której król jest zagrożony biciem w następnej turze. Gdy to nastąpi musi on natychmiast uniknąć zagrożenia,
- mat - sytuacja, w której król jest w pozycji szacha i nie ma żadnej możliwości obrony przed nim. Mat jest jednoznaczny ze zwycięstwem gracza,
- pat - sytuacja, w której gracz nie ma możliwości wykonania ruchu oraz nie jest w pozycji szacha. Gra wtedy kończy się remisem,
- remis - sytuacja, w której na planszy zabraknie bierek do matowania, dojdzie do 50 posunięć bez zmiany bierek na szachownicy lub posunięć pionków, nastąpi trzykrotne powtórzenie pozycji lub zostanie on obustronnie postanowiony przez graczy.

1.3.3. Specjalne ruchy

W szachach występują specjalne ruchy, które bierki mogą wykonać. Te ruchy to:

- roszada, tj. ruch, którego celem jest zapewnienie bezpieczeństwa królowi; król przemieszcza się o dwa pola w stronę wieży, a wieża przechodzi na pole obok króla,
- promocja pionka, gdy pion dojdzie do swojego ostatniego pola to zamienia się on na dowolną figurę wybraną przez gracza, którą zwykle jest hetman,
- bicie w przelocie, tj. specjalne bicie pionka, które może nastąpić, gdy pionek przeciwnika wykonuje ruch o dwa pola do przodu, a jego nowe pole jest obok pionka gracza; gracz może wówczas zbić tego pionka, jakby poruszył się on tylko o jedno pole; możliwość ta mija w następnej turze.

1.3.4. Wartości liczbowe bierek

Bierki posiadają przyporządkowane im wartości liczbowe, które odzwierciedlają ich potencjalny wpływ na rozgrywkę. Wartości bierek to:

- król - królowi nie przypisuje się konkretnej wartości, ponieważ jest najważniejszą bierką i jego strata oznacza koniec gry.
- hetman - 9 punktów,
- wieża - 5 punktów.

- goniec - 3 punkty,
- skoczek - 3 punkty,
- pionek - 1 punkt.

2. Silniki szachowe

Pierwszy silnik szachowy zdolny do gry w szachy powstał już na początku lat 50-tych XX wieku [4]. W tym samym roku ukazała się publikacja C. Shannona, jednego z twórców teorii informacji, na temat wykorzystania komputera do gry w szachy [5]. Niestety poziom gry pierwszego algorytmu był na tyle niski, że nawet początkujący gracze mogli z łatwością go pokonać. W kolejnych dekadach rozwijano i optymalizowano algorytmy silników, wprowadzając m.in. cięcia alfa-beta, wyjaśnione w rozdziale 3.4.2, które znacząco poprawiły ich wydajność. Głównym ograniczeniem pozostawały jednak niewystarczające moce obliczeniowe komputerów. Wraz z postępem technologicznym i wzrostem mocy obliczeniowej silniki stawały się coraz lepsze. Przełomowym wydarzeniem było zwycięstwo komputera nad arcymistrzem szachowym, który to tytuł oznacza najwyższy tytuł szachowy, w oficjalnym turnieju w 1988 roku.

Globalny rozgłos przyniósł jednak rok 1997, kiedy komputer IBM Deep Blue pokonał ówczesnego mistrza świata Garry'ego Kasparowa, w meczu składającym się z sześciu partii [6]. Wynik meczu wynosił 3,5–2,5 na korzyść komputera. Było to pierwsze zwycięstwo „maszyny" nad mistrzem świata w klasycznych partiach turniejowych i uznawane jest za jedno z najważniejszych wydarzeń w historii uczenia maszynowego/sztucznej inteligencji. Kolejne lata przyniosły szybki rozwój silników szachowych, napędzany postępem w optymalizacji algorytmów i mocy obliczeniowej. W 2017 roku nastąpił przełom wraz z pojawieniem się silnika AlphaZero [7], opracowanego przez DeepMind. AlphaZero korzystał z głębokich sieci neuronowych i uczenia ze wzmocnieniem, co pozwoliło mu samodzielnie opanować grę w szachy na najwyższym poziomie, bez dostępu do bazy danych dotyczącej wiedzy na temat gry w szachy. W meczu AlfaZero przeciwko Stockfish [8], tj. ówcześnie najlepszym silnikiem opartym na metodach klasycznych, silnik AlphaZero wygrał 28 z 100 partii, a 72 zakończyły się remisem. Był to znaczący przełom, który wyznaczył nowy kierunek rozwoju silników szachowych. W 2020 roku do silnika Stockfish wprowadzono technologię NNUE (z ang. Efficiently Updatable Neural Network), która umożliwiła bardziej precyzyjną ocenę pozycji na szachownicy przy jednoczesnym zachowaniu wysokiej wydajności obliczeniowej. Dzięki temu Stockfish znacznie podniósł swój poziom gry, łącząc klasyczne podejście z nowoczesnymi metodami opartymi na sieciach neuronowych.

Obecnie najlepsze silniki szachowe osiągnęły poziom, który znacznie przewyższa możliwości człowieka. Zawodnicy nie traktują już tych programów jako przeciwników, lecz jako zaawansowane narzędzia do treningu, analizy partii i odkrywania nowych strategii. Szachy stały się także częściowo rozwiązane z matematycznego punktu widzenia, tzn. istnieje baza danych zawierająca ocenę wszystkich pozycji, w których na szachownicy znajduje się maksymalnie 7 figur, wtedy dostępne są dokładne analizy wskazujące, czy pozycja jest wygrana, remisowa czy przegrana, niezależnie od dalszych ruchów [9].

3. Klasyczne podejście do tworzenia silników szachowych

Zgodnie z klasycznym podejściem do gry w szachy, najważniejszymi aspektami silników szachowych są:

- reprezentacja stanu gry,
- ocena pozycji,
- przeszukiwanie możliwych ruchów.

Stan gry, czyli pozycja figur na planszy oraz inne informacje o grze, na przykład prawa do roszady, są przechowywane w pamięci i aktualizowane po wykonaniu ruchu. Na podstawie tego stanu algorytm generuje możliwe ruchy i przeszukuje ich warianty, analizując kolejne pozycje do określonej głębokości. Następnie ocenia pozycje zgodnie z przyjętymi kryteriami, np. stanem materialnym na szachownicy, tj. suma wartości bierek na planszy, bezpieczeństwem króla lub strukturą pionów. Na tej podstawie wybierane są ruchy maksymalizujące przewagę dla danego gracza. Przechodząc wstecz przez drzewo przeszukiwań, algorytm wyznacza najlepszy ruch dla bieżącej pozycji na szachownicy.

3.1. Reprezentacja stanu gry

Stan gry w szachach to nie tylko pozycja figur na szachownicy, ale także informacje niezbędne do przestrzegania zasad gry, takie jak:

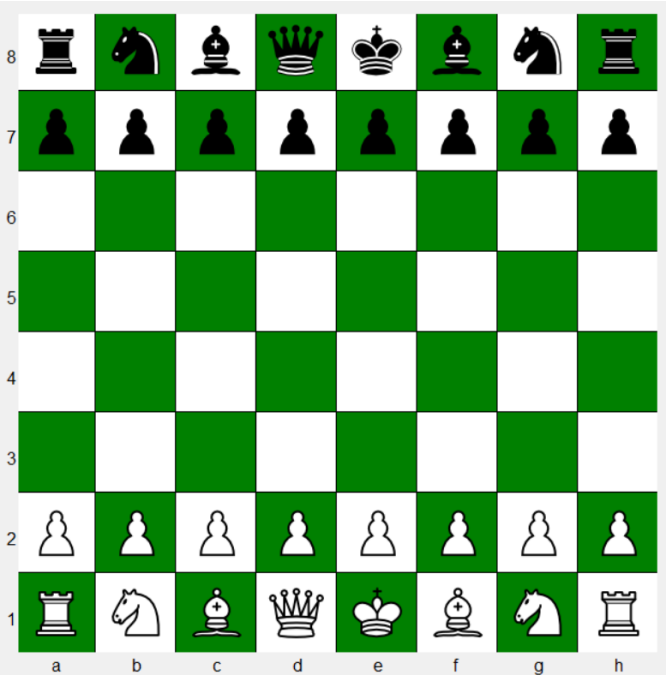
- możliwość wykonania roszad,
- licznik dla zasady 50 posunięć,
- możliwość bicia w przelocie.

Reprezentacja może być przedstawiona na różne sposoby, np. może być to dwuwymiarowa tablica 8x8, która przechowuje informacje o pozycji oraz dodatkowa tablica, która przechowuje inne informacje dotyczące gry. Możliwym jest zapisanie wszystkich informacji w jednej dwuwymiarowej tablicy, która ma wymiar zależny od potrzeb silnika. Zaawansowanym podejściem jest przedstawienie pozycji za pomocą tzw. bitboardów, które pozwalają na szybkie operacje matematyczne.

Przykładowym podejściem opartym na dwuwymiarowej tablicy przechowującej pozycje i tablicy przechowującej inne informacje dla pozycji początkowej, przedstawionej na rysunku 1, jest przykład zamieszczony na listingu 1.

Listing 1. Przykład reprezentacji stanu gry.

```
szachownica = [
  ["r", "n", "b", "q", "k", "b", "n", "r"],
  ["p", "p", "p", "p", "p", "p", "p", "p"],
  [".", ".", ".", ".", ".", ".", ".", "."],
  [".", ".", ".", ".", ".", ".", ".", "."],
  [".", ".", ".", ".", ".", ".", ".", "."],
  [".", ".", ".", ".", ".", ".", ".", "."],
  ["P", "P", "P", "P", "P", "P", "P", "P"],
  ["R", "N", "B", "Q", "K", "B", "N", "R"]]

  info_o_rozgrywce = [
  "bialy",                              # Gracz na ruchu
  {"bialy": {"krotka": True, "dluga": True},  # Mozliwosci roszad
   "czarny": {"krotka": True, "dluga": True}},
  "-",                                  # Bicie w przelocie
  0,                                    # Dla zasady 50 ruchow
  1                                     # Numer pelnego ruchu
  ]
```



Rys. 1. Szachownica z rozmieszczonymi bierkami w pozycji początkowej.

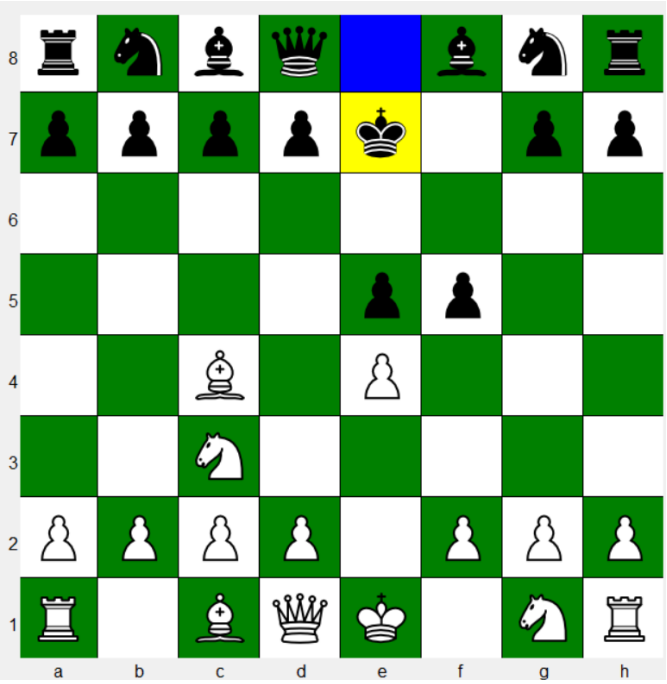
3.2. Ocena pozycji

Ocena pozycji polega na przyporządkowaniu wartości liczbowej do pozycji na szachownicy, która

odzwierciedla przewagę któregoś z graczy lub jej brak. Główne kryteria wpływające na ocenę to:

- stan materialny na planszy, tzn. suma wartości bierek na planszy obydwu graczy,
- pozycjonowanie bierek, tzn. wpływ figur na grę w zależności od ich położenia na planszy,
- bezpieczeństwo króla, tzn. czy król jest narażony na ataki,
- mobilność, tzn. ilość możliwych ruchów do wykonania.

Ponadto można uwzględnić więcej czynników w ocenie pozycji, takich jak struktura pionów czy kontrola centrum. Natomiast im bardziej skomplikowana jest ocena pozycji tym więcej wymaganej jest mocy obliczeniowej. Przykładowo dla pozycji przedstawionej na rysunku 2 król czarnych zrobił ruch z pozycji e8 na pozycję e7, który go odsłonił na potencjalne ataki oraz zablokował ruchy swoich figur co spowodowało spadek bezpieczeństwa króla oraz mobilności figur gracza czarnego, więc ocena pozycji przesunęła się na korzyść białych.



Rys. 2. Ocena pozycji.

3.3. Przeszukiwanie możliwych ruchów

Przeszukiwanie możliwych ruchów w silnikach szachowych polega na generowaniu możliwych sekwencji ruchów dla danej pozycji i wyborze opcji, której ocena jest najbardziej korzystna dla gracza. Najczęściej stosowane algorytmy do przeszukiwania w szachach to:

- algorytm Minimax z cięciami alfa-beta [10], czyli techniką optymalizacji, która pomija przeszukiwanie gałęzi drzewa nie wpływających na ostateczny wynik,
- algorytm Negamax z cięciami alfa-beta [11], który jest uproszczoną wersją algorytmu Minimax, i zakłada, że każdy gracz jedynie maksymalizuje wynik z odpowiednim znakiem,

- MTCS (z ang. Monte Carlo Tree Search) [12].

Silniki szachowe wykorzystują te algorytmy w sposób zoptymalizowany, dostosowany do swojej architektury, np.:

- silnik Stockfish, uważany za najbardziej zaawansowany, opiera się na algorytmie Negamax z cięciami alfa-beta,
- silnik AlfaZero, korzysta z algorytmu MCTS.

3.4. Negamax

Algorytm Negamax jest uproszczoną wersją algorytmu Minimax, dedykowaną dla gier dwuosobowych o sumie zerowej. Zaletą algorytmu Negamax jest ujednolicenie metryki maksymalizacji i minimalizacji wyników, dzięki czemu implementacja staje się bardziej „elegancka".

3.4.1. Opis działania algorytmu

Algorytm Negamax działa następująco:

1. Sprawdzenie czy jest spełniony warunek końcowy, tzn. zakończenie gry lub osiągnięcie maksymalnej głębokości przeszukiwania, wtedy algorytm zwraca ocenę pozycji.
2. Generowane są wszystkie możliwe ruchy dla danej poyzycji.
3. Algorytm symuluje dany ruch na planszy.
4. Algorytm jest wywoływany rekurencyjnie dla nowo powstałej pozycji i znak oceny jest zmieniany na przeciwny.
5. Wybranie ruchu, który maksymalizuje ocenę pozycji dla aktualnego gracza.

Pseudokod algorytmu Negamax jest przedstawiony na listingu 2.

Listing 2. Pseudokod algorytmu Negamax.

```
function negamax(position, depth):
    if depth == 0 or game_over(position):
        return evaluate(position)

    max_score = -
    for each move in generate_moves(position):
        new_position = apply_move(position, move)
        score = -negamax(new_position, depth - 1)
        max_score = max(max_score, score)

    return max_score
```

3.4.2. Algorytm z cięciami alfa-beta

Modyfikacją algorytmu, która pozwala na filtrowanie sporej liczby ruchów, za pomocą tzw. cięcia gałęzi w drzewie jest wprowadzenie cięć określanych terminem alfa-beta. Zasada cięć alfa-beta polega na pomijaniu gałęzi drzewa przeszukiwań, które nie wpłyną na ostateczną decyzję, na podstawie progów alpha i beta, co przyspiesza działanie algorytmu. Po wprowadzenie cięć, równoważny pseudokod algorytmu jest przedstawiony na listingu 3, w którym argumenty funkcji to:

- position, tj. pozycja na szachownicy,
- depth, tj. maksymalna głębokość przeszukiwania,
- alpha, tj. najlepsza możliwa ocena pozycji gracza; algorytm Negamax przerywa przeszukiwanie gałęzi, gdy wartość alpha przekroczy wartość beta, ponieważ dalsze przeszukiwanie nie wpływnie na końcowy wynik,
- beta, tj. najlepszy możliwa ocena pozycji drugiego gracza.

Listing 3. Pseudokod algorytmu Negamax z cięciami alfa-beta.

```
function negamax_alpha_beta(position, depth, alpha, beta):
    if depth == 0 or game_over(position):
        return evaluate(position)

    max_score = -
    for each move in generate_moves(position):
        new_position = apply_move(position, move)
        score = -negamax_alpha_beta(new_position, depth - 1, -beta, -
            alpha)
        max_score = max(max_score, score)
        alpha = max(alpha, score)

        if alpha >= beta:  # Ciecie
            break  # Nie ma potrzeby przeszukiwac dalej, ruch jest
                nieoplacalny

    return max_score
```

3.5. Przeszukiwanie a ocena pozycji

Aby silnik szachowy znajdował jak najlepsze ruchy w jak najkrótszym czasie należy odpowiednio dobrać przeszukiwanie i ocenę pozycji. Możliwa jest realizacja głębszego przeszukiwania z prostszą oceną pozycji lub odwrotnie, tzn. płytkie przeszukiwanie z zaawansowaną oceną pozycji. Odpowiednie wyważenie tych możliwości jest ważne dla efektywności silnika. Gdy któreś kryterium oceny jest czasochłonne to nie opłaca się go stosować, ponieważ zaważy to na szybkości przeszukiwania, co spowoduje, np. przeszukiwanie na mniejszej głębokości drzewa lub dłuższy czas obliczeń.

3.6. Optymalizacja

Aby algorytm działał szybciej stosuje się różne heurystyki, które pozwalają na przeszukiwanie na większą głębokość, która jest kluczowa dla efektywności działania silnika. Silnik, który osiąga większą głębokość przewiduje większą liczbę ruchów naprzód, co wiąże się z podejmowaniem lepszych decyzji. Techniki, które stosuje się aby zwiększyć wydajność algorytmu to:

- porządkowanie ruchów - priorytetowe przetwarzanie ruchów o największym potencjale (np. bicie, szachy), co zwiększa liczbę cięć w drzewie,
- wprowadzenie tzw. Quiescence search - dalsze przeszukiwanie drzewa w sytuacjach dynamicznych (np. po biciu lub gdy jest szach), aby uniknąć "efektu horyzontu", czyli błędnej oceny pozycji, wynikającej z pominięcia wygrywającego ruchu,
- NMH (z ang. Null Move Heuristic) - tymczasowe oddanie ruchu dla przeciwnika w celu szybkiego oszacowania pozycji,
- tabela transpoyzcji - przechowywanie w pamięci wyliczonych już ocen pozycji w celu ponownego ich wykorzystania.

4. Projekt i wykonanie aplikacji do gry w szachy

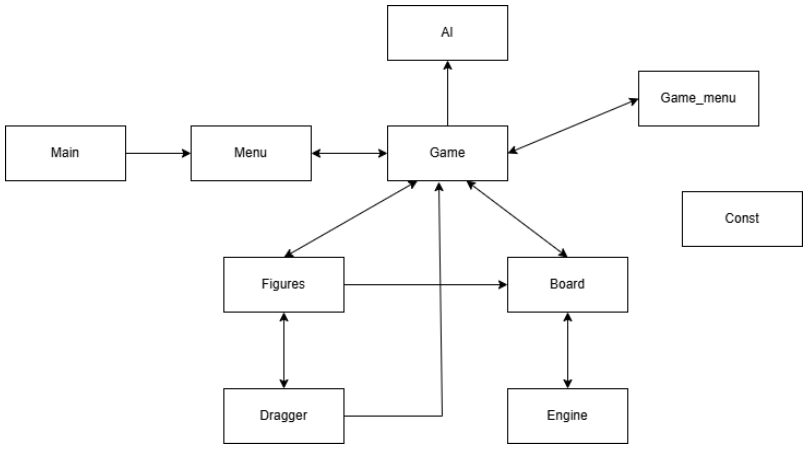
W ramach realizacji pracy dyplomowej zaprojektowano, a następnie wykonano aplikację do gry w szachy. W tym celu skorzystano z języka programowania Python oraz biblioteki Tkinter, która pozwala na wykonanie prostej aplikacji graficznej. Aplikacja po uruchomieniu wyświetla menu gry z opcjami takimi jak:

- nowa gra,
- zagraj z komputerem,
- wyjście.

Po wyborze jednego z dwóch trybów, tzn. gry samodzielnej lub gry z komputerem, aplikacja wyświetla ponumerowaną szachownicę z bierkami na planszy, co pokazano na rysunku 1. Ponadto, aplikacja zawiera menu gry oraz dodatkowe informacje: ocenę pozycji, liczbę punktów, zbite figury oraz historię gry. Aplikacja pozwala na przeprowadzenie rozgrywki zgodnie z regułami gry w szachy, korzystając z przesuwania figur myszką. W zależności od wybranego trybu, gracz gra sam ze sobą lub z komputerem. W menu gry wyświetlają się również dwa przyciski: „Poddaj się" oraz „Wróć do menu". Interaktywna szachownica pozwala na wykonywanie ruchów jedynie zgodnych z zasadami gry w szachy, a gdy gra dobiegnie końca wyświetla się informacja o tym, który gracz wygrał. Kod wykonanej aplikacji został zamieszczony w Dodatku.

4.1. Struktura wykonanej aplikacji

Aplikacja została podzielona na logiczne moduły, które obejmują: menu gry, interfejs graficzny, implementację zasad gry oraz algorytmy uczenia maszynowego/sztucznej inteligencji. W aplikacji wykonano 10 modułów, które komunikują się pomiędzy sobą. Moduły wraz z ich zależnościami zostały przedstawione na rysunku 3, gdzie moduł *Const* zawiera stałe wykorzystywane w programie i jest używany w wielu modułach, dlatego nie zaznaczono jego powiązań.



Rys. 3. Struktura aplikacji.

4.2. Reprezentacja stanu gry

Stan gry reprezentowany jest za pomocą:

- dwuwymiarowej tablicy, która przechowuje aktualne położenie figur na planszy.
- tablicy pomocniczej, która zawieraja dodatkowe informacje o grze, np. informacje o prawach do roszady.

Przykładowy stan gry został przedstawiony na listingu 4.

Listing 4. Kod programu przedstawiający reprezentację stanu gry.

```
    self.board = [
            [-5, -3, -4, -9, -2, -4, -3, -5],
            [-1, -1, -1, -1, -1, -1, -1, -1],
            [0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0],
            [1, 1, 1, 1, 1, 1, 1, 1],
            [5, 3, 4, 9, 2, 4, 3, 5]
        ]
     self.info = [self.white_queen_castling_right, self.
        white_king_castling_right, self.black_queen_castling_right, self.
        black_king_castling_right,  last_move, move, promotion, enpassant
        , self.threefold_repetition, self.fifty_move_rule, self.score]
```

Poniżej przedstawiono sposób kodowania bierek:

- 1 - pionek,
- 2 - król,
- 3 - skoczek,
- 4 - goniec,
- 5 - wieża,
- 9 - hetman.

Sposób kodowania bierek pozwala na proste odróżnienie ich w programie. Bierki czarne mają przypisane wartości ujemne, co pozwala łatwo rozróżnić kolory. Tablica *self.info* zawiera dane niezbędne zarówno do analizy mechaniki gry, jak i do wizualizacji położenia figur w interfejsie graficznym.

4.3. Implementacja zasad gry w szachy

Do poprawnego działania silnika potrzebna jest implementacja zasad działania gry w szachy[1]. W tym celu, w ramach opracowanego algorytmu, skorzystano z następujących kluczowych zasad:

- sprawdzenie czy król jest szachowany,

[1]Zasady gry w szachy znajdują się m.in. na stronie: Wikipedia: Zasady gry w szachy.

- sprawdzenie czy na szachownicy jest mat,
- sprawdzenie czy na szachownicy jest remis,
- generowanie możliwych ruchów dla danej figury,
- generowanie wszystkich możliwych ruchów dla danego koloru,
- wykonanie ruchu figurą na planszy,
- cofnięcie ostatniego ruchu.

4.3.1. Pozycja szach

Algorytm sprawdza czy król danego koloru jest szachowany w sposób następujący:

- najpierw lokalizowana jest pozycja króla na planszy,
- następnie generowane są wszystkie możliwe ruchy dla przeciwnika,
- sprawdzanie czy któryś z ruchów przeciwnika może zbić króla.

Algorytm zaimplementowany w aplikacji jest przedstawiony na listingu 5.

Listing 5. Kod programu do sprawdzanie pozycji szach.

```
        def is_check(self, for_white):
            king_position = self.king_position(for_white)
            all_moves = self.valid_moves_black if for_white else self.
               valid_moves_white
            if all_moves is None:   # Przy inicjacji
                return False

            length = len(all_moves)

            for i in range(length):
                move = all_moves[i]
                if move[2] == king_position[0] and move[3] ==
                   king_position[1]:
                    return True

            return False
```

4.3.2. Pozycja mat

Aby dowiedzieć się czy na planszy jest pozycja mata należy sprawdzić czy dla któregoś z kolorów jednocześnie jest pozycja szacha i nie ma możliwości wykonania żadnego ruchu. Implementacja tej reguły w kodzie programu została przedstawiona na listingu 6.

Listing 6. Kod programu do sprawdzania pozycji mat.

```
    def checkmate(self):
        if self.is_check(WHITE) and self.valid_moves_white == []:
```

```
3               return False
4           if self.is_check(BLACK) and self.valid_moves_black == []:
5               return True
6
7           return None
```

4.3.3. Pozycja remis

Remis na szachownicy pojawia się wtedy, gdy:

- trzykrotnie powtórzy się ta sama pozycja,
- nastąpi 50 ruchów bez zbicia bierki, zmiany bierek na szachownicy i przesunięcia pionów,
- pojawi się pat w pozycji(sytuacja, w której gracz nie ma legalnych ruchów do wykonania i nie znajduje się w pozycji szach),
- brak na szachownicy wystarczającego materiału do zamatowania króla (np. król kontra król).

W aplikacji zaimplementowano wszystkie reguły oprócz trzykrotnego powtórzenia pozycji, która zostanie dodana do kodu w przyszłości, np. stosując następujące rozwiązania:

- przechowywanie w strukturze danych, np. w słowniku, zahaszowanych pozycji, tj. skrócony zapis stanu gry, wraz z ich częstością występowania,
- dynamiczna aktualizacja i weryfikacja - aktualizacja słownika po każdym ruchu i sprawdzanie, czy któraś z pozycji w słowniku wystąpiła trzykrotnie.

4.3.4. Generowanie możliwych ruchów dla danej bierki

Aby wykonać ruch na szachownicy potrzebna jest informacja czy dany ruch bierką jest zgodny z zasadami gry. Ruch zapisywany jest jako zmienna typu *tuple*, tj. krotka i zawiera informacje dotyczące pola początkowego i pola końcowego, które rozbite są jeszcze na zmienne x i y, odpowiadające dwuwymiarowej tablicy, tj.: *(x_start, y_start, x_koniec, y_koniec)*.

Do wygenerowania możliwych ruchów wymagane są informacje o:

- typie figury,
- zasadach ruchu danej figury,
- aktualnym położeniu figury,
- rozmieszczeniu pozostałych figur na szachownicy.

Przykładowo skoczek na planszy porusza się w kształcie litery „L” - przesuwa się o 2 pola w jednym kierunku i o 1 pole w drugim. Podczas generowaniu ruchów należy sprawdzić czy ruch mieści się na szachownicy o wymiarach 8x8 oraz czy na docelowym polu nie znajduje się już bierka. Jeśli pole jest zajęte przez bierkę przeciwnika to jest to możliwość wykonania operacji bicia. Ruchy skoczka zostały zaimplementowane w postaci kodu przedstawionego na listingu 7.

Listing 7. Kod programu do generowania ruchów skoczka.

```
1       def generate_knight_moves(self, x, y, moves):
```

```
        direction = [(1, 2), (2, 1), (1, -2), (-1, 2), (-2, 1), (-1, -2),
            (-2, -1), (2, -1)]
        is_white = self.is_white_piece(x, y)

        for dx, dy in direction:
            x_move = x + dx
            y_move = y + dy
            move = (x, y, x_move, y_move)
            if not (x_move >= 0 and y_move >= 0 and x_move < SIZE and
               y_move < SIZE):
                continue
            if self.board[y_move][x_move] != 0:
                if self.is_white_piece(x_move, y_move) == is_white:
                    continue
                else:
                    moves.append(move)
                    continue
            moves.append(move)
```

Powyższy kod określa wszystkie możliwe kierunki ruchu skoczka, a następnie sprawdza ich zgodność z zasadami gry. Dla kolejnych figur takich jak goniec, wieża czy hetman zastosowano podobne zasady zgodne z regułami dla tych figur. Różnice polegają na kierunkach ruchu oraz możliwości wielokrotnego przemieszczania się w jednym kierunku, co zostało zaimplementowane za pomocą dodatkowej pętli.

Bardziej skomplikowane okazało się generowanie ruchów pionków, które mają bardziej złożone zasady poruszania się, tzn.:

- mogą poruszać się o 1 pole w kierunku pola przemiany,
- z pola startowego mogą wykonać ruch o dwa pola,
- mogą zbijać figury przeciwnika po przekątnej,
- gdy dojdą do pola przemiany mogą zamienić się w dowolną figurę oprócz króla. Generowanie promocji pionka zostało zrealizowane inaczej niż reszta bierek, tzn. do wykonania ruchu potrzebne było dodanie 5 wartości do zmiennej typu tupla, tj parametr określający na jaką figurę zamienia się pion, np. (3, 6, 3, 7, "Q"), gdzie ostatnia wartość "Q"oznacza zamianę na hetmana,
- najbardziej skomplikowany ruch pionów to enpassant (tzw. bicie w przelocie), które wymaga ścisłego spełnienia następujących warunków:

  1. Przeciwnik musi wykonać ruch pionem o 2 pola.
  2. Pion musi znajdować się na sąsiednim polu w chwili wykonania ruchu.
  3. Możliwość bicia w przelocie istnieje tylko bezpośrednio po ruchu przeciwnika i mija w następnej turze.

Zasada bicia w przelocie komplikuje logikę z powodu tymczasowych praw, które dynamicznie trzeba aktualizować. Niemniej zasada ta została zaimplementowana w kodzie programu.

Generowanie ruchów króla również jest odmienne, tzn. oprócz zwykłych ruchów, należało dodatkowo wygenerować roszady[2], które są ruchem wymagającym specjalnych warunków. Warunki konieczne do wykonania roszady to:

- król nie wykonał dotychczas ruchu,
- wieża, z którą wykonywana jest roszada nie wykonała jeszcze ruchu,
- król oraz pola przez które przechodzi podczas roszady nie są szachowane.

Informacje dotyczące zasad dla roszady są przechowywane w programie i po wykonaniu każdego ruchu sprawdzane jest czy bierka, która się poruszyła to król lub wieża. Jeśli tak to warunki dla roszady są aktualizowane.

Każdorazowo lista ruchów jest analizowana, pod kątem zagrożenia dla króla. Każdy ruch jest symulowany na kopii bieżącego stanu gry, aby sprawdzić, czy po jego wykonaniu król będzie szachowany. Jeśli tak, ruch jest usuwany z listy możliwych. Zapobiega to wykonaniu ruchu, który odsłoniłby króla i doprowadził do pozycji szacha lub ustawieniu króla w pozycji do zbicia przez inną bierkę. Podczas implementacji napotkano dwa istotne błędy:

- niewykrywanie szacha w pewnych pozycjach - problem dotyczył funkcji cofania ruchu po symulacji. Błędnie zaimplementowany fragment kodu powodował niepoprawne odtwarzanie stanu gry po symulacji. Poprawa tego fragmentu kodu spowodowała, że funkcja ta działa zgodnie z założeniami,
- działanie rekurencji prowadziło do nieskończonej pętli - funkcja do generowania możliwych ruchów wymaga sprawdzenia pozycji szacha, a sprawdzenie szacha wymaga wygenerowania ruchów przeciwnika, które wymaga sprawdzenia szacha. To wzajemne wywoływanie funkcji prowadziło do rekurencji wzajemnej. Problem rozwiązano poprzez wprowadzenie zmiennej typu *bool* kontrolującej proces generowania ruchów. Przy drugim wywołaniu funkcji generowania ruchów pomijane jest sprawdzanie szacha, co zapobiega zapętleniu.

4.3.5. Generowanie wszystkich możliwych ruchów dla gracza

Aby wygenerować wszystkie możliwe ruchy gracza należy przeanalizować pozycje wszystkich bierek danego koloru na planszy. Dla każdej bierki generowane są jej możliwe ruchy, które następnie łączy się w złożoną strukturę danych tj. listę. Implementację funkcji generującej wszystkie możliwe ruchy dla gracza przedstawiono na listingu 8.

Listing 8. Kod programu do generowania wszystkich możliwych ruchów.

```
1     def all_valid_moves(self, for_white, checking = False):
2         all_moves = []
3         for i in range(SIZE):
4             for k in range(SIZE):
5                 if self.board[i][k] == 0:
```

[2]Roszada to specjalny ruch w szachach, w którym jednocześnie poruszają się król i wieża. Szczegóły na stronie: Wikipedia: Roszada.

```
                    continue

                if for_white != self.is_white_piece(k, i):
                    continue

                moves = self.valid_moves(k, i, checking)
                length = len(moves)
                for j in range(length):
                    all_moves.append(moves[j])

        return all_moves
```

4.3.6. Wykonanie ruchu

Wykonanie ruchu na szachownicy możliwe jest tylko wtedy, gdy zostanie on uznany za zgodny z zasadami, tj. na podstawie wcześniej wygenerowanych możliwych ruchów dla danej bierki. Aby wykonać ruch, konieczne są następujące informacje: pole początkowe i końcowe ruchu, typ ruchu oraz bierka, która ma zostać przemieszczona. Na szachownicy można wykonywać zarówno ruchy regularne, jak i specjalne. W celu określenia jaki typ ruchu należy wykonać, wykorzystuje się odpowiednie funkcje, które analizują pole początkowe, końcowe i przemieszczaną bierkę. Oprócz ruchów regularnych można wykonać także ruchy specjalne takie jak:

- roszada,
- promocja pionka,
- bicie w przelocie.

Po każdym ruchu aktualizowane są informacje o stanie gry i ostatnich ruchach takie jak:

- czy obecny ruch to promocja,
- czy obecny ruch to bicie w przelocie,
- aktualizowana jest informacja dotycząca zasady 50 ruchów,
- zapisanie ostatniego ruchu i ustawienie aktualnego ruchu,
- aktualizowane są prawa do roszady,
- zapisywane są zmiany, które są później wykorzystywane do cofnięcia ruchu.

Funkcja realizująca wykonanie ruchu na planszy została przedstawiona na listingu 9.

Listing 9. Kod programu, który wykonuje ruch na planszy.

```
        def move_board(self, start_x, start_y, end_x, end_y, promotion =
            None):
            is_white = self.is_white_piece(start_x, start_y)
            piece_from = self.get_figure(start_x, start_y)
            piece_to = self.get_figure(end_x, end_y)

            color = 1 if is_white else -1
```

```
7
8               prev_promotion = self.info[6]
9               prev_enpassant = self.info[7]
10              self.info[6] = False # promotion
11              self.info[7] = False # enpassant move
12
13              prev_50 = self.update_fifty_move_rule(start_x, start_y, end_x
                    , end_y)
14              prev_score = self.info[10]
15
16              # [queen_castling_rigths, king_castling_rigths, prev_50_move,
                     promotion, enpassant]
17              changes = [False, False, prev_50, prev_promotion,
                    prev_enpassant, prev_score]
18
19              if self.is_it_capture(end_x, end_y):
20                  self.info[10] -= piece_to
21
22
23              if self.is_castling(start_x, start_y, end_x, end_y,
                    piece_from):
24                  self.castle(start_x, start_y, end_x, color)
25
26              elif self.is_pawn_promotion(start_x, start_y, end_x, end_y):
27                  self.promotion(start_x, start_y, end_x, end_y, color,
                        promotion)
28
29              elif self.is_enpassant(start_x, start_y, end_x, end_y,
                    piece_from):
30                  self.enpassant(start_x, start_y, end_x, end_y, color,
                        piece_from)
31              else:
32                  self.regular_move(start_x, start_y, end_x, end_y,
                        piece_from)
33
34
35              last2_move = self.info[4]
36              last_move = self.info[5] # now last move is previous move
37              self.info[4] = last_move
38              self.info[5] = (start_x, start_y, end_x, end_y)
39
40              self.update_info_castling_rights(start_x, end_x, changes,
                    is_white, piece_from, piece_to)
41
42              return piece_to, is_white, changes, last2_move
```

4.3.7. Cofnięcie ruchu

Cofnięcie ruchu polega na przywróceniu stanu gry do stanu poprzedniego. Aby operacja ta była możliwa, podczas wykonywania ruchu należy zapisać wszystkie istotne informacje o pozycji bierek, co umożliwi późniejsze wywołanie funkcji cofającej. Funkcja odpowiedzialna za cofanie ruchu przywraca wszystkie zmiany, które wprowadziła funkcja wykonująca ruch, w tym: stan planszy, prawa do roszady, informacje o zasadzie 50 ruchów oraz wynik punktowy. Implementacja tej funkcji w aplikacji jest przestawiona na listingu 10.

Listing 10. Kod programu, który cofa ostatni ruch.

```
1         def undo_move_board(self, start_x, start_y, end_x, end_y, piece,
              is_white,changes, last2_move):
2         self.undo_castlig_right_changes(changes, is_white)
3
4         self.info[5] = self.info[4] ## cofanie aktualnego ruchu na
              ostatni ruchu
5         self.info[4] = last2_move
6
7
8         color = 1 if is_white else -1
9
10        piece_from = self.get_figure(end_x, end_y)
11
12        if self.is_castling(start_x, start_y, end_x, end_y, piece_from):
13            self.uncastle(start_x, start_y, end_x, end_y, color)
14        elif self.info[7]: # enpssant
15            self.undo_enpassant(start_x, start_y, end_x, end_y, color,
                  piece_from)
16        else:
17            self.undo_regular_move(start_x, start_y, end_x, end_y, color,
                  piece_from, piece)
18
19        self.info[6] = changes[3] # cofanie promocji
20        self.info[7] = changes[4] # cofanie enpassant
21        self.info[9] = changes[2] # cofanie zasady 50 ruchow
22        self.info[10] = changes[5] # cofanie score
```

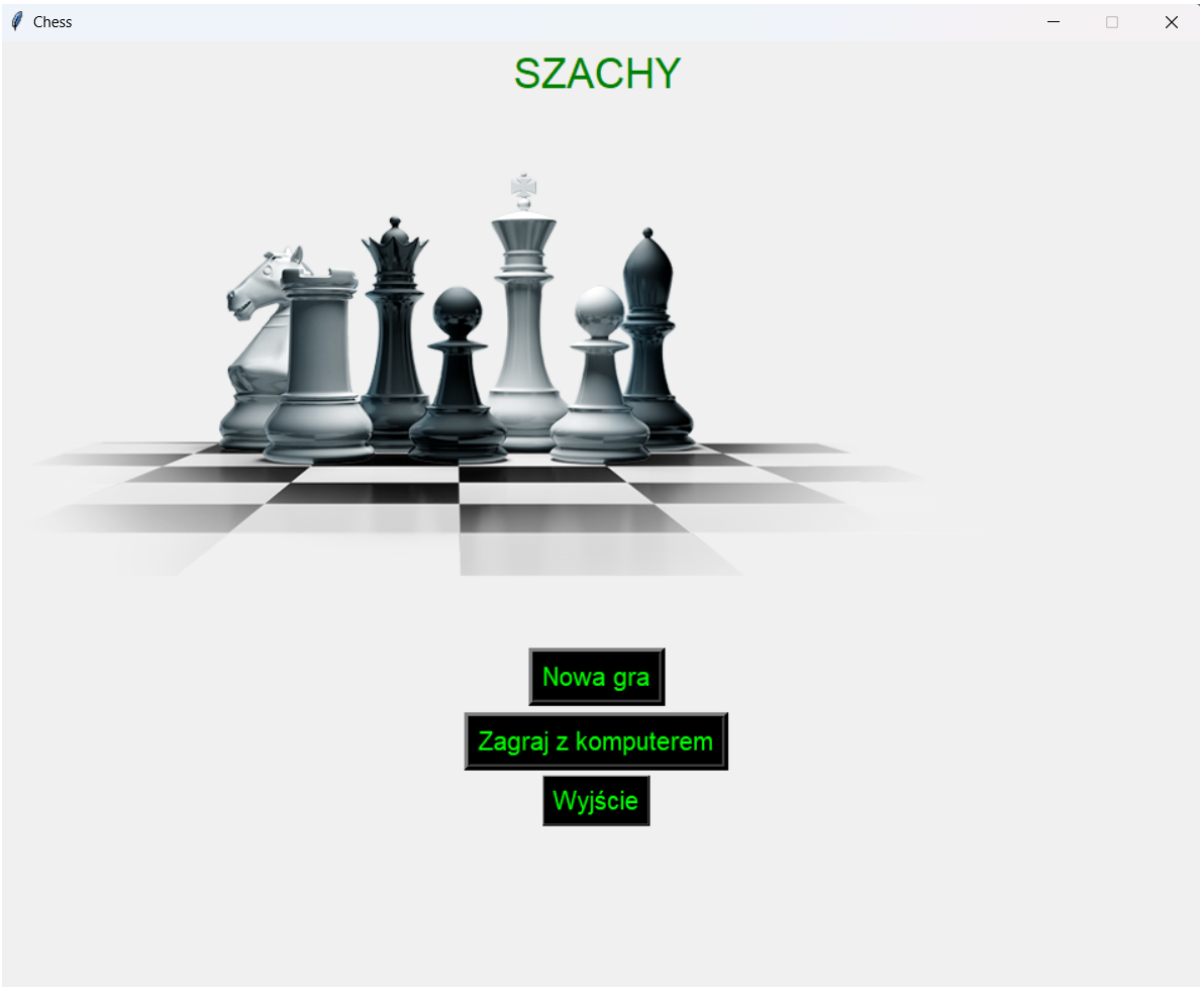
4.4. Interfejs graficzny

Interfejs graficzny podzielony został na 4 panele:

- menu główne,
- wyświetlenie szachownicy wraz z bierkami,
- menu gry,
- interakcja z szachownicą.

4.4.1. Menu główne

Menu główne aplikacji zawiera przyciski umożliwiające wybór trybu gry oraz wyjście z programu. Na górze ekranu wyświetlany jest napis „SZACHY”, a w tle widoczna jest szachownica z bierkami. Przyciski zostały zaprogramowane tak, aby reagowały na kliknięcia myszą, realizując odpowiednie funkcje, np. uruchomienie wybranego trybu gry lub zamknięcie aplikacji. Menu gry zostało przestawione na rysunku 4.



Rys. 4. Menu główne wykonanej aplikacji.

4.4.2. Szachownica i figury

Szachownica ma klasyczny dwukolorowy układ pól, gdzie kolor każdego pola zależy od parzystości sumy współrzędnych jej indeksów. Taki wzór tworzy charakterystyczny efekt szachowy. Przykład szachownicy został przedstawiony na rysunku 1, a odpowiadający kod został zamieszczony na listingu 11.

Listing 11. Kod programu do wyświetlania szachownicy.

```
    def display_board(self):
      for row in range(0, SIZE):
          y1 = row * SPACE_SIZE
          y2 = y1 + SPACE_SIZE
          for col in range(0, SIZE):
              if((row + col) %2 == 0):
                  color = SQUARES_COLORS[0]
              else:
                  color = SQUARES_COLORS[1]
              x1 = col * SPACE_SIZE
              x2 = x1 + SPACE_SIZE
              item = self.canvas_board.create_rectangle(x1,
                                        y1,
                                        x2,
                                        y2,
                                        fill=color)
              self.squares[row][col] = item

      self.canvas_board.place(x=50, y=100)
```

Rysowanie planszy odbywa się w pętli po wszystkich wierszach i kolumnach szachownicy, gdzie każdy prostokąt odpowiada jednemu polu. Kolory pól są przypisywane w zależności od ich współrzędnych, a gotowa plansza jest wyświetlana na ekranie. Proces wyświetlania bierek jest bardziej złożony i składa się z trzech głównych etapów:

- załadowania obrazów do programu, tj. przygotowanie grafik reprezentujących bierki,
- wyświetlenia bierek na szachownicy, tj. umieszczenie odpowiednich obrazów na planszy,
- powiązania każdej bierki z funkcjami, które umożliwią przesuwanie ich czyli zapewnienie z nimi interakcji.

Program najpierw wczytuje odpowiednio przygotowane grafiki, a następnie przechowuje je w zmiennej przypisanej do obiektu klasy *Figures*. Procedura wczytania grafik do programu jest przedstawiona na listingu 12.

Listing 12. Kod przedstawiający wczytanie grafik bierek do programu.

```
    def load_canvas_images(self, board):
      for i in range(SIZE):
          for j in range(SIZE):
              if board[i][j] != 0:
                  piece = board[i][j]
                  image = ImageTk.PhotoImage(Image.open(f"images/{piece
                     }.png"))
                  self.images[i][j] = image
      return self.images
```

Następnym etapem jest wyświetlenie bierek na planszy. Obrazy figur są rysowane w odpowiednich miejscach, a ich pozycje są zapisywane w strukturze danych. Proces ten realizuje kod przestawiony na listingu 13. Każda bierka jest oznaczana specjalną zmienną *tag*, która pozwala na jej identyfikację.

Listing 13. Kod przedstawiający wyświetlenie bierek.

```
        def display_figures(self, first_cordinates, canvas, board):
            for i in range(SIZE):
                for j in range(SIZE):
                    if board[i][j] != 0:
                        tag = (j, i)  # tag bazuj cy na pozycji
                            pocz tkowej
                        self.canvas_images[i][j] = canvas.create_image(
                            first_cordinates[0] + (j * SPACE_SIZE),
                            first_cordinates[1] + (i * SPACE_SIZE), image=
                            self.images[i][j], tags= tag)
                        canvas.tag_raise(self.canvas_images[i][j])

            return self.canvas_images
```

Na zakończenie powiązano bierki z funkcjami umożliwiającymi ich przesuwanie. W tym celu stworzono klasę *Dragger*, która zarządza interakcjami użytkownika z figurami za pomocą następujących funkcji:

- *drag_start()* - jest wywoływana, po kliknięciu na daną figurę,
- *drag_motion()* - jest wywoływana, gdy figura jest przemieszczana,
- *drag_stop()* - jest wywoływana, gdy figura zostanie upuszczona.

Powiązanie figur z funkcjami do przemieszczania zostało zrealizowane w postaci kodu przedstawionego na listingu 14.

Listing 14. Kod przedstawiający powiązanie bierek.

```
       def bind_figures(self, canvas):
            dragger = Dragger(self, canvas, self.board, self.game)
            for i in range(SIZE):
                for j in range(SIZE):
                    if self.board.board[i][j] != 0:
                        canvas.addtag_withtag("piece", self.canvas_images
                            [i][j])
                        canvas.tag_bind(self.canvas_images[i][j],  "<
                            ButtonPress-1>", dragger.drag_start)
                        canvas.tag_bind(self.canvas_images[i][j],  "<B1-
                            Motion>", dragger.drag_motion)
                        canvas.tag_bind(self.canvas_images[i][j], "<
                            ButtonRelease-1>", dragger.drag_stop)
```

4.4.3. Interakcja z szachownicą

Klasa *Dragger* obsługuje interakcję użytkownika z bierkami na szachownicy, umożliwiając ich przemieszczanie za pomocą myszy. Proces ten odbywa się w trzech etapach:

1. Funkcja *drag_start()* jest wywoływana po kliknięciu na bierkę. Funkcja podświetla możliwe ruchy bierki, zaznaczając je na czerwono, co ilustruje rysunek 5. Funkcja identyfikuje współrzędne bierki na planszy oraz pozycję kursora, zapisując je w zmiennych klasy.

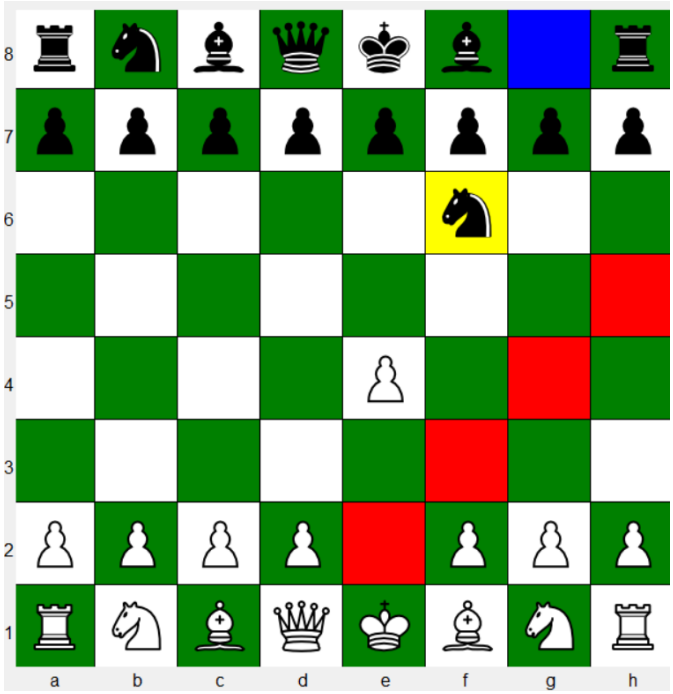
2. Funkcja *drag_motion()* jest aktywna podczas przeciągania bierki. Funkcja aktualizuje pozycję bierki na planszy w czasie rzeczywistym, umożliwiając jej płynne przemieszczanie.

3. Funkcja *drag_stop()* jest uruchamiana po puszczeniu przycisku myszy. Na podstawie pozycji kursora względem szachownicy podejmowane są odpowiednie działania:

   - jeśli bierka została wyprowadzona poza pole szachownicy, to wraca ona na miejsce początkowe i ruch nie jest wykonywany,
   - jeśli bierka została opuszczona w miejscu, w którym początkowo się znajdowała to nie jest wykonywany ruch i centruje się położenie bierki w polu,
   - ruch może być wykonany tylko wtedy, gdy jest on dozwolony; gdy nie jest on dozwolony to bierka wraca na swoje położenie początkowe,
   - gdy ruch jest dozwolony to wykonywana jest metoda przedstawiona na listingu 15.

Listing 15. Kod przedstawiający aktualizowanie bierek na szachownicy.

```
1                self.game.update(self.drag_data, x_start, y_start,
                     x_end, y_end)
```

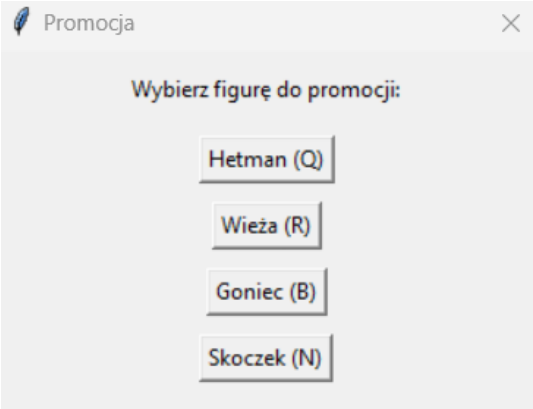


Rys. 5. Widok podświetlenia ruchów w wykonanej aplikacji.

4.4.4. Wykonywanie ruchu

Ruch jest realizowany za pomocą metody *update()* w klasie *Game*. Sprawdza ona, czy ruch wymaga promocji piona. Jeśli tak, wyświetlane jest okno wyboru figury promocyjnej, co jest pokazane na rysunku 6. Kolejne kroki obejmują:

- aktualizację pozycji na poziomie logicznym i graficznym,
- zmianę gracza na ruchu,
- modyfikację listy możliwych ruchów,
- aktualizację menu gry,
- sprawdzenie stanu gry.



Rys. 6. Okno wyboru figury do promocji.

Kod metody *update()*, powodujący aktualizację aktualnego stanu gry, został przedstawiony na listingu 16.

Listing 16. Kod przedstawiający aktualizację stanu gry.

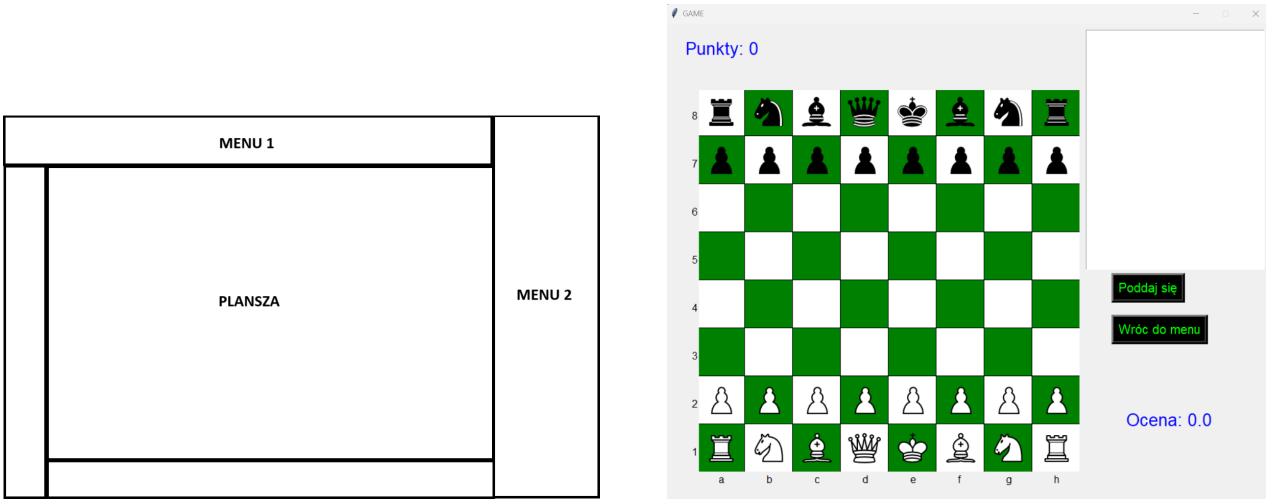
```
    def update(self, drag_data, x_start, y_start, x_end, y_end):
        promotion_val = None

        if self.board.engine.is_pawn_promotion(x_start, y_start,
            x_end, y_end):
            print("promotion")
            promotion_val = self.board.choose_piece()
            print(f"value {promotion_val}")

        self.move(drag_data, x_start, y_start, x_end, y_end,
            promotion_val)
        self.board.white_turn = not self.board.white_turn
        self.board.engine.update_valid_moves()
        self.update_menu_and_highlight(x_start, y_start, x_end, y_end
            )
        self.board.check_game_state()

        if settings["AI"] and self.board.white_turn != settings["TURN
            "] and self.board.game_on:
            self.make_ai_move()
            self.update_menu_and_highlight(x_start, y_start, x_end,
                y_end)
            self.board.check_game_state()
```

4.4.5. Menu gry

Menu gry zostało podzielone na kilka segmentów:

- plansza - centralny obszar z szachownicą i figurami,
- menu 1 – wyświetla punkty graczy oparte na stanie materialnym na planszy oraz zbite bierki, po zakończeniu gry pokazuje wynik końcowy,
- menu 2 - zawiera historię ruchów, przyciski „Poddaj się”, „Wróć do menu” oraz ocenę pozycji przez silnik szachowy,
- oznaczenia planszy - numery i litery wokół szachownicy.

Układ segmentów przedstawiono na rysunku 7a, a graficzny widok całości pokazano na rysunku 7b.



(a) Segmenty planszy (b) Widok gry aplikacji

Rys. 7. Podział ekranu na części: a) segmenty planszy, b) widok gry aplikacji.

Przykładowo, przycisk „Poddaj się” wywołuje metodę *surrender()* w klasie *Game*, która kończy grę, przypisując wynik zależny od tury gracza, a następnie wyświetla komunikat końcowy. Kod metody przedstawiono na listingu 17.

Listing 17. Kod programu do poddania się.

```
        def surrender(self):
            self.board.result = -1 if self.board.white_turn else 1
            self.board.game_on = False
            self.handle_game_end()
```

4.5. Plany związane z rozbudową aplikacji

Wykonaną aplikację planuje się rozbudować w przyszłości o następujące funkcjonalności:

- wprowadzenia zasady trzykrotnego powtórzenia pozycji,
- cofanie ruchu graficznie za pomocą interfejsu (obecnie można to robić tylko na warstwie logicznej),

- ustawianie pozycji na planszy za pomocą wpisanego ciągu znaków,
- zapisywanie historii gry do pliku,
- bardziej estetyczne menu do gry,
- dodanie możliwości grania na czas,
- możliwość robienia ruchu za pomocą kliknięcia na figurę, a później na pole docelowe,
- podczas przesuwania figury kliknięcie prawym przyciskiem myszy powoduje powrót do pozycji początkowej.

Te zmiany zwiększą wszechstronność aplikacji, czyniąc ją bardziej przyjazną dla użytkownika.

5. Implementacja klasycznego silnika szachowego

Klasyczny silnik szachowy został zaimplementowany przy użyciu zmodyfikowanego algorytmu Negamax oraz funkcji oceny pozycji. Silnik poprawnie generuje ruchy, umożliwiając grę z komputerem.

5.1. Algorytm Negamax

W implementacji algorytmu Negamax zastosowano dwie kluczowe optymalizacje: cięcia alfa-beta oraz szeregowanie ruchów. Modyfikacje te znacząco poprawiły wydajność silnika, skracając czas przeszukiwania drzewa możliwych ruchów. Kod zaimplementowanego algorytmu Negamax został przedstawiony na listingu 18.

Listing 18. Kod przedstawiający implementację algorytmu Negamax.

```
        def negamax(self, depth, color, engine_copy, alpha, beta, moves):
            if depth == 0:
                return color * self.evaluate(engine_copy), self.best_move

            for_white = True if color == 1 else False
            max_eval = float('-inf')

            moves = self.order_moves(moves, engine_copy)
            for move in moves:
                start_x, start_y, end_x, end_y, *promotion = move
                promotion_type = promotion[0] if promotion else None

                piece, is_white, changes, last2_move = engine_copy.
                    move_board(start_x, start_y, end_x, end_y,
                    promotion_type)
                engine_copy.update_valid_moves(True)

                next_moves  = engine_copy.valid_moves_black if for_white
                    is True else engine_copy.valid_moves_white

                evaluation, _ = self.negamax( depth - 1, -color,
                    engine_copy, -beta, -alpha, next_moves)
                evaluation = -evaluation

                engine_copy.undo_move_board(start_x, start_y, end_x,
                    end_y, piece, is_white, changes, last2_move)

                engine_copy.update_valid_moves(True)

                if evaluation > max_eval:
                    max_eval = evaluation
```

```
                    if depth == DEPTH:
                        self.best_move = move

                self.iteration += 1

                alpha = max(alpha, max_eval)
                if alpha >= beta:
                    break

            return (max_eval, self.best_move)
```

5.2. Optymalizacja silnika

W celu zwiększenia wydajności silnika przeprowadzono proces optymalizacji, który pozwolił na przyspieszenie obliczeń. W tym celu skorzystano z pomiaru czasu wykonywania obliczeń z pomocą biblioteki *time*. Przyjęto następujące parametry dla procesu optymalizacji:

- cięcia alfa-beta,
- szeregowanie ruchów,
- redukcja nadmiarowych obliczeń.

Cięcia alfa-beta są częścią optymalizacji algorytmu Negamax. Dzięki nim możliwe jest pomijanie wielu gałęzi drzewa gry, które nie prowadzią do lepszego wyniku. To usprawnienie pozwala na znaczną redukcję liczby analizowanych pozycji. Szeregowanie ruchów pozwala na przeszukiwanie obiecujących ruchów z pierwszeństwem, co zwiększa skuteczność cięć alfa-beta. Ruchy mające pierwszeństwo to te, w których dochodzi do zbicia bierki lub promocji piona. Promocja jest najbardziej priorytetowa, natomiast bicia są szeregowane ze względu na to jaka figura jest zbijana. Bicie hetmana jest bardziej priorytetowe niż bicie pionka. Implementacja szeregowania została przedstawiona na listingu 19.

Listing 19. Kod przedstawiający implementację szeregowania ruchów.

```
    def order_moves(self, moves, engine_copy):
        eval_moves = []
        for move in moves:
            start_x, start_y, end_x, end_y, *promotion = move

            promotion_bonus = 10 if promotion else 0
            capture_bonus = engine_copy.get_figure(end_x, end_y)

            eval_moves.append((capture_bonus + promotion_bonus, move))

        # Zwracnie posortowanych ruchow malejaco
        return [move for _, move in sorted(eval_moves, key=lambda x: x
            [0], reverse=True)]
```

Pierwotna wersja silnika przeprowadzała nadmiarowe obliczenia, takie jak wielokrotne generowanie możliwych ruchów przy każdym wywołaniu funkcji sprawdzających stan gry tj.: mat, szach, remis. Zostało to zmienione poprzez:

- przechowywanie aktualnych możliwych ruchów w zmiennych klasy *Engine*,
- aktualizowanie ruchów po wykonaniu lub cofnięciu ruchu.

Te zmiany znacząco zmniejszyły liczbę powtarzających się obliczeń. Nie wszystkie wprowadzone modyfikacje przyniosły jednak oczekiwane efekty, np.:

- zastosowanie macierzy typu array z biblioteki *Numpy* zamiast standardowych tablic dwuwymiarowych, tj. bazujących na listach, spowodowało spadek wydajności, czego powodem może być nieodpowiedni sposób implementacji,
- próba wdrożenia obliczeń bazujących na oddzielnych wątkach zakończyła się niepowodzeniem, ponieważ implementacja była nieodpowiednia, co spowodowało opóźnienia zamiast przyspieszenia.

5.3. Ewaluacja pozycji

Zaimplementowana ewaluacja pozycji bazuje na trzech kryteriach:

- ocena materiału,
- ocena pozycji,
- mobilność.

Każdemu z kryteriów oceny przypisano odpowiednią wagę tak, aby miały one różny wpływ na ocenę pozycji. Wagi te zostały określone na podstawie wiedzy o szachach oraz przykładowych grach z opracowanym silnikiem. Kod funkcji realizującej pełną ewaluację pozycji jest przedstawiony na listingu 20.

Listing 20. Kod przedstawiający implementację oceny pozycji.

```
1       def evaluate(self, engine):
2            material_score = self.evaluate_material(engine)
3            position_score = self.evaluate_position(engine)
4            mobility_score = self.evaluate_mobility(engine)
5
6            return (
7                    material_score * 1.2 +
8                    position_score * 0.5 +
9                    mobility_score * 0.1
10           )
```

5.3.1. Ocena materiału

Ocena materiału polega na sumowaniu wartości przypisanych poszczególnym bierkom na szachownicy. Najważniejszą figurą jest król, któremu przypisano wartość 100, natomiast pionek, jako najsłabsza bierka otrzymuje wartość 1. Kod funkcji został przedstawiony na listingu 21.

Listing 21. Kod przedstawiający implementację oceny materiału.

```
    def evaluate_material(self, engine):
        piece_values = {1: 1, 2: 100, 3: 3, 4: 3, 5: 5, 9: 9, -1: -1,
            -2: -100, -3: -3, -4: -3, - 5: -5, -9: -9,}
        score = 0
        for col in range(SIZE):
            for row in range(SIZE):
                piece = engine.get_figure(row, col)
                if piece != 0:
                    value = piece_values[piece]
                    score += value
        return score
```

5.3.2. Ocena ustawienia bierek

Ocena pozycjonowania bierze pod uwagę umiejscowienie bierek na planszy. Bazując na wiedzy o szachach można wywnioskować, które pozycje na szachownicy zwiększają potencjał danej figury, np.:

- pionek zbliżający się do pola promocji jest bardziej wartościowy niż pionek na pozycji wyjściowej,
- skoczek w centrum planszy ma większą mobilność niż skoczek w narożniku szachownicy.

Na tej podstawie skonstruowano tablice pozycji, które pozwalają na ocenę potencjału poszczególnych figur w zależności od ich położenia. Przykładowa tablica dla skoczka została przedstawiona na listingu 22.

Listing 22. Tabela oceny pozycjonowania dla skoczka.

```
knight_table = [
    [-5, -4, -3, -3, -3, -3, -4, -5],
    [-4, -2, 0, 0, 0, 0, -2, -4],
    [-3, 0, 1, 1.5, 1.5, 1, 0, -3],
    [-3, 0.5, 1.5, 2, 2, 1.5, 0.5, -3],
    [-3, 0, 1.5, 2, 2, 1.5, 0, -3],
    [-3, 0.5, 1, 1.5, 1.5, 1, 0.5, -3],
    [-4, -2, 0, 0.5, 0.5, 0, -2, -4],
    [-5, -4, -3, -3, -3, -3, -4, -5]
]
```

Tablice te są wykorzystywane do oceny pozycjonowania wszystkich figur, a ich wartości są sumowane, tworząc ostateczną ocenę pozycji dla obu graczy. Kod oceny pozycjonowania bierek został przedstawiony na listingu 23.

Listing 23. Kod przestawiający ocenę pozycjonowania bierek.

```
    def evaluate_position(self, engine):
        score = 0
```

```
3
4               for col in range(SIZE):
5                   for row in range(SIZE):
6                       piece = engine.get_figure(row, col)
7                       piece_abs = abs(piece)
8                       for_white = engine.is_white_piece(row, col)
9                       color = 1 if for_white else -1
10                      position_value = self.piece_square_score(piece, row,
                            col, color)
11                      if piece!= 0:
12                          total_value = piece_abs + position_value
13                          score += total_value if engine.is_white_piece(row
                                , col ) else -total_value
14
15
16              return score
```

5.3.3. Mobilność

Mobilność oceniana jest na podstawie ilości możliwych ruchów do wykonania przez graczy. Ten gracz, który ma ich większą możliwość uzyskuje większą ocenę mobilności. Kod oceny mobilności został przedstawiony na listingu 24.

Listing 24. Kod przestawiający ocenę mobilności figur.

```
1       def evaluate_mobility(self, engine):
2               return len(engine.valid_moves_white) - len(engine.
                    valid_moves_black)
```

5.4. Implementacja w grze

Podczas aktualizowania stanu gry, który został pokazany na listingu 16, ruch wykonany przez gracza powoduje odpowiednią odpowiedź silnika, gdy są spełnione następujące warunki:

- wybrano tryb gry z silnikiem,
- gra nie została zakończona.

Ruch wykonywany przez silnik polega na znalezieniu najlepszego posunięcia, a następnie wykonaniu go na planszy. Proces ten został zaimplementowany w sposób przedstawiony na listingu 25.

Listing 25. Kod przestawiający wykonanie ruchu przez komputer.

```
1           def make_ai_move(self):
2               promotion_val = None
3
4               start_time = time.time()
```

```
            best_move = self.ai.find_best_move(DEPTH, -1, self.board.
                engine)
            end_time = time.time()

            elapsed_time = end_time - start_time
            print("Elapsed time:", elapsed_time, "seconds")

            if best_move is None:
                print("No valid moves found for AI.")
                return

            if len(best_move[0]) == 5:
                promotion_val = best_move[0][4]

            print(f"best move {best_move}")

            self.move(None, best_move[0][0], best_move[0][1], best_move
                [0][2], best_move[0][3], promotion_val)
            self.board.white_turn = not self.board.white_turn
```

Znajdowanie najlepszego ruchu odbywa się za pomocą metody *find_ best_ move()*, która jest częścią klasy *AI*. Metoda ta uruchamia algorytm Negamax z wykorzystaniem kopii bieżącego stanu gry. Implementacja tej funkcji została przedstawiona na listingu 26.

Listing 26. Kod przestawiający znalezienie najlepszego ruchu przez komputer.

```
    def find_best_move(self, depth, color, engine):
        engine_copy = copy.deepcopy(engine)

        for_white = True if color == 1 else False
        valid_moves = engine_copy.all_valid_moves(for_white)
        score, best_move = self.negamax(depth, color, engine_copy, -
            MATE, MATE, valid_moves)

        return best_move, score
```

6. Testy

Aby ocenić działanie wykonanej aplikacji przeprowadzono szereg testów:

- poprawności działania aplikacji oraz logiki gry w szachy,
- analizę szybkości przeszukiwania na określonej głębokości przed i po wprowadzeniu poszczególnych etapów optymalizacji,
- „siły” gry korzystając z rozgrywki pomiędzy silnikami o różnym stopniu zaawansowania na najpopularniejszej platformie szachowej *chess.com* [13].

Przetestowano każdą funkcjonalność aplikacji, w tym: działanie przycisków, poprawność wyświetlania oraz interakcję z szachownicą. W żadnym z tych aspektów nie wykryto błędów. Zasady gry w szachy zostały dokładnie sprawdzone w różnych scenariuszach, co potwierdziło ich prawidłową implementację.

W celu oceny wpływu wprowadzonych optymalizacji na szybkość działania silnika przeanalizowano czas wykonania funkcji *find_best_move()* z klasy *AI*. Testy przeprowadzono w sposób, w którym mierzono czas działania przed i po wprowadzeniu każdej zmiany. Sprawdzono następujące optymalizacje:

- wprowadzenia cięć alfa-beta,
- szeregowanie ruchów,
- przechowywanie wyników obliczeń w zmiennych,
- zrównoleglenie procesu,
- zamiana dwuwymiarowej tablicy, bazującej na typie list na macierz typu array z biblioteki *Numpy*.

Efekty testów zostały przedstawione w tabelach 1-5.

Tabela 1. Test szybkości przed i po wprowadzeniu równoległości.

Czas po[s]	**Czas przed [s]**
2,69	0,63

Tabela 2. Test szybkości przed i po wprowadzeniu cięć alfa-beta.

Czas po[s]	**Czas przed [s]**
0,63	6,99

Tabela 3. Test szybkości przed i po wprowadzeniu przechowywania obliczeń w zmiennej.

Czas po[s]	**Czas przed [s]**
0,06	2,77

Tabela 4. Test szybkości przed i po wprowadzeniu macierzy typu array.

Nr. ruchu	**Czas po[s]**	**Czas przed [s]**
1	0,87	0,54
4	2,39	1,38
6	7,60	4,10

Tabela 5. Test szybkości przed i po wprowadzeniu szeregowania ruchów.

Nr. ruchu	**Czas po[s]**	**Czas przed [s]**
1	0,06	0,06
2	0,09	0,19
3	0,12	0,16
12	0,14	0,41
18	0,50	1,1

Wyniki testów, przedstawione w tabelach 1-5, pokazują, że takie zmiany jak cięcia alfa-beta,

szeregowanie ruchów oraz ograniczenie liczby obliczeń przyspieszyły działanie silnika. Natomiast optymalizacje związane z równoległością oraz wykorzystaniem biblioteki *Numpy* miały odwrotny efekt, tj. spowolnienie.

Dodatkowo, testy związane z wprowadzeniem biblioteki *Numpy* oraz szeregowanie ruchów przeprowadzono na większej liczbie pozycji, aby uwzględnić niewielkie różnice wyników. W szczególności szeregowanie ruchów nie przyniosło znaczących korzyści w początkowej fazie gry, gdy brak było bić lub promocji, które mają priorytet w tym algorytmie. Jednak w bardziej skomplikowanych pozycjach, z większą liczbą tych ruchów, optymalizacja ta zwiększyła szybkość przeszukiwania.

Na platformie *chess.com*, umożliwiającej grę z silnikiem o określonym poziomie rankingowym, przeprowadzono testy silnika na czterech różnych *poziomach zaawansowania* przy głębokości przeszukiwania ustawionej na 3. Wyniki testów zostały zestawione w tabeli 6. Opracowany silnik uzyskał wyniki wskazujące na poziom gry odpowiadający rankingowi w zakresie 1000–1200.

Tabela 6. Test „siły" gry silnika na stronie *chess.com*.

Ranking silnika	**Rezultat**
1000	Remis
1100	Wygrana
1200	Remis
1300	Przegrana

Niestety silnik napotyka trudności w końcówkach partii, szczególnie przy znajdywaniu mata, co skutkuje powtarzaniem ruchów, stagnacją w rozwoju pozycji lub dopuszczaniem do pata. Przykładem jest partia z silnikiem o rankingu 1000, gdzie z pozycji znacząco wygrywającej powstał pat i partia zakończyła się remisem. Według danych *chess.com* średni ranking graczy wynosi 630 [14]. Opracowany silnik szachowy można więc uznać za odpowiadający poziomowi średniozaawansowanego gracza.

7. Wnioski

Celem pracy dyplomowej było opracowanie i wykonanie aplikacji do gry w szachy z intuicyjnym interfejsem graficznym, umożliwiającej zarówno grę samodzielną, jak i z komputerem. W ramach projektu zaimplementowano pełną logikę gry w szachy oraz silnik szachowy oparty na udoskonalonym algorytmie Negamax i funkcji oceny pozycji. Ważnym elementem było również przeanalizowanie wpływu stosowanych heurystyk na szybkość działania oraz jakość gry. Wszystkie powyższe cele zostały zrealizowane, z wyjątkiem implementacji zasady trzech powtórzeń. Nie udało się zrealizować części dotyczącej wdrożenia sieci neuronowej NNUE do ewaluacji pozycji oraz porównania jej z klasyczną metodą oceny pozycji, co było spowodowane ograniczonym czasem na realizację projektu.

W ramach realizacji pracy wykonano aplikację szachową z przyjaznym interfejsem, umożliwiającą grę zarówno w trybie jednoosobowym, jak i przeciwko komputerowi. Opracowany silnik szachowy, w wyniku przeprowadzonych testów, został oceniony na poziom gry średniozaawansowanego gracza, co odpowiada rankingowi 1000–1200 według klasyfikacji *chess.com*. Przy głębokości przeszukiwania 3, silnik znajduje ruchy w ciągu kilku sekund, jednak przy głębokości 4 czas przeszukiwania znacząco się wydłuża, dochodząc nawet do 30 sekund.

Projektowanie silników szachowych jest złożonym i rozbudowanym zagadnieniem, wymagającym głębokiej wiedzy i znacznych nakładów czasu, aby opracować rozwiązanie o wysokiej efektywności. W przyszłości opracowana aplikacja może zostać rozwinięta o elementy, które nie zostały zaimplementowane w obecnej wersji, w szczególności wdrożenie sieci neuronowej NNUE oraz zasady trzech powtórzeń. Dodatkowe rozszerzenia mogą obejmować poprawę funkcjonalności aplikacji szachowej (omówione w podrozdziale 4.5) oraz udoskonalenie algorytmu przeszukiwania w celu zwiększenia głębokości przeszukiwania i poprawy wydajności.

Literatura

[1] *Python*. https://www.python.org, dostęp 1.12.2024.

[2] *Tkinter*. https://docs.python.org/3/library/tkinter.html, dostęp 1.12.2024.

[3] *Wikipedia: NNUE*. https://en.wikipedia.org/wiki/Efficiently_updatable_neural_network, dostęp 1.12.2024.

[4] *Wikipedia: szachy komputerowe*. https://pl.wikipedia.org/wiki/Szachy_komputerowe, dostęp 30.11.2024.

[5] CLAUDE E. SHANNON *Philosophical Magazine, Ser.7, Vol. 41, No. 314 - March 1950. XXII. Programming a Computer for Playing Chess1*. November 8, 1949

[6] *Wikipedia: Deep Blue versus Garry Kasparov*. https://en.wikipedia.org/wiki/Deep_Blue_versus_Garry_Kasparov, dostęp 1.12.2024.

[7] Dominik Klein, *Neural Networks for Chess The magic of deep and reinforcement learning revealed*, June 11, 2022.

[8] *Stockfish*. https://stockfishchess.org/, dostęp 1.12.2024.

[9] *Wikipedia: Endgame tablebase*. https://en.wikipedia.org/wiki/Endgame_tablebase, dostęp 1.12.2024.

[10] *Wikipedia: Algorytm alfa-beta*. https://pl.wikipedia.org/wiki/Algorytm_alfa-beta, dostęp 1.12.2024.

[11] *Wikipedia: Negamax*. https://en.wikipedia.org/wiki/Negamax, dostęp 1.12.2024.

[12] *Wikipedia: MCTS*. https://pl.wikipedia.org/wiki/Monte-Carlo_Tree_Search, dostęp 1.12.2024.

[13] *Strona internetowa do gry w szachy: Chess.com*. https://www.chess.com, dostęp 8.12.2024.

[14] *Średni ranking graczy na Chess.com*. https://www.chess.com/leaderboard/live, dostęp 8.12.2024.

[15] *Chess Programming*. https://www.chessprogramming.org/Main_Page, dostęp 27.11.2024.

[16] *Chess programmin wiki: Evaluation*. https://www.chessprogramming.org/Evaluation, dostęp 2.12.2024.

[17] *Chess programmin wiki: Search*. https://www.chessprogramming.org/Search, dostęp 2.12.2024.

[18] Francois Chollet, *Deep learning with Python*, 2018.

[19] George Allen and Unwin. *Games Playing with Computers*. https://www.chilton-computing.org.uk/acl/literature/books/gamesplaying/p003.htm, 1972, dostęp 25.11.2024.

[20] *LaTeX*. https://www.latex-project.org/, dostęp 2.12.2024.

[21] *Wikipedia: roszada*. https://pl.wikipedia.org/wiki/Roszada, dostęp 30.11.2024.

[22] *Writing a Chess Engine*. https://markus7800.github.io/blog/AI/chess_engine.html, dostęp 27.11.2024.

Dodatek

Aktualizowany kod programu jest umieszczony na platformie github:
https://github.com/JakubJakubczak/Chess.

Aktualny kod programu podzielony na moduły przedstawiony jest poniżej:

Listing 27. Moduł main stworzonej aplikacji.

```
from menu import *

if __name__ == '__main__':
    menu = Menu_own()
```

Listing 28. Moduł menu stworzonej aplikacji.

```
from tkinter import *
import Const
from Const import *
from game import *
from PIL import Image, ImageTk

class Menu_own:
    def __init__(self):
        self.menu_window = Tk()
        self.background_photo = None
        self.display_menu()
        self.menu_window.mainloop()

    def display_menu(self):
        self.menu_window.title("Chess")
        self.menu_window.resizable(False, False)

        canvas_menu = Canvas(self.menu_window, width=GAME_WIDTH, height=GAME_HEIGHT)
        canvas_menu.pack()

        self.background_photo = ImageTk.PhotoImage(Image.open(f"images/tlo2.png"))
        background = canvas_menu.create_image(0, 0, image=self.background_photo, anchor=NW)
        canvas_menu.coords(background, 0, 100)

        image_size = canvas_menu.bbox(background)
        end_of_backgroundY = image_size[3]

        game_Name = "SZACHY"
        font_size = 25
        game_label = canvas_menu.create_text(GAME_WIDTH / 2, 0 + font_size, text=game_Name, font=(
    "Roboto:", font_size), fill="green")

        button1 = Button(self.menu_window, text="Nowa gra",
                         font=("Roboto"),
                         command=self.start,
                         fg="#00FF00",
                         bg="black",
                         borderwidth=5,
                         relief="raised")

        button3 = Button(self.menu_window, text="Zagraj z komputerem",
                         font=("Roboto"),
                         command=self.start_ai,
                         fg="#00FF00",
```

```
                        bg="black",
                        borderwidth=5,
                        relief="raised")

        button2 = Button(self.menu_window, text="Wyjscie",
                        font=("Roboto"),
                        command=self.exit,
                        fg="#00FF00",
                        bg="black", )
        button1.place(x=0, y=0)
        button2.place(x=0, y=0)
        button3.place(x=0, y=0)
        self.menu_window.update_idletasks()

        width_button1 = button1.winfo_width()
        height_button1 = button1.winfo_height()
        height_button2 = button2.winfo_height()
        width_button2 = button2.winfo_width()
        width_button3 = button3.winfo_width()

        end_of_button1Y = end_of_backgroundY + height_button1
        end_of_button2Y = end_of_backgroundY + 2 * height_button1
        start_of_button3Y = end_of_backgroundY + 3 * height_button1 + 10
        start_of_button2Y = end_of_button1Y + 5
        end_of_button2Y = start_of_button2Y + height_button2
        start_of_button3Y = end_of_button2Y + 10

        button1.place(x=(GAME_WIDTH - width_button1) // 2, y=end_of_backgroundY)
        button3.place(x=(GAME_WIDTH - width_button3) // 2, y=start_of_button2Y)
        button2.place(x=(GAME_WIDTH - width_button2) // 2, y=start_of_button3Y)


    def start(self):
        self.menu_window.destroy()
        Const.settings["AI"] = False
        game = Game(self.back_to_menu)

    def start_ai(self):
        self.menu_window.destroy()
        Const.settings["AI"] = True
        game = Game(self.back_to_menu)

    def setup(self):
        pass

    def exit(self):
        self.menu_window.destroy()

    def back_to_menu(self):
        menu = Menu_own()
```

Listing 29. Moduł game menu stworzonej aplikacji.

```
from tkinter import *
from Const import *
from PIL import Image, ImageTk

class Game_menu:
    def __init__(self,frame, game):
        self.frame = frame
        self.game = game

```

```
        self.canvas_game_menu_1= Canvas(self.frame, width=GAME_MENU_1WIDTH, height=GAME_MENU_1HEIGHT)
        self.canvas_game_menu_2 = Canvas(self.frame, width=GAME_MENU_2WIDTH, height=GAME_MENU_2HEIGHT)
        self.canvas_game_menu_1.place(x=0, y=0)
        self.canvas_game_menu_2.place(x=650, y=0)

        button1 = Button(self.canvas_game_menu_2, text="Poddaj si ",
                         font=("Roboto"),
                         command=self.game.surrender,
                         fg="#00FF00",
                         bg="black",
                         borderwidth=5,
                         relief="raised")

        button2 = Button(self.canvas_game_menu_2, text="Wr c do menu",
                         font=("Roboto"),
                         command=self.game.back_to_menu,
                         fg="#00FF00",
                         bg="black",
                         borderwidth=5,
                         relief="raised")

        width_button1 = button1.winfo_width()
        height_button1 = button1.winfo_height()
        width_button2 = button2.winfo_width()
        height_button2 = button2.winfo_height()

        pady = 50

        button1.place(x=(GAME_MENU_2WIDTH) // 2 - width_button1 * 100,
                      y=GAME_MENU_2HEIGHT // 2 + 15 * height_button1 )
        button2.place(x=(GAME_MENU_2WIDTH) / 2 - width_button2 * 100,
                      y=GAME_MENU_2HEIGHT // 2 + 15 * ( height_button1 + height_button2) + pady)

        self.move_history_text = Text(self.canvas_game_menu_2, width=20, height=20, wrap="word")
        self.move_history_text.tag_configure("center", justify="center")
        self.move_history_text.place(x=10, y=10, width=GAME_MENU_2WIDTH - 20, height=GAME_MENU_2HEIGHT // 2)
        self.move_history_text.config(state=DISABLED)

        self.white_pieces = 0
        self.black_pieces = 0
        self.images = []

        self.evaluation_label = None
        self.score_of_game = None
    def display_history(self, history):
        self.move_history_text.config(state=NORMAL)
        self.move_history_text.delete(1.0, END)

        formatted_moves = ""
        for i, move in enumerate(history):
            if i % 2 == 0:
                formatted_moves += f"{(i // 2) + 1}. "
            formatted_moves += f"{move} "
        self.move_history_text.insert(END, formatted_moves.strip())
        self.move_history_text.tag_add("center", 1.0, END)
        self.move_history_text.config(state=DISABLED)


    def display_result(self, result):
```

```
69          text = None
70          if result == 0:
71              text = "REMIS"
72
73          if result == 1:
74              text = "WYGRANA BIA YCH"
75
76          if result == -1:
77              text = "WYGRANA CZARNYCH"
78
79          result_label = Label(self.canvas_game_menu_1, text=text, font=("Arial", 24), fg = "red")
80          result_label.place(x = 250, y = 19)
81
82      def display_score(self, score):
83          padx = 30
84          pady = 0
85
86          y = SMALL_PIECE_SIZE + (pady)
87          x = padx
88
89          if self.score_of_game is not None:
90              self.score_of_game.destroy()
91
92          self.score_of_game = Label(self.canvas_game_menu_1, text=f"Punkty: {score}", font=("Arial"
    , 20), fg = "blue")
93          self.score_of_game.place(x = x, y = y)
94
95      def display_eval(self, eval):
96          y = 600
97          x = 70
98
99          if self.evaluation_label is not None:
100             self.evaluation_label.destroy()
101
102         self.evaluation_label = Label(self.canvas_game_menu_2, text=f"Ocena: {eval}", font=("Arial
    ", 20), fg="blue")
103         self.evaluation_label.place(x=x, y=y)
104
105     def add_piece_to_player(self, for_white, tags):
106         start_coords = (int(tags[0]), int(tags[1]))
107         pady = 10
108         padx = 30
109         y = GAME_MENU_1HEIGHT - pady
110
111         if not for_white:
112             x = SMALL_PIECE_SIZE * self.black_pieces + padx
113             self.black_pieces += 1
114
115         if for_white:
116             x = GAME_MENU_1WIDTH - SMALL_PIECE_SIZE * self.white_pieces - padx
117             self.white_pieces += 1
118
119         piece = self.start_coordinates_to_figure(start_coords)
120
121         image = ImageTk.PhotoImage(Image.open(f"images/small/{piece}.png"))
122         self.images.append(image)
123         self.canvas_game_menu_1.create_image(x, y, image=image)
124
125     def start_coordinates_to_figure(self, coord):
126         if coord[1] == 1:
127             piece_value = -1
128             return piece_value
```

```
        elif coord[1] == 6:
            piece_value = 1
            return piece_value

        if coord[1] == 0 or coord[1] == 7:
            abs_value = None
            if coord[0] == 0 or coord[0] == 7:
                abs_value = 5
            if coord[0] == 1 or coord[0] == 6:
                abs_value = 3
            if coord[0] == 2 or coord[0] == 5:
                abs_value = 4
            if coord[0] == 3:
                abs_value = 9
            if coord[0] == 4:
                abs_value = 2

            color = -1 if coord[1] == 0 else 1

            return abs_value * color
```

Listing 30. Moduł game stworzonej aplikacji.

```
from Const import *
from board import *
from figures import *
from game_menu import *
from tkinter import *
from ai import *
from tkinter import messagebox
from Const import *
from menu import *
import time

class Game:
    def __init__(self, back_to_menu_callback):
        self.back_to_menu_callback = back_to_menu_callback
        self.game_window = Tk()
        self.game_window.title("GAME")
        self.game_window.resizable(False, False)
        self.frame = Frame(self.game_window, width=GAME_WIDTH, height=GAME_HEIGHT)
        self.frame.pack()
        self.board = Board(self.frame, self)
        self.board.engine.update_valid_moves()
        self.game_menu = Game_menu(self.frame, self)
        self.figures = Figures(self.board, self)
        self.ai = Ai()
        self.game_menu.display_score(self.board.score)
        evalu = self.ai.evaluate(self.board.engine)
        self.game_menu.display_eval(evalu)
        self.game_window.mainloop()


        if settings["AI"] == True and settings["TURN"] == False:
           random_move =  self.ai.generate_random_move(not settings["TURN"], self.board.engine)
           self.move(None, random_move[0], random_move[1], random_move[2], random_move[3])

    def handle_game_end(self):
        print("Game ended")
        result = self.board.get_result()
        self.game_menu.display_result(result)

```

```
    def update(self, drag_data, x_start, y_start, x_end, y_end):
        promotion_val = None

        if self.board.engine.is_pawn_promotion(x_start, y_start, x_end, y_end):
            print("promotion")
            promotion_val = self.board.choose_piece()
            print(f"value {promotion_val}")

        self.move(drag_data, x_start, y_start, x_end, y_end, promotion_val)
        self.board.white_turn = not self.board.white_turn
        self.board.engine.update_valid_moves()
        self.update_menu_and_highlight(x_start, y_start, x_end, y_end)
        self.board.check_game_state()

        if settings["AI"] and self.board.white_turn != settings["TURN"] and self.board.game_on:
            self.make_ai_move()
            self.update_menu_and_highlight(x_start, y_start, x_end, y_end)
            self.board.check_game_state()

        print(f"all_valid_moves {self.board.engine.all_valid_moves(False)}")

    def update_menu_and_highlight(self, x_start, y_start, x_end, y_end):
        self.board.add_to_history(x_start, y_start, x_end, y_end)
        self.game_menu.display_history(self.board.history)
        self.game_menu.display_score(self.board.info[10])
        eval = self.ai.evaluate(self.board.engine)
        self.game_menu.display_eval(eval)

        self.board.dehighlight_valid_moves()
        self.board.dehighlight_last_move()
        self.board.highlight_move()

    def make_ai_move(self):
        promotion_val = None

        start_time = time.time()
        best_move = self.ai.find_best_move(DEPTH, -1, self.board.engine)
        end_time = time.time()

        elapsed_time = end_time - start_time
        print("Elapsed time:", elapsed_time, "seconds")

        if best_move is None:
            print("No valid moves found for AI.")
            return

        if len(best_move[0]) == 5:
            promotion_val = best_move[0][4]

        print(f"best move {best_move}")
        self.move(None, best_move[0][0], best_move[0][1], best_move[0][2], best_move[0][3],
    promotion_val)
        self.board.white_turn = not self.board.white_turn

    def move(self, drag_data, x_start, y_start, x_end, y_end, promotion_val = None):
        self.board.engine.move_board(x_start, y_start, x_end, y_end, promotion_val)
        self.board.engine.update_valid_moves()
        self.figures.move_images(drag_data, x_start, y_start, x_end, y_end)

    def surrender(self):
        self.board.result = -1 if self.board.white_turn else 1
        self.board.game_on = False
```

```
        self.handle_game_end()

    def back_to_menu(self):
        self.game_window.destroy()
        self.back_to_menu_callback()
```

Listing 31. Moduł figures stworzonej aplikacji.

```

from tkinter import *
from PIL import Image, ImageTk
from Const import *
from dragger import *

class Figures:
    def __init__(self, board, game):
       self.board = board
       self.game = game
       self.images = [[None for _ in range(SIZE)] for _ in range(SIZE)]
       self.canvas_images = [[None for _ in range(SIZE)] for _ in range(SIZE)]
       self.images = self.load_canvas_images(board.board) # list
       image_dimensions = self.calculate_image_dimensions(self.images[0][0]) ## imgaes[0][0]
    because all images are the same size
       first_cordinates = self.calculate_first_cordinates()
       self.display_figures(first_cordinates, board.canvas_board, board.board) # list
       self.bind_figures(board.canvas_board)
       self.promotion = None

    def load_canvas_images(self, board):
        for i in range(SIZE):
            for j in range(SIZE):
                if board[i][j] != 0:
                    piece = board[i][j]
                    image = ImageTk.PhotoImage(Image.open(f"images/{piece}.png"))
                    self.images[i][j] = image
        return self.images

    def move_images(self, drag_data, x_start, y_start, x_end, y_end):
        if self.board.engine.is_it_capture(x_end, y_end):
            item_del = self.canvas_images[y_end][x_end]
            if item_del != None:
                tags = self.board.canvas_board.gettags(item_del)
                print(f"tags {tags}")
                self.game.game_menu.add_piece_to_player(self.board.white_turn, tags)
            self.board.canvas_board.delete(item_del)

        color = 1 if self.board.engine.is_white_piece(x_end, y_end) else -1
        piece = self.board.engine.get_figure(x_end, y_end)
        ## centrowanie figury po przeniesieniu
        figure_coords_x, figure_coords_y = self.calculate_first_cordinates()

        center_coords_x = figure_coords_x + (x_end * SPACE_SIZE)
        center_coords_y = figure_coords_y + (y_end * SPACE_SIZE)

        if drag_data == None:
            delta_center_x = -(x_end - x_start) * SPACE_SIZE
            delta_center_y = -(y_end - y_start) * SPACE_SIZE

        else:
            delta_center_x = drag_data["x"] - center_coords_x
            delta_center_y = drag_data["y"] - center_coords_y

```

```
54          item = self.canvas_images[y_start][x_start]
55          self.board.canvas_board.move(item, -delta_center_x, -delta_center_y)
56          self.canvas_images[y_start][x_start] = None
57          self.canvas_images[y_end][x_end] = item
58
59          if self.board.engine.is_castling(x_start, y_start, x_end, y_end, piece):
60              print("castling")
61              if self.board.engine.is_left_castling(x_start, x_end):
62                  # znalezc wieze
63                  item = self.canvas_images[y_start][0]
64                  # wyliczyc coordy i przeniesc ja w odpowiednie miejsce
65                  square_transported = 3
66                  delta_x =  square_transported * SPACE_SIZE
67                  self.board.canvas_board.move(item, delta_x, 0)
68                  # zmieni  canvas_images
69                  self.canvas_images[y_start][3] = item
70                  self.canvas_images[y_start][0] = None
71              if not self.board.engine.is_left_castling(x_start, x_end):
72                  print("right-castling")
73                  # znalezc wieze
74                  item = self.canvas_images[y_start][7]
75                  # wyliczyc coordy i przeniesc ja w odpowiednie miejsce
76                  square_transported = 2
77                  delta_x = square_transported * SPACE_SIZE
78                  self.board.canvas_board.move(item, -delta_x, 0)
79                  # zmieni  canvas_images
80                  self.canvas_images[y_start][5] = item
81                  self.canvas_images[y_start][7] = None
82
83          if self.is_pawn_promoted():
84              self.promote()
85
86          if self.board.info[7]:
87              print("move_images enpas")
88              item_del = self.canvas_images[y_end + color][x_end]
89              self.board.canvas_board.delete(item_del)
90              # self.canvas_images[y_end + color][x_end] = None
91
92      def is_pawn_promoted(self):
93          print(f"before")
94          for i in range(SIZE):
95              item_tag1 = None
96              item_tag2 = None
97
98              if not self.canvas_images[0][i] == None:
99                  item_tag1 = self.board.canvas_board.gettags(self.canvas_images[0][i])
100                 if item_tag1[1] == '6':
101                     value = self.board.board[0][i]
102                     self.promotion = (i, 0, value)
103                     return True
104
105             if not self.canvas_images[7][i] == None:
106                 item_tag2 = self.board.canvas_board.gettags(self.canvas_images[7][i])
107                 if item_tag2[1] == '1':
108                     value = self.board.board[7][i]
109                     self.promotion = (i, 7, value)
110                     return True
111
112         self.promotion = None
113
114         return False
115
```

```
    def promote(self):
        image_dict = {3: 1, 4: 2, 5: 0, 9: 3}
        promotion = self.promotion
        value = self.promotion[2]
        image = None
        x = promotion[0]
        y = promotion[1]

        if promotion[1] == 0:
            image = self.images[7][image_dict[value]]

        if promotion[1] == 7:
            image = self.images[0][image_dict[abs(value)]]

        print(f"promotion value{image_dict[abs(value)]}  ")
        self.board.canvas_board.itemconfig(self.canvas_images[y][x], image=image)

    def calculate_image_dimensions(self, image):
        return [image.width(),image.height()]

    def calculate_first_cordinates(self):
        figure_coords_x = SPACE_SIZE/2
        figure_coords_y = SPACE_SIZE/2
        return [figure_coords_x, figure_coords_y]

    def display_figures(self, first_cordinates, canvas, board):
        for i in range(SIZE):
            for j in range(SIZE):
                if board[i][j] != 0:
                    tag = (j, i)  # tag bazuj cy na pozycji pocz tkowej
                    self.canvas_images[i][j] = canvas.create_image(first_cordinates[0] + (j *
    SPACE_SIZE), first_cordinates[1] + (i * SPACE_SIZE), image=self.images[i][j], tags= tag)
                    canvas.tag_raise(self.canvas_images[i][j])

        return self.canvas_images

    def bind_figures(self, canvas):
        dragger = Dragger(self, canvas, self.board, self.game)
        for i in range(SIZE):
            for j in range(SIZE):
                if self.board.board[i][j] != 0:
                    canvas.addtag_withtag("piece", self.canvas_images[i][j])
                    canvas.tag_bind(self.canvas_images[i][j],  "<ButtonPress-1>", dragger.
    drag_start)
                    canvas.tag_bind(self.canvas_images[i][j],  "<B1-Motion>", dragger.drag_motion)
                    canvas.tag_bind(self.canvas_images[i][j], "<ButtonRelease-1>", dragger.
    drag_stop)
```

Listing 32. Moduł engine stworzonej aplikacji.

```
from Const import *
import copy

class Engine:
    def __init__(self, board, info, turn = None):
        self.board = board
        self.info = info
        self.turn = turn
        self.boolean = False
        self.last_valid_moves_white = None
        self.last_valid_moves_black = None
        self.valid_moves_white = None
```

```
13          self.valid_moves_black = None
14
15
16      def is_check(self, for_white):
17          king_position = self.king_position(for_white)
18          all_moves = self.valid_moves_black if for_white else self.valid_moves_white
19          if all_moves is None:   # Przy inicjacji
20              return False
21
22          length = len(all_moves)
23
24          for i in range(length):
25              move = all_moves[i]
26              if move[2] == king_position[0] and move[3] == king_position[1]:
27                  return True
28
29          return False
30
31      def is_king_on_board(self, for_white):
32          if for_white:
33              value = 2
34          else:
35              value = -2
36
37          for col in range(SIZE):
38              for row in range(SIZE):
39                  piece = self.board[col][row]
40                  if piece == value:
41                      return True
42
43          return False
44
45      def checkmate(self):
46          if self.is_check(WHITE) and self.valid_moves_white == []:
47              return False
48          if self.is_check(BLACK) and self.valid_moves_black == []:
49              return True
50
51          return None
52
53      def is_stalemate(self):
54          if self.valid_moves_white == [] and not self.is_check(WHITE):
55              return True
56          if self.valid_moves_black == [] and not self.is_check(BLACK):
57              return True
58
59          return False
60      def draw(self):
61          if self.is_threefold_repetition() or self.is_fifty_move_rule() or self.
    is_insufficient_material() or self.is_stalemate():
62              return True
63
64          return False
65
66      def is_threefold_repetition(self):
67          pass
68
69      def game_over(self):
70          checkmate = self.checkmate()
71          if checkmate == True:
72              return 1
73          if checkmate == False:
```

```
74              return -1
75          elif self.draw():
76              print("draw")
77              return 0
78          else:
79              return None
80
81      def is_fifty_move_rule(self):
82          if self.info[9] >= 100:
83              print("50 powtorzen")
84              return True
85
86          return False
87      def is_pawn_move(self, x_start, y_start):
88          piece = self.board[y_start][x_start]
89
90          if abs(piece) == 1:
91              return True
92
93          return False
94      def update_fifty_move_rule(self, x_start, y_start, x_end, y_end):
95          prev = self.info[9]
96          if self.is_pawn_move(x_start, y_start) or self.is_it_capture(x_end, y_end):
97              self.info[9] = 0
98
99          else:
100             self.info[9] += 1
101
102         return prev
103
104     def is_insufficient_material(self):
105         pieces_white = []
106         pieces_black = []
107         white_sum = 0
108         black_sum = 0
109
110         for i in range(SIZE):
111             for j in range(SIZE):
112                 piece = self.board[i][j]
113                 if piece!= 0:
114                     if piece > 0:
115                         pieces_white.append(piece)
116                         white_sum += piece
117                     else:
118                         pieces_black.append(piece)
119                         black_sum += piece
120
121                 if white_sum > 8 or abs(black_sum) > 8: # 8, poniewa  KNN daj  sume najwi ksz
122                     return False
123
124         # KB vs K
125         # KN vs K
126         if len(pieces_white) == 2 and len(pieces_black) == 1:
127             if 4 in pieces_white or 3 in pieces_white:
128                 return True
129
130         if len(pieces_black) == 2 and len(pieces_white) == 1:
131             if -4 in pieces_black or -3 in pieces_black:
132                 return True
133
134         # KNN vs K
135         if len(pieces_white) == 3 and len(pieces_black) == 1:
```

```
136             if pieces_white.count(3) == 2:
137                 return True
138 
139         if len(pieces_black) == 3 and len(pieces_white) == 1:
140             if pieces_black.count(-3) == 2:
141                 return True
142 
143         return False
144 
145     def is_square_empty(self,x, y):
146         if self.board[y][x] == 0:
147             return True
148 
149         return False
150     def is_pawn_move_of_2_sqr(self, x_start, y_start, x_end, y_end):
151         piece = self.get_figure(x_end, y_end)
152         if abs(piece) != 1:
153             return False
154 
155         if abs(y_end - y_start) == 2:
156             return True
157 
158         return False
159 
160     def is_enpassanat_pawn_nearby(self, x_pawn, y_pawn, x_enpas, y_enpas):
161         if not y_enpas == y_pawn:
162             return False, None
163 
164         if x_enpas == x_pawn - 1:
165             return True, -1
166 
167         if x_enpas == x_pawn + 1:
168             return True, +1
169 
170         return False, None
171 
172     def is_enpassant(self, x_start, y_start, x_end, y_end, piece):
173         if abs(piece) != 1:
174             return False
175 
176         if abs(x_end - x_start) == 0:
177             return False
178 
179         if self.is_square_empty(x_end, y_end):
180             return True
181 
182         return False
183 
184 
185     def is_pawn_promotion(self, x_start, y_start, x_end, y_end):
186 
187         piece = self.get_figure(x_start, y_start)
188         if not abs(piece) == 1:
189             return False
190 
191         if y_end == 0 or y_end == 7:
192             return True
193 
194         return False
195 
196     def promote(self,x,y, piece):
197         color = 1 if self.is_white_piece(x,y) else -1
```

```
198
199         self.board[y][x] = piece * color
200
201
202     def is_square_attacked(self, for_white, x, y):
203         all_moves = self.valid_moves_black if for_white else self.valid_moves_white
204
205         for move in all_moves:
206             if move[2] == x and move[3] == y:
207                 return True
208
209         return False
210
211     def queen_castling_rights(self, for_white):
212         if for_white:
213             return self.info[0]
214         else:
215             return self.info[2]
216
217     def king_castling_rights(self,for_white):
218         if for_white:
219             return self.info[1]
220         else:
221             return self.info[3]
222
223     def change_queen_castling_rights(self,for_white, change):
224         if for_white:
225             self.info[0] = change
226         else:
227             self.info[2] = change
228
229     def change_king_castling_rights(self, for_white, change):
230         if for_white:
231             self.info[1] = change
232         else:
233             self.info[3] = change
234
235     def is_queen_castling_possible(self, for_white):
236         if for_white:
237             row = 7
238         else:
239             row = 0
240
241         if not self.is_check(for_white) and \
242             self.board[row][1] == 0 and self.board[row][2] == 0 and self.board[row][3] == 0 and \
243             not self.is_square_attacked(for_white, 2, row) and not self.is_square_attacked(
    for_white, 3, row):
244
245             return True
246
247         return False
248
249     def is_king_castling_possible(self, for_white):
250         if for_white:
251             row = 7
252         else:
253             row = 0
254
255         if not self.is_check(for_white) and \
256                 self.board[row][5] == 0 and self.board[row][6] == 0 and \
257                 not self.is_square_attacked(for_white, 5, row) and not self.is_square_attacked(
    for_white, 6, row):
```

```
258             return True
259
260         return False
261
262     def is_castling(self, x_start, y_start, x_end, y_end, piece):
263         if abs(piece) == 2: # jest to kr l
264             if abs(x_start - x_end) == 2:   # ruch o 2 pola to roszada
265                 return True
266
267         return False
268
269     def is_left_castling(self, x_start, x_end):
270         if x_start > x_end:
271             return True
272         else:
273             return False
274
275     def is_it_capture(self, x_end, y_end):
276         if self.board[y_end][x_end] == 0:
277             return False
278         else:
279             return True
280
281     def king_position(self, for_white):
282         if for_white:
283             value = 2
284         else:
285             value = -2
286
287         for i in range(SIZE):
288             for k in range(SIZE):
289                 if self.board[k][i] == value:
290                     position = (i, k)
291                     return position
292
293     def is_valid_move(self, x_start, y_start, x_end, y_end):
294         moves = self.valid_moves(x_start, y_start)
295         move = (x_start, y_start, x_end, y_end)
296
297         if moves == None:
298             return False
299
300         for valid_move in moves:
301             if valid_move[:4] == move:
302                 return True
303
304         return False
305
306     def all_valid_moves(self, for_white, checking = False):
307         all_moves = []
308         for i in range(SIZE):
309             for k in range(SIZE):
310                 if self.board[i][k] == 0:
311                     continue
312
313                 if for_white != self.is_white_piece(k, i):
314                     continue
315
316                 moves = self.valid_moves(k, i, checking)
317                 length = len(moves)
318                 for j in range(length):
319                     all_moves.append(moves[j])
```

```
320
321         return all_moves
322
323     def append_move_as_promotion(self, x_start, y_start, x_end, y_end, moves):
324         moves.append((x_start, y_start, x_end, y_end, 9))
325         moves.append((x_start, y_start, x_end, y_end, 5))
326         moves.append((x_start, y_start, x_end, y_end, 4))
327         moves.append((x_start, y_start, x_end, y_end, 3))
328
329     def generate_pawn_moves(self, x, y, piece, moves, color):
330         # vertical for white pawn
331         if piece == 1 and y > 0 and self.board[y - 1][x] == 0:
332             if y - 1 == 0:
333                 self.append_move_as_promotion(x, y, x, y - 1, moves)
334             else:
335                 moves.append((x, y, x, y - 1))
336
337             if y == 6 and self.board[y - 2][x] == 0:
338                 moves.append((x, y, x, y - 2))
339
340         # vertical for black pawn
341         if piece == -1 and y + 1 < SIZE and self.board[y + 1][x] == 0:
342             if y + 1 == 7:
343                 self.append_move_as_promotion(x, y, x, y + 1, moves)
344             else:
345                 moves.append((x, y, x, y + 1))
346
347             if y == 1 and self.board[y + 2][x] == 0:
348                 moves.append((x, y, x, y + 2))
349
350         # diagonal to left for white pawn
351         if piece == 1 and y > 0 and x > 0 and self.board[y - 1][x - 1] < 0:
352             if y - 1 == 0:
353                 self.append_move_as_promotion(x, y, x - 1, y - 1, moves)
354             else:
355                 moves.append((x, y, x - 1, y - 1))
356
357         # diagonal to right for white pawn
358         if piece == 1 and y > 0 and x < SIZE - 1 and self.board[y - 1][x + 1] < 0:
359             if y - 1 == 0:
360                 self.append_move_as_promotion(x, y, x + 1, y - 1, moves)
361             else:
362                 moves.append((x, y, x + 1, y - 1))
363
364         # diagonal to left for black pawn
365         if piece == -1 and y < SIZE - 1 and x > 0 and self.board[y + 1][x - 1] > 0:
366             if y + 1 == 7:
367                 self.append_move_as_promotion(x, y, x - 1, y + 1, moves)
368             else:
369                 moves.append((x, y, x - 1, y + 1))
370
371         # diagonal to right for black pawn
372         if piece == -1 and y < SIZE - 1 and x < SIZE - 1 and self.board[y + 1][x + 1] > 0:
373             moves.append((x, y, x + 1, y + 1))
374             if y + 1 == 7:
375                 self.append_move_as_promotion(x, y, x + 1, y + 1, moves)
376             else:
377                 moves.append((x, y, x + 1, y + 1))
378
379         ## EN PASSANT
380         move = self.info[5]
381         # print(f"move {move}")
```

```
382         if move != None:
383             if self.is_pawn_move_of_2_sqr(move[0], move[1], move[2], move[3]):
384                 enpassant = self.is_enpassanat_pawn_nearby(x, y, move[2], move[3])
385                 if enpassant[0]:
386                     moves.append((x, y, x + enpassant[1], y - color))
387 
388     def generate_bishop_moves(self, x, y, moves):
389         direction = [(1, 1), (-1, 1), (1, -1), (-1, -1)]
390         is_white = self.is_white_piece(x, y)
391 
392         for dx, dy in direction:
393             for k in range(1, SIZE):
394                 x_move = x + k * dx
395                 y_move = y + k * dy
396                 move = (x, y, x_move, y_move)
397                 if not (x_move >= 0 and y_move >= 0 and x_move < SIZE and y_move < SIZE):
398                     continue
399                 if self.board[y_move][x_move] != 0:
400                     if self.is_white_piece(x_move, y_move) == is_white:
401                         break
402                     else:
403                         moves.append(move)
404                         break
405                 moves.append(move)
406 
407     def generate_knight_moves(self, x, y, moves):
408         direction = [(1, 2), (2, 1), (1, -2), (-1, 2), (-2, 1), (-1, -2), (-2, -1), (2, -1)]
409         is_white = self.is_white_piece(x, y)
410 
411         for dx, dy in direction:
412             x_move = x + dx
413             y_move = y + dy
414             move = (x, y, x_move, y_move)
415             if not (x_move >= 0 and y_move >= 0 and x_move < SIZE and y_move < SIZE):
416                 continue
417             if self.board[y_move][x_move] != 0:
418                 if self.is_white_piece(x_move, y_move) == is_white:
419                     continue
420                 else:
421                     moves.append(move)
422                     continue
423             moves.append(move)
424 
425     def generate_rook_moves(self, x, y, moves):
426         direction = [(1, 0), (-1, 0), (0, -1), (0, 1)]
427         is_white = self.is_white_piece(x, y)
428 
429         for dx, dy in direction:
430             for k in range(1, SIZE):
431                 x_move = x + k * dx
432                 y_move = y + k * dy
433                 move = (x, y, x_move, y_move)
434                 if not (x_move >= 0 and y_move >= 0 and x_move < SIZE and y_move < SIZE):
435                     continue
436                 if self.board[y_move][x_move] != 0:
437                     if self.is_white_piece(x_move, y_move) == is_white:
438                         break
439                     else:
440                         moves.append(move)  ## capture
441                         break
442                 moves.append(move)
443     def generate_queen_moves(self, x, y, moves):
```

```
        direction = [(1, 0), (-1, 0), (0, -1), (0, 1), (1, 1), (-1, 1), (1, -1), (-1, -1)]
        is_white = self.is_white_piece(x, y)

        for dx, dy in direction:
            for k in range(1, SIZE):
                x_move = x + k * dx
                y_move = y + k * dy
                move = (x, y, x_move, y_move)
                if not (x_move >= 0 and y_move >= 0 and x_move < SIZE and y_move < SIZE):
                    continue
                if self.board[y_move][x_move] != 0:
                    if self.is_white_piece(x_move, y_move) == is_white:
                        break
                    else:
                        moves.append(move)  ## capture
                        break
                moves.append(move)
    def generate_king_moves(self, x, y, moves, checking):
        direction = [(1, 0), (-1, 0), (0, -1), (0, 1), (1, 1), (-1, 1), (1, -1), (-1, -1)]
        is_white = self.is_white_piece(x, y)

        for dx, dy in direction:
            x_move = x + dx
            y_move = y + dy
            move = (x, y, x_move, y_move)
            if not (x_move >= 0 and y_move >= 0 and x_move < SIZE and y_move < SIZE):
                continue
            if self.board[y_move][x_move] != 0:
                if self.is_white_piece(x_move, y_move) == is_white:
                    continue
                else:
                    moves.append(move)
                    continue
            moves.append(move)

        if is_white:
            row = 7
        else:
            row = 0

        if not checking:
            if self.queen_castling_rights(is_white):
                if self.is_queen_castling_possible(is_white):
                    moves.append((x, y, 2, row))

            if self.king_castling_rights(is_white):
                if self.is_king_castling_possible(is_white):
                    moves.append((x, y, 6, row))

    def delete_moves_with_check(self, moves, color):
        copy_board = copy.deepcopy(self.board)
        copy_info = copy.deepcopy(self.info)
        valid_moves = []

        new_engine = Engine(copy_board, copy_info)

        for move in moves:
            if len(move) == 5:
                piece, is_white, changes, last2_move = new_engine.move_board(move[0], move[1],
    move[2], move[3],
                                                                             move[4])
            else:
```

```
505                 piece, is_white, changes, last2_move = new_engine.move_board(move[0], move[1],
    move[2], move[3])
506
507             new_engine.update_valid_moves(True)
508
509             if not new_engine.is_check(is_white):
510                 valid_moves.append(move)
511
512             new_engine.undo_move_board(move[0], move[1], move[2], move[3], piece, is_white,
    changes, last2_move)
513             new_engine.update_valid_moves(True)
514
515         return valid_moves
516
517     def valid_moves(self,x,y, checking = False):
518         piece = self.get_figure(x, y)
519         moves = []
520         if piece is None:
521             return moves
522
523         if x < 0 and y < 0 or x >= SIZE or y >= SIZE:
524             return moves
525
526         color = 1 if self.is_white_piece(x,y) else -1
527
528         ## Ruchy pionkiem
529         if abs(piece) == 1:
530             self.generate_pawn_moves(x,y, piece, moves, color)
531
532         ## Ruchy skoczkiem
533         if abs(piece) == 3:
534             self.generate_knight_moves(x, y, moves)
535
536         ## Ruchy go cem
537         if abs(piece) == 4:
538             self.generate_bishop_moves(x, y, moves)
539
540         ## Ruchy wie
541         if abs(piece) == 5:
542             self.generate_rook_moves(x, y, moves)
543
544         ## Ruchy kr low
545         if abs(piece) == 9:
546             self.generate_queen_moves(x, y, moves)
547
548         ## Ruchy kr lem(wraz z roszad )
549         if abs(piece) == 2:
550             self.generate_king_moves(x, y, moves, checking)
551
552         # Usuwanie tych ruch w, po kt rych wykonaniu kr l jest szachowany.
553         if not checking:
554             valid_moves = self.delete_moves_with_check(moves, color)
555
556             return valid_moves
557
558         return moves
559
560     def castle(self, start_x, start_y, end_x, color):
561         if start_x - end_x == 2:  # Castling long
562             self.board[start_y][2] = 2 * color
563             self.board[start_y][4] = 0
564             self.board[start_y][3] = 5 * color
```

```
565             self.board[start_y][0] = 0
566         elif start_x - end_x == -2:  # Castling short
567             self.board[start_y][6] = 2 * color
568             self.board[start_y][4] = 0
569             self.board[start_y][5] = 5 * color
570             self.board[start_y][7] = 0
571 
572     def promotion(self, start_x, start_y, end_x, end_y, color, promotion):
573         self.info[6] = True
574         self.board[end_y][end_x] = promotion * color
575         self.board[start_y][start_x] = 0
576 
577     def enpassant(self, start_x, start_y, end_x, end_y, color, piece_from):
578         self.board[end_y][end_x] = piece_from
579         self.board[start_y][start_x] = 0
580         self.board[end_y + color][end_x] = 0
581         self.info[7] = True
582 
583     def regular_move(self,start_x, start_y, end_x, end_y, piece_from):
584         if piece_from != 0:
585             self.board[end_y][end_x] = piece_from
586             self.board[start_y][start_x] = 0
587 
588     def update_info_castling_rights(self, start_x, end_x, changes, is_white, piece_from, piece_to)
    :
589         if abs(piece_from) == 2:
590 
591             if self.queen_castling_rights(is_white):
592                 self.change_queen_castling_rights(is_white, False)
593                 changes[0] = True  # change of rigths for castling to enable undo_move
594 
595             if self.king_castling_rights(is_white):
596                 self.change_king_castling_rights(is_white, False)
597                 changes[1] = True
598 
599                 # ruch wie
600         if abs(piece_from) == 5:
601             if start_x == 0:  # left_rook
602                 if self.queen_castling_rights(is_white):
603                     self.change_queen_castling_rights(is_white, False)
604                     changes[0] = True
605             if start_x == 7:  # right_rook
606                 if self.king_castling_rights(is_white):
607                     self.change_king_castling_rights(is_white, False)
608                     changes[1] = True
609 
610         # zbicie wie y przez przeciwnika
611         if abs(piece_to) == 5:
612             if end_x == 0:
613                 if self.queen_castling_rights(is_white):
614                     self.change_queen_castling_rights(is_white, False)
615                     changes[0] = True
616             if end_x == 7:
617                 if self.king_castling_rights(is_white):
618                     self.change_king_castling_rights(is_white, False)
619                     changes[1] = True
620     def move_board(self, start_x, start_y, end_x, end_y, promotion = None):
621         is_white = self.is_white_piece(start_x, start_y)
622         piece_from = self.get_figure(start_x, start_y)
623         piece_to = self.get_figure(end_x, end_y)
624 
625         color = 1 if is_white else -1
```

```
626
627         prev_promotion = self.info[6]
628         prev_enpassant = self.info[7]
629         self.info[6] = False # promotion
630         self.info[7] = False # enpassant move
631 
632         prev_50 = self.update_fifty_move_rule(start_x, start_y, end_x, end_y)
633         prev_score = self.info[10]
634 
635         # [queen_castling_rigths, king_castling_rigths, prev_50_move, promotion, enpassant]
636         changes = [False, False, prev_50, prev_promotion, prev_enpassant, prev_score]
637 
638         if self.is_it_capture(end_x, end_y):
639             self.info[10] -= piece_to
640 
641 
642         if self.is_castling(start_x, start_y, end_x, end_y, piece_from):
643             self.castle(start_x, start_y, end_x, color)
644 
645         elif self.is_pawn_promotion(start_x, start_y, end_x, end_y):
646             self.promotion(start_x, start_y, end_x, end_y, color, promotion)
647 
648         elif self.is_enpassant(start_x, start_y, end_x, end_y, piece_from):
649             self.enpassant(start_x, start_y, end_x, end_y, color, piece_from)
650         else:
651             self.regular_move(start_x, start_y, end_x, end_y, piece_from)
652 
653 
654         last2_move = self.info[4]
655         last_move = self.info[5] # now last move is previous move
656         self.info[4] = last_move
657         self.info[5] = (start_x, start_y, end_x, end_y)
658 
659         self.update_info_castling_rights(start_x, end_x, changes, is_white, piece_from, piece_to)
660 
661         return piece_to, is_white, changes, last2_move
662 
663 
664     def update_valid_moves(self,checking = False):
665         self.last_valid_moves_white = self.valid_moves_white
666         self.last_valid_moves_black = self.valid_moves_black
667         self.valid_moves_white = self.all_valid_moves(True,checking)
668         self.valid_moves_black = self.all_valid_moves(False,checking)
669 
670     def update_valid_moves_white(self,checking = False):
671         self.last_valid_moves_white = self.valid_moves_white
672         self.valid_moves_white = self.all_valid_moves(True,checking)
673 
674     def update_valid_moves_black(self,checking = False):
675         self.last_valid_moves_black = self.valid_moves_black
676         self.valid_moves_black = self.all_valid_moves(False,checking)
677 
678     def undo_update_valid_moves(self):
679         self.valid_moves_white = self.last_valid_moves_white
680         self.valid_moves_black = self.last_valid_moves_black
681         self.last_valid_moves_white = None
682         self.last_valid_moves_black = None
683 
684     def undo_valid_moves_white(self):
685         self.valid_moves_white = self.last_valid_moves_white
686         self.last_valid_moves_white = None
687 
```

```
688     def undo_valid_moves_black(self):
689         self.valid_moves_black = self.last_valid_moves_black
690         self.last_valid_moves_black = None
691 
692     def undo_castlig_right_changes(self, changes, is_white):
693         if changes[0]:
694             self.change_queen_castling_rights(is_white, True)
695         if changes[1]:
696             self.change_king_castling_rights(is_white, True)
697 
698     def uncastle(self, start_x, start_y, end_x, end_y, color):
699         if start_x - end_x == 2:  # Castling long
700             self.board[start_y][0] = 5 * color
701             self.board[start_y][2] = 0
702             self.board[start_y][3] = 0
703             self.board[start_y][4] = 2 * color
704         elif start_x - end_x == -2:  # Castling short
705             self.board[start_y][4] = 2 * color
706             self.board[start_y][5] = 0
707             self.board[start_y][6] = 0
708             self.board[start_y][7] = 5 * color
709 
710     def undo_enpassant(self,  start_x, start_y, end_x, end_y, color, piece_from):
711         self.board[end_y][end_x] = 0
712         self.board[start_y][start_x] = piece_from
713         self.board[end_y + color][end_x] = -color
714 
715     def undo_regular_move(self, start_x, start_y, end_x, end_y, color, piece_from, piece):
716         piece_from = self.get_figure(end_x, end_y)
717         if piece_from != 0:
718             self.board[start_y][start_x] = piece_from
719             self.board[end_y][end_x] = piece
720     def undo_move_board(self, start_x, start_y, end_x, end_y, piece, is_white,changes, last2_move)
    :
721         self.undo_castlig_right_changes(changes, is_white)
722 
723         self.info[5] = self.info[4] ## cofanie aktualnego ruchu na ostatni ruchu
724         self.info[4] = last2_move
725 
726 
727         color = 1 if is_white else -1
728 
729         piece_from = self.get_figure(end_x, end_y)
730 
731         if self.is_castling(start_x, start_y, end_x, end_y, piece_from):
732             self.uncastle(start_x, start_y, end_x, end_y, color)
733         elif self.info[7]: # enpssant
734             self.undo_enpassant(start_x, start_y, end_x, end_y, color, piece_from)
735         else:
736            self.undo_regular_move(start_x, start_y, end_x, end_y, color, piece_from, piece)
737 
738         self.info[6] = changes[3] # cofanie promocji
739         self.info[7] = changes[4] # cofanie enpassant
740         self.info[9] = changes[2] # cofanie zasady 50 ruchow
741         self.info[10] = changes[5] # cofanie score
742 
743     def is_white_piece(self, x, y):
744         piece = self.get_figure(x, y)
745 
746         if piece > 0 :  return True
747         elif piece < 0 : return False
748         return None
```

```
749
750     def get_figure(self, x, y):
751         if x == None or y == None or x >= SIZE or y >= SIZE:
752             return None
753 
754         figure = self.board[y][x]
755         return figure
```

Listing 33. Moduł dragger stworzonej aplikacji.

```
1  import board
2  from Const import *
3  class Dragger:
4      def __init__(self, figures, canvas, board, game):
5          self.figures = figures
6          self.board = board
7          self.canvas = canvas
8          self.game = game
9  
10         self.drag_data = {"x": 0, "y": 0, "item": None}
11         self.initial_position = {"x": 0, "y": 0, "item": None}
12         self.initial_index = [None,None]
13         self.figure = 0
14 
15     def calculate_board_position(self, x ,y):
16        if x < 0 or y < 0 or x >= BOARD_WIDTH or y >= BOARD_HEIGHT:
17            return None,None
18 
19        x_index = int (x // SPACE_SIZE)
20        y_index = int (y // SPACE_SIZE)
21 
22        return x_index,y_index
23 
24     def drag_start(self, event):
25         if not self.board.game_on:
26             return
27 
28         if settings["AI"] and self.board.white_turn == settings["TURN"] or not settings["AI"]:
29             self.board.dehighlight_valid_moves()
30             x_board,y_board = self.calculate_board_position(event.x, event.y)
31 
32             if self.board.engine.is_white_piece(x_board, y_board) == self.board.white_turn:
33                 self.drag_data["item"] = self.figures.canvas_images[y_board][x_board]
34 
35             if self.drag_data["item"]:
36                 self.drag_data["x"] = event.x
37                 self.drag_data["y"] = event.y
38                 self.initial_index = [x_board, y_board]
39                 self.figure = self.board.engine.get_figure(x_board, y_board)
40 
41                 # centrowanie figury po klikni ciu
42                 x, y = self.canvas.coords(self.drag_data["item"])
43                 delta_x = x - event.x
44                 delta_y = y - event.y
45                 self.initial_position = {'x': x, 'y': y, 'item': self.drag_data["item"]}
46                 self.canvas.move(self.drag_data["item"], -delta_x, -delta_y)
47 
48                 valid_moves = self.board.engine.valid_moves(x_board, y_board)
49                 self.board.highlight_valid_moves(valid_moves)
50 
51 
52     def drag_motion(self, event):
```

```
        if self.drag_data["item"] is not None:
            delta_x = event.x - self.drag_data["x"]
            delta_y = event.y - self.drag_data["y"]

            self.canvas.move(self.drag_data["item"], delta_x, delta_y)

            self.drag_data["x"] = event.x
            self.drag_data["y"] = event.y

    def drag_stop(self, event):
        if self.drag_data["item"] is not None:
            x_start, y_start = self.calculate_board_position(self.initial_position["x"], self.
    initial_position["y"])
            delta_x = self.drag_data["x"] - self.initial_position["x"]
            delta_y = self.drag_data["y"] - self.initial_position["y"]
            x_end, y_end = self.calculate_board_position(self.drag_data["x"], self.drag_data["y"])
            if x_end is None or y_end is None:
                self.canvas.move(self.drag_data["item"], -delta_x, -delta_y)
                self.drag_data["item"] = None
                return

            if x_start == x_end and y_start == y_end:
                self.canvas.move(self.drag_data["item"], -delta_x, -delta_y)
                self.drag_data["item"] = None
                return

            if not self.board.engine.is_valid_move(x_start, y_start, x_end, y_end):
                self.canvas.move(self.drag_data["item"], -delta_x, -delta_y)
                self.drag_data["item"] = None
                print("Not valid move")
                return


            self.game.update(self.drag_data, x_start, y_start, x_end, y_end)

            print(self.board.board)
            self.drag_data["item"] = None
```

Listing 34. Moduł board stworzonej aplikacji.

```
from tkinter.constants import BOTTOM
from tkinter import *
from Const import *
from engine import *
import numpy as np

class Board:
    def __init__(self, frame, game):
        self.game_on = True
        self.white_turn = True
        self.history = [] # historia jest w formacie LAN
        self.result = None
        self.threefold_repetition = 0
        self.fifty_move_rule = 0
        self.white_queen_castling_right = True
        self.black_queen_castling_right = True
        self.white_king_castling_right = True
        self.black_king_castling_right = True
        self.score = 0

        move = None
        last_move = None
```

```
23          promotion = False
24          enpassant = None
25          self.promotion_choice = None
26          self.info = [self.white_queen_castling_right, self.white_king_castling_right, self.
        black_queen_castling_right, self.black_king_castling_right,  last_move, move, promotion,
        enpassant, self.threefold_repetition, self.fifty_move_rule, self.score]
27
28          self.game = game
29          self.frame = frame
30          self.canvas_board = Canvas(self.frame, width=BOARD_WIDTH, height=BOARD_HEIGHT)
31          self.canvas_label1 = Canvas(self.frame, width=LABEL_VERTUCAL_WIDTH, height=
        LABEL_VERTICAL_HEIGHT)
32          self.canvas_label2 = Canvas(self.frame, width=LABEL_VERTICAL_HEIGHT, height=
        LABEL_VERTUCAL_WIDTH)
33
34          self.squares = [[None for _ in range(SIZE)] for _ in range(SIZE)]
35          self.squares_highlighted = [[None for _ in range(SIZE)] for _ in range(SIZE)]
36
37          self.display_board()
38          self.display_labels()
39
40          self.board = [
41              [-5, -3, -4, -9, -2, -4, -3, -5],
42              [-1, -1, -1, -1, -1, -1, -1, -1],
43              [0, 0, 0, 0, 0, 0, 0, 0],
44              [0, 0, 0, 0, 0, 0, 0, 0],
45              [0, 0, 0, 0, 0, 0, 0, 0],
46              [0, 0, 0, 0, 0, 0, 0, 0],
47              [1, 1, 1, 1, 1, 1, 1, 1],
48              [5, 3, 4, 9, 2, 4, 3, 5]
49          ]
50
51          self.engine = Engine(self.board, self.info)
52
53      def display_board(self):
54          for row in range(0, SIZE):
55              y1 = row * SPACE_SIZE
56              y2 = y1 + SPACE_SIZE
57              for col in range(0, SIZE):
58                  if((row + col) %2 == 0):
59                      color = SQUARES_COLORS[0]
60                  else:
61                      color = SQUARES_COLORS[1]
62                  x1 = col * SPACE_SIZE
63                  x2 = x1 + SPACE_SIZE
64                  item = self.canvas_board.create_rectangle(x1,
65                                                            y1,
66                                                            x2,
67                                                            y2,
68                                                            fill=color)
69                  self.squares[row][col] = item
70
71          self.canvas_board.place(x=50, y=100)
72
73      def display_labels(self):
74
75          for rank in range(8):
76              rank_label = Label(self.canvas_label1, text=str(8 - rank), font=("Arial", 12))
77              letter_height = rank_label.winfo_height()
78              letter_width = rank_label.winfo_width()
79              rank_label.place(x=LABEL_VERTUCAL_WIDTH - 10 * letter_width, y= rank * SPACE_SIZE +
        SPACE_SIZE // 2 - 5 * letter_height)
```

```
80
81          for file in range(8):
82              file_label = Label(self.canvas_label2, text=chr(file + 97), font=("Arial", 12))
83              letter_width = file_label.winfo_width()
84              file_label.place(x=file * SPACE_SIZE + SPACE_SIZE // 2 - 5 * letter_width, y = 0)
85
86          self.canvas_label1.place(x=0, y=100)
87          self.canvas_label2.place(x=50, y=700)
88
89      def draw_game(self):
90          self.result = -1 if self.white_turn else 1
91          self.game_on = False
92
93      def choose_piece(self):
94          popup = Toplevel(self.frame)
95          popup.title("Promocja")
96          popup.geometry("300x200")
97          popup.transient(self.frame)
98
99          label = Label(popup, text="Wybierz figur  do promocji:")
100         label.pack(pady=10)
101
102         self.promotion_choice = None
103         def set_piece(value):
104             self.promotion_choice = value
105             popup.destroy()
106
107         Button(popup, text="Hetman (Q)", command=lambda: set_piece(9)).pack(pady=5)
108         Button(popup, text="Wie a (R)", command=lambda: set_piece(5)).pack(pady=5)
109         Button(popup, text="Goniec (B)", command=lambda: set_piece(4)).pack(pady=5)
110         Button(popup, text="Skoczek (N)", command=lambda: set_piece(3)).pack(pady=5)
111
112         popup.wait_window()
113
114         return self.promotion_choice
115
116     def change_move_to_LAN(self, x_start, y_start, x_end, y_end):
117         move_lan = f"{chr(x_start + 97)}{8 - y_start}{chr(x_end + 97)}{8 - y_end}"
118         return move_lan
119
120     def add_to_history(self,x_start, y_start, x_end, y_end):
121         move_lan = self.change_move_to_LAN(x_start, y_start, x_end, y_end)
122         self.history.append(move_lan)
123
124     def is_Game_On(self):
125         if self.engine.game_over() == None:
126             return True
127
128         else:
129             self.game_on = False
130             self.result = self.engine.game_over()
131             return False
132
133     def is_white_turn(self):
134         if self.white_turn:
135             return True
136         else:
137             return False
138
139     def get_result(self):
140         return self.result
141
```

```
142     def check_game_state(self):
143         if not self.is_Game_On():
144             self.game.handle_game_end()
145 
146     def change_color_of_square(self,x,y, color):
147         item = self.squares[y][x]
148         self.canvas_board.itemconfig(item, fill=color)
149 
150     def highlight_valid_moves(self, valid_moves):
151         for move in valid_moves:
152             x = move[2]
153             y = move[3]
154             self.change_color_of_square(x,y, "red")
155             self.squares_highlighted[y][x] = True
156 
157 
158     def dehighlight_valid_moves(self):
159         for row in range(SIZE):
160             for col in range(SIZE):
161                 if self.squares_highlighted[row][col]:
162                     self.change_color_of_square(col, row, SQUARES_COLORS[(row + col )% 2])
163                     self.squares_highlighted[row][col] = False
164 
165 
166     def highlight_move(self):
167         move = self.info[5]
168 
169         if move != None:
170             self.change_color_of_square(move[0], move[1], "blue")
171             self.change_color_of_square(move[2], move[3], "yellow")
172 
173 
174     def dehighlight_last_move(self):
175         last_move = self.info[4]
176 
177         if last_move != None:
178             self.change_color_of_square(last_move[0], last_move[1], SQUARES_COLORS[(last_move[0] +
     last_move[1] )% 2])
179             self.change_color_of_square(last_move[2], last_move[3], SQUARES_COLORS[(last_move[2] +
     last_move[3])% 2])
```

Listing 35. Moduł ai stworzonej aplikacji.

```
1  import random
2  import copy
3  from Const import *
4  class Ai:
5      def __init__(self):
6          self.best_move = None
7          self.iteration = 0
8  
9      def negamax(self, depth, color, engine_copy, alpha, beta, moves):
10         if depth == 0:
11             return color * self.evaluate(engine_copy), self.best_move
12 
13         for_white = True if color == 1 else False
14         max_eval = float('-inf')
15 
16         moves = self.order_moves(moves, engine_copy)
17         for move in moves:
18             start_x, start_y, end_x, end_y, *promotion = move
19             promotion_type = promotion[0] if promotion else None
```

```
20
21              piece, is_white, changes, last2_move = engine_copy.move_board(start_x, start_y, end_x,
     end_y, promotion_type)
22              engine_copy.update_valid_moves(True)
23
24              next_moves  = engine_copy.valid_moves_black if for_white is True else engine_copy.
    valid_moves_white
25
26              evaluation, _ = self.negamax( depth - 1, -color, engine_copy, -beta, -alpha,
    next_moves)
27              evaluation = -evaluation
28
29              engine_copy.undo_move_board(start_x, start_y, end_x, end_y, piece, is_white, changes,
    last2_move)
30              engine_copy.update_valid_moves(True)
31
32              if evaluation > max_eval:
33                  max_eval = evaluation
34                  if depth == DEPTH:
35                      self.best_move = move
36
37              self.iteration += 1
38
39              alpha = max(alpha, max_eval)
40              if alpha >= beta:
41                  break
42
43          return (max_eval, self.best_move)
44
45      def find_best_move(self, depth, color, engine):
46          engine_copy = copy.deepcopy(engine)
47
48          for_white = True if color == 1 else False
49          valid_moves = engine_copy.all_valid_moves(for_white)
50          score, best_move = self.negamax(depth, color, engine_copy, -MATE, MATE, valid_moves)
51
52          print(f"Liczba  ga   zi {self.iteration}")
53          return best_move, score
54
55      def generate_random_move(self, is_white, engine):
56          valid_moves = engine.all_valid_moves(is_white)
57          if not valid_moves:
58              return None
59
60          random_move = random.choice(valid_moves)
61
62          return random_move
63
64      def order_moves(self, moves, engine_copy):
65          eval_moves = []
66          for move in moves:
67              start_x, start_y, end_x, end_y, *promotion = move
68
69              promotion_bonus = 10 if promotion else 0
70              capture_bonus = engine_copy.get_figure(end_x, end_y)
71
72              eval_moves.append((capture_bonus + promotion_bonus, move))
73
74          # Zwracnie ruch w posortowanych malej co
75          return [move for _, move in sorted(eval_moves, key=lambda x: x[0], reverse=True)]
76
77      def evaluate(self, engine):
```

```
78          material_score = self.evaluate_material(engine)
79          position_score = self.evaluate_position(engine)
80          mobility_score = self.evaluate_mobility(engine)
81
82          return (
83                  material_score * 1.2 +
84                  position_score * 0.5 +
85                  mobility_score * 0.1
86          )
87
88      def evaluate_mobility(self, engine):
89          return len(engine.valid_moves_white) - len(engine.valid_moves_black)
90
91      def evaluate_material(self, engine):
92          piece_values = {1: 1, 2: 100, 3: 3, 4: 3, 5: 5, 9: 9, -1: -1, -2: -100, -3: -3, -4: -3, -
    5: -5, -9: -9}
93          score = 0
94          for col in range(SIZE):
95              for row in range(SIZE):
96                  piece = engine.get_figure(row, col)
97                  if piece != 0:
98                      value = piece_values[piece]
99                      score += value
100         return score
101
102     def piece_square_score(self, piece, x, y, color):
103         if abs(piece) == 1:
104             return self.pawn_table[y][x] if color == 1 else self.pawn_table[7 - y][x]
105
106         elif abs(piece) == 2:
107             return self.king_table_early[y][x] if color == 1 else self.king_table_early[7 - y][x]
108
109         elif abs(piece) == 3:
110             return self.knight_table[y][x] if color == 1 else self.knight_table[7 - y][x]
111
112         elif abs(piece) == 4:
113             return self.bishop_table[y][x]
114
115         elif abs(piece) == 5:
116             return self.rook_table[y][x]
117
118         elif abs(piece) == 9:
119             return self.queen_table[y][x]
120
121     def evaluate_position(self, engine):
122         score = 0
123
124         for col in range(SIZE):
125             for row in range(SIZE):
126                 piece = engine.get_figure(row, col)
127                 piece_abs = abs(piece)
128                 for_white = engine.is_white_piece(row, col)
129                 color = 1 if for_white else -1
130                 position_value = self.piece_square_score(piece, row, col, color)
131                 if piece!= 0:
132                     total_value = piece_abs + position_value
133                     score += total_value if engine.is_white_piece(row, col ) else -total_value
134         return score
135
136     pawn_table = [
137         [0, 0, 0, 0, 0, 0, 0, 0],
138         [5, 5, 5, 5, 5, 5, 5, 5],
```

```
        [1, 1, 2, 3, 3, 2, 1, 1],
        [0.5, 0.5, 1, 2.5, 2.5, 1, 0.5, 0.5],
        [0, 0, 0, 2, 2, 0, 0, 0],
        [0.5, -0.5, -1, 0, 0, -1, -0.5, 0.5],
        [0.5, 1, 1, -2, -2, 1, 1, 0.5],
        [0, 0, 0, 0, 0, 0, 0, 0]
    ]

    knight_table = [
        [-5, -4, -3, -3, -3, -3, -4, -5],
        [-4, -2, 0, 0, 0, 0, -2, -4],
        [-3, 0, 1, 1.5, 1.5, 1, 0, -3],
        [-3, 0.5, 1.5, 2, 2, 1.5, 0.5, -3],
        [-3, 0, 1.5, 2, 2, 1.5, 0, -3],
        [-3, 0.5, 1, 1.5, 1.5, 1, 0.5, -3],
        [-4, -2, 0, 0.5, 0.5, 0, -2, -4],
        [-5, -4, -3, -3, -3, -3, -4, -5]
    ]

    bishop_table = [
        [-2, -1, -1, -1, -1, -1, -1, -2],
        [-1, 0, 0, 0, 0, 0, 0, -1],
        [-1, 0, 1, 1, 1, 1, 0, -1],
        [-1, 0, 1, 2, 2, 1, 0, -1],
        [-1, 0, 1, 2, 2, 1, 0, -1],
        [-1, 0, 1, 1, 1, 1, 0, -1],
        [-1, 0, 0, 0, 0, 0, 0, -1],
        [-2, -1, -1, -1, -1, -1, -1, -2]
    ]

    rook_table = [
        [0, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 1, 1, 1, 1, 1, 0],
        [0, 1, 2, 2, 2, 2, 1, 0],
        [0, 1, 2, 3, 3, 2, 1, 0],
        [0, 1, 2, 3, 3, 2, 1, 0],
        [0, 1, 2, 2, 2, 2, 1, 0],
        [0, 0, 1, 1, 1, 1, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0]
    ]

    queen_table = [
        [-2, -1, -1, 0, 0, -1, -1, -2],
        [-1, 0, 1, 1, 1, 1, 0, -1],
        [-1, 1, 1, 2, 2, 1, 1, -1],
        [0, 1, 2, 2, 2, 2, 1, 0],
        [0, 1, 2, 2, 2, 2, 1, 0],
        [-1, 0, 1, 1, 1, 1, 0, -1],
        [-2, -1, -1, 0, 0, -1, -1, -2],
        [-2, -1, -1, 0, 0, -1, -1, -2]
    ]

    king_table_early = [
        [-3, -4, -4, -5, -5, -4, -4, -3],
        [-3, -4, -4, -5, -5, -4, -4, -3],
        [-3, -4, -4, -5, -5, -4, -4, -3],
        [-3, -4, -4, -5, -5, -4, -4, -3],
        [-2, -3, -3, -4, -4, -3, -3, -2],
        [-2, -3, -3, -4, -4, -3, -3, -2],
        [-1, -2, -2, -3, -3, -2, -2, -1],
        [2, 2, 0, 0, 0, 0, 2, 2]
    ]
```

```
201
202     king_table_endgame = [
203         [-50, -40, -30, -20, -20, -30, -40, -50],
204         [-30, -20, -10, 0, 0, -10, -20, -30],
205         [-30, -10, 10, 20, 20, 10, -10, -30],
206         [-30, -10, 20, 30, 30, 20, -10, -30],
207         [-30, -10, 20, 30, 30, 20, -10, -30],
208         [-30, -10, 10, 20, 20, 10, -10, -30],
209         [-30, -20, -10, 0, 0, -10, -20, -30],
210         [-50, -40, -30, -20, -20, -30, -40, -50]
211     ]
```

Listing 36. Moduł Const stworzonej aplikacji.

```
1  BOARD_WIDTH = 600
2  BOARD_HEIGHT = 600
3  GAME_MENU_1WIDTH = 650
4  GAME_MENU_1HEIGHT = 100
5  GAME_MENU_2WIDTH = 300
6  GAME_MENU_2HEIGHT = 750
7  LABEL_VERTUCAL_WIDTH = 50
8  LABEL_VERTICAL_HEIGHT = 600
9
10 GAME_WIDTH = GAME_MENU_1WIDTH + GAME_MENU_2WIDTH
11 GAME_HEIGHT = GAME_MENU_2HEIGHT
12 SIZE = 8
13 SPACE_SIZE = BOARD_WIDTH/SIZE
14 SQUARES_COLORS = ["white", "green"]
15
16 WHITE = True
17 BLACK = False
18
19 settings = {
20     "AI": False,
21     "TURN": True
22 }
23
24 SMALL_PIECE_SIZE = 20
25 PIECE_SIZE = 60
26 DEPTH = 3
27 MATE = 900
28 DRAW = 0
```