Wrocław, 06. czerwca 2006

Automatyczne Pozyskiwanie Wiedzy
+
Sieci Neuronowe 2

Politechnika Wrocławska
Wydział Informatyki i Zarządzania
IV rok studiów

Zastosowanie technik sztucznej inteligencji w realizacji komputerowej gry w szachy

(projekt)

Autor dokumentu: **STAWARZ Paweł**
Indeks: **125939**

Prowadzący: **prof. Halina Kwaśnicka**
dr Urszula Markowska-Kaczmar

Spis treści

WPROWADZENIE **3**

APLIKACJA TESTOWA **4**

OPIS FUNKCJONALNOŚCI 4
MODUŁ ŚWIATA SZACHOWEGO 7
ChessBoard *7*
Piece *8*
Move *8*
Player *8*

TD-LEARNING W SZACHACH **9**

WSTĘP 9
REALIZACJA PROBLEMU 11
WYNIKI BADAŃ 15
Eksperyment 1 ..... *16*
Eksperyment 2 ..... *16*
WNIOSKI 17

POZYSKIWANIE WIEDZY Z SZACHOWYCH BAZ DANYCH **18**

WSTĘP 18
REALIZACJA PROBLEMU 19
WYNIKI BADAŃ 26
WNIOSKI 26

LITERATURA **27**

Wprowadzenie

Głównym celem realizowanego projektu jest wstępne ukierunkowanie działań w stronę pracy dyplomowej pt. „Zastosowanie technik sztucznej inteligencji w komputerowej realizacji gry w szachy", której promotorem jest Pani Dr Urszula Markowska-Kaczmar.

Realizacja dobrze grającego programu szachowego jest z pewnością nie lada wyzwaniem. Złożoność tej królewskiej gry jeszcze długo będzie dawać się we znaki wszystkim śmiałkom, którzy będą chcieli w pełni zautomatyzować wygrywającą strategię. Niemożność ogarnięcia w skończonym czasie niemalże nieskończonych możliwości zarówno strategicznych jak i taktycznych tej gry tym bardziej pobudza do działania – chciałoby się odnaleźć przynajmniej część optymalnej strategii gry.

Na przestrzeni ostatnich kilkudziesięciu lat próbowano już wieloma sposobami zautomatyzować grę w szachy. Zauważalną tendencją gwarantującą wysoki poziom gry komputerów szachowych było zwiększanie szybkości oraz głębokości przeszukiwania drzewa gry w dowolnej pozycji szachowej, które najczęściej uzyskiwano coraz to wymyślniejszymi realizacjami sprzętowymi. Nie zmieniło to jednak faktu, iż określenie jedynie słusznej heurystyki oceniającej bieżący stan gry jest dość kłopotliwe. Problemem jest nawet ustalenie najważniejszych cech gry, które mają być istotne z punktu widzenia oceny odpowiedniej pozycji. Jeszcze trudniejsze okazuję się być określenie ważności wybranych własności, ponieważ jedne mogą być znaczące w początkowej fazie gry, inne – w fazie końcowej, a jeszcze inne – w fazie środkowej. Z tego też względu bardzo często rozkłada się grę w szachy na czynniki pierwsze – każda faza gry poddawana jest odrębnej analizie.

W niniejszym projekcie zostaną rozpoczęte prace nad środkową oraz końcową fazą gry. W ramach środkowej fazy gry za pomocą sztucznej sieci neuronowej utworzony zostanie szkielet funkcji oceniającej, natomiast w ramach końcowej fazy gry wypróbowana zostanie technika wydobywania wiedzy z określonej bazy końcówek szachowych.

W początkowej fazie projektu zostanie zaimplementowana aplikacja testująca, w ramach której zrealizowany będzie pierwszy duży krok milowy – model świata szachowego. Zadaniem modułu odpowiedzialnego za ten model będzie przechowywanie aktualnego stanu gry i generowanie dla niego wszystkich prawidłowych posunięć.

W dalszej fazie projektu zostanie zaprojektowana wspomniana wcześniej sztuczna sieć neuronowa, która uczona będzie metodą TD-learning na podstawie rozgrywanych partii z algorytmem losowo generującym ruchy.

W ostatniej fazie projektu wygenerowane zostanie drzewo decyzyjne klasyfikujące końcówki typu król i wieża przeciwko królowi, które następnie wykorzystane będzie do utworzenia strategii gry wygrywającej w tego typu końcówkach.

Aplikacja testowa

W niniejszym rozdziale opisana zostanie aplikacja, która wykorzystywana jest do testowania różnych strategii gry oraz do wykonywania eksperymentów związanych z uczeniem sztucznej sieci neuronowej metodą TD-learning.

Opis funkcjonalności

Aplikację testową „Neural Chess” zaprojektowano z myślą o indywidualnym testowaniu wszelkich algorytmów grających w szachy oraz hurtowym testowaniu sieci neuronowej w ramach jej uczenia metodą TD-learning. W ramach indywidualnego testowania możliwe jest wybranie dwóch dowolnych algorytmów grających przeciwko sobie, bądź wybranie dowolnego algorytmu grającego w bezpośredniej grze z użytkownikiem. Na Rysunku 1. przedstawiono zrzut ekranu omawianej aplikacji tuż po wykonaniu eksperymentu uczenia sztucznej sieci neuronowej.

Rysunek 1. Zrzut ekranu aplikacji testowej tuż po wykonaniu eksperymentu uczenia metodą TD(λ).

W lewym górnym rogu aplikacji znajduje się szachownica, na której znajduje się aktualny rozkład figur. Dzięki niej możliwy jest bieżący podgląd gry ustalonych algorytmów grających. Na Rysunku 1. znajduje się pozycja końcowa (mat) setnej gry sztucznej sieci neuronowej z algorytmem losowo generującym ruchy. W dowolnej chwili czasu możliwe jest rozgrywanie co najwyżej jednej partii.

W prawym górnym rogu znajduje się panel do definiowania indywidualnych testów algorytmów grających oraz określenia dodatkowych parametrów związanych z prezentowaną na szachownicy grą – na Rysunku 1. aktywny jest tylko przycisk ‘Odwróć szachownicę’ oraz pole tekstowe ‘Czas odpowiedzi:’.

Przycisk **`Odwróć szachownicę`** jest zawsze dostępny i umożliwia zmianę perspektywy (ze strony białych lub czarnych), z której obserwowany jest układ figur na szachownicy. Po uruchomieniu aplikacji szachownica przedstawiana jest domyślnie ze strony figur białych.

Pole tekstowe **‘Czas odpowiedzi:’** aktywne jest w przypadku indywidualnego testu dwóch algorytmów grających bądź w przypadku wykonywania eksperymentu. Pole to pozwala definiować minimalny czas pomiędzy kolejnymi posunięciami w grze.

Listy rozwijane **‘Białe:’** oraz **‘Czarne:’** aktywne są w przypadku gdy nie wykonywany jest ani test indywidualny ani eksperyment i służą do definiowania graczy odpowiednio koloru białego oraz czarnego w celu przeprowadzenia indywidualnego testu (rozegrania jednej partii).

Lista rozwijana **‘Pozycja początkowa:’** aktywna jest w identycznych przypadkach jak opisane wyżej listy i pozwala wybrać typ pozycji początkowej do testu indywidualnego. Możliwy jest wybór pozycji wyjściowej, a więc takiej, od której zawsze rozpoczyna się nową grę w szachy, albo tzw. pozycji losowej KRK, w której w sposób całkowicie losowy rozmieszczone są trzy figury: biały król, biała wieża oraz czarny król – przy założeniu, że pierwszy ruch wykonują białe, a więc czarny król nie może być szachowany przez białą wieżę i oczywiście oba króle nie mogą znajdować się obok siebie, tzn. nie mogą się nawzajem „szachować”.

Przyciski **‘Start’**, **‘Pauza’** oraz **‘Stop’** służą do zarządzania testem indywidualnym. W przypadku gdy nie wykonywany jest ani test indywidualny ani eksperyment, aktywny jest jedynie przycisk ‘Start’, który pozwala rozpocząć test indywidualny dla odpowiednio zdefiniowanych parametrów ‘Białe’, ‘Czarne’, ‘Pozycja początkowa’ oraz ewentualnie ‘Czas odpowiedzi’. Po rozpoczęciu testu indywidualnego przycisk ‘Start’ przestaje być aktywny, natomiast aktywne stają się przyciski ‘Pauza’ oraz ‘Stop’. Przycisk ‘Pauza’ pozwala w dowolnej chwili przeprowadzanego testu indywidualnego zatrzymać tymczasowo dalszą rozgrywkę (żaden algorytm grający nie może wówczas wykonać ruchu). W tej sytuacji przycisk ‘Pauza’ przestaje być aktywny, natomiast aktywne stają się przyciski ‘Start’ oraz ‘Stop’. Dalsze wznowienie gry możliwe jest poprzez aktywowany przycisk ‘Start’. W każdym momencie od rozpoczęcia testu indywidualnego możliwe jest jego całkowite anulowanie poprzez przycisk ‘Stop’ – wówczas niemożliwe staje się wznowienie zatrzymanej gry i możliwe jest jedynie zdefiniowanie kolejnego testu indywidualnego.

Poniżej omawianego panelu kontrolnego znajduje się panel informacyjny, który podczas testu indywidualnego, jak również podczas wykonywania eksperymentu, wyświetla bieżące informacje na temat

rozgrywanej partii, obserwowanej na szachownicy. W panelu tym po każdym ruchu odświeżane są informacje o ostatnim wykonanym ruchu, liczbie ruchów wykonanych od początku partii oraz liczbie ruchów wykonanych od ostatniego bicia. Dodatkowo na zakończenie partii wyświetlany jest jej ostateczny wynik. W ramach informacji o ostatnim ruchu przyjęto notację, w której pierwsza litera oznacza odpowiednią figurę określonego koloru, po której następuje odwołanie do pola, z którego figura się ruszyła, a następnie po myślniku – do pola, na którym stanęła. Ustalono, że wielka litera odnosi się do figury białej, natomiast mała litera – do figury czarnej i wykorzystano następujące oznaczenia: 'p' – pionek, 'r' – wieża, 'n' – skoczek, 'b' – goniec, 'k' – król, 'q' – hetman. Zgodnie z tymi założeniami wynika, że na Rysunku 1. przedstawiona jest informacja o tym, iż ostatni ruch wykonany został białym pionkiem z pola h7 na pole h8, a ponieważ na tym polu tak naprawdę stoi hetman, więc łatwo się domyślić, że ów pionek dochodząc do linii przemiany został przemianowany właśnie na hetmana. Liczbę ruchów wykonaną od początku partii przyjęto oznaczać połówką w przypadku gdy ostatnim ruchem był ruch białych, natomiast całością gdy ostatnim ruchem był ruch czarny. Jak widać na Rysunku 1. widnieje informacja o tym, że ostatnio wykonano 75. ruch białych i ostatecznym wynikiem partii jest wygrana białych (1 : 0).

Poniżej szachownicy znajduje się panel umożliwiający przeprowadzenie eksperymentu dla zadanych w nim parametrów. Panel ten jest całkowicie nieaktywny w przypadku przeprowadzania testu indywidualnego.

Możliwe jest ustalenie stałych wartości dla współczynników wykorzystywanych w procesie uczenia. Pole tekstowe oraz suwak **'alpha:'** pozwalają ustalić wartość współczynnika uczenia a na dowolną wartość rzeczywistą z przedziału $[0;1]$. Pole tekstowe oraz suwak **'beta:'** pozwalają ustalić wartość współczynnika b (stromość sigmoidy funkcji aktywacji neuronu) na dowolną wartość rzeczywistą z przedziału $[0;10]$. Pole tekstowe oraz suwak **'lambda:'** pozwalają ustalić wartość współczynnika l (główny parametr metody TD(λ)) na dowolną wartość rzeczywistą z przedziału $[0;1]$.

Na prawo od panelu definiującego wartości współczynników znajduje się panel umożliwiający ustalanie odpowiedniej konfiguracji treningu (eksperymentu). Lista rozwijana **'Przeciwnik:**' pozwala określić przeciwnika, z którym uczona sieć neuronowa będzie grała kolejne partie. Przeciwnikiem tym może być także użytkownik ('Gracz'). Pole tekstowe **'Liczba gier:'** pozwala zdefiniować długość treningu, tzn. liczbę partii, które zostaną rozegrane podczas wykonywania eksperymentu. Pole wyboru **'Kolory na przemian:'** pozwala określić czy w kolejnych grach wybrane algorytmy będą grały różnymi kolorami (w przypadku opcji 'Tak') czy też tymi samymi kolorami przez cały czas trwania treningu (w przypadku opcji 'Nie'). W bieżącej wersji aplikacji testowej możliwe jest jedynie przeprowadzenie eksperymentu, w którym uczona sieć neuronowa zawsze gra kolorem białym.

Po zdefiniowaniu wszystkich potrzebnych parametrów eksperymentu możliwe jest rozpoczęcie treningu poprzez przycisk **'Rozpocznij trening'**. W trakcie trwania treningu niemożliwa jest już zmiana ustalonych wartości parametrów.

Poniżej paneli konfiguracyjnych znajduje się panel informacyjny odświeżający po każdej rozegranej partii wyniki uczonej sieci neuronowej. Panel dostarcza informacji o liczbie zakończonych do tej pory partii

(**'Rozegrane:'**) wraz z liczbą odniesionych zwycięstw **('Wygrane:'**), uzyskanych remisów (**'Zremisowane:'**) oraz poniesionych porażek **('Przegrane:'**). Po prawej stronie znajduje się dodatkowo informacja o liczbie zdobytych punktów (**'Liczba punktów:'**) wraz z wykresem przedstawiającym historię wyników uczącej się sieci. Wykres koloru czerwonego przedstawia udział procentowy punktów zdobytych po kolejnych partiach. Wykres koloru szarego przedstawia udział procentowy punktów zdobytych w przeciągu ostatnich dziesięciu partii. Na Rysunku 1. przedstawiono sytuację, w której eksperyment dobiegł do końca, ponieważ rozegrano już 100 partii na 100 zdefiniowanych w **'Liczba gier:'**. Uczona sieć neuronowa zdołała wygrać 37 razy, zremisować 62 razy oraz 1 raz przegrać, zdobywając w ten sposób 68 punktów (1 punkt za zwycięstwo, 0.5 punktu za remis, 0 punktów za przegraną).

W dowolnym momencie przeprowadzania eksperymentu możliwe jest jego przerwanie poprzez przycisk **'Przerwij trening'**, po którym można już jedynie zdefiniować dane nowego treningu.

Moduł świata szachowego

Moduł świata szachowego został zaimplementowany przy pomocy klas **ChessBoard**, **Move**, **Piece**, **Pawn**, **Knight**, **Bishop**, **Rook**, **King**, **Queen**, **Player**, **ComputerPlayer**, **HumanPlayer**, **KingRefugeePlayer**, **KRKPlayer**, **TDLambdaPlayer**.

Kody źródłowe tych klas znajdują się w katalogach 'chessrules', 'chessrules/pieces' oraz 'chessrules/players'.

ChessBoard

Klasa **ChessBoard** dostarcza wielu funkcji związanych stricte z obsługą szachownicy. Obiekt tej klasy przechowuje m.in. dane dotyczące rozmieszczenia figur oraz dane grających „zawodników". Udostępnione funkcje pozwalają generować wszystkie możliwe (prawidłowe!) posunięcia dla strony będącej na ruchu, sprawdzać występowania szacha bądź mata, wykonywać jeden ze wskazanych (prawidłowych!) ruchów, itp. Wszystkie możliwe ruchy generowane są na podstawie funkcji dostarczonych przez zdefiniowane figury, które muszą zapewnić poprawność ich wykonywania:

- każda figura ma ściśle zdefiniowany sposób poruszania – szczególne własności pod tym względem ma pionek, który porusza się o jedno pole do przodu, ale z linii wyjściowej może wykonać ruch o dwa pola do przodu; po dojściu do linii przemiany (ostatni wiersz, do którego może dojść pionek) pionek zamieniany jest na dowolną inną figurę (hetman, goniec, skoczek, wieża); wśród pionków występuje również tzw. bicie w przelocie, które zachodzi tylko i wyłącznie w ściśle określonych warunkach;
- żadna figura z wyjątkiem skoczka nie może przeskoczyć żadnej innej figury;
- każda figura może zbić tylko i wyłącznie figurę koloru przeciwnego;
- strona wykonująca ruch nie może znajdować się w szachu po jego wykonaniu;
- roszadę można wykonać tylko i wyłącznie w przypadku gdy król wraz z roszowaną wieżą nie wykonali dotychczas żadnych innych ruchów oraz król nie znajduje się w szachu ani żadne pole, po których przechodzi król w trakcie roszady nie jest szachowane;

- partia jest rozstrzygnięta (wygrana) w przypadku ustawienia mata, czyli sytuacji w której strona szachowana nie może wykonać żadnego obronnego ruchu oraz nierozstrzygnięta (remis) w przypadku gdy strona będąca na ruchu nie znajduje się w szachu i nie może wykonać żadnego poprawnego ruchu (pat) lub w sytuacji gdy w ciągu 50 ruchów od ostatniego bicia nie doszło do rozstrzygnięcia.

Piece

Klasa **Piece** jest abstrakcyjną klasą specyfikującą wspólne własności figur szachowych. Każda figura definiowana jest poprzez odrębną klasę będącą specjalizacją klasy **Piece**. Każdy obiekt klasy reprezentujący konkretną figurę przechowuje informacje o tym z którą szachownicą jest ona związana, na jakim polu się znajduje oraz jakiego jest koloru. Każda klasa figury musi dostarczyć definicje co najmniej dwóch metod: sprawdzającej czy dana figura szachuje króla przeciwnika oraz generującej wektor możliwych (prawidłowych!) posunięć.

Specjalizacjami klasy **Piece** są klasy:

- **Pawn** – specyfikacja pionka,
- **Knight** – specyfikacja skoczka,
- **Bishop** – specyfikacja gońca,
- **Rook** – specyfikacja wieży,
- **King** – specyfikacja króla,
- **Queen** – specyfikacja hetmana.

Move

Klasa **Move** jest właściwie pomocniczą strukturą danych definiującą określony ruch poprzez dwa punkty (skąd – dokąd) i ewentualnie informację o nowej figurze w przypadku dojścia przez pionka do linii przemiany.

Player

Klasa **Player** jest abstrakcyjną klasą specyfikującą podstawowe własności graczy szachowych. Każda klasa dostarczająca nową strategię gry będzie specjalizacją klasy **Player** udostępniającą definicję tej strategii w odpowiedniej metodzie wykonującej ruch na szachownicy. Każdy obiekt gracza będzie przechowywał informacje o szachownicy, na której wykonuje ruch, oraz o kolorze figur, którymi porusza się na tej szachownicy.

Specjalizacjami klasy **Player** są klasy:

- **HumanPlayer** – klasa umożliwiająca grę użytkownikowi, która obsługuje zdarzenia wywoływane przez myszkę,
- **ComputerPlayer** – klasa specyfikująca najprostszą strategię, w której wykonywane jest wybrane losowo posunięcie ze zbioru wszystkich możliwych ruchów,

- **KingRefugeePlayer** – klasa specyfikująca prostą strategię, w której wykonywane jest wybrane losowo posunięcie ze zbioru wszystkich możliwych ruchów, które utrzymują własnego króla jak najbliżej centrum szachownicy,
- **KRKPlayer** – klasa specyfikująca wygrywającą strategię w końcówkach typu KRK (*King-Rook-King*), czyli w końcówkach król z wieżą przeciwko królowi,
- **TDLambdaPlayer** – klasa specyfikująca strategię opartą na sztucznej sieci neuronowej uczonej metodą TD(λ).

TD-learning w szachach

W niniejszej części projektu utworzony zostanie szkielet sztucznej sieci neuronowej, uczonej metodą TD(λ), pełniącej rolę funkcji oceniającej aktualną pozycję szachową, która będzie wykorzystywana do oceny liści wyprowadzanego drzewa gry w celu wybrania najlepszego (z punktu widzenia tej funkcji) ruchu ze zbioru wszystkich możliwych ruchów.

Wstęp

Metoda uczenia TD(λ) została sformułowana w 1988 roku przez Suttona jako nowa klasa przyrostowych procedur uczących ukierunkowanych na predykcję. W pracy [1] Sutton opisuje, że owa predykcja pewnych przyszłych zachowań/wyników całkowicie nieznanego systemu bazuje jedynie na przeszłych doświadczeniach. W odróżnieniu od dotychczasowych klasycznych metod uczenia, metoda TD(λ) nie porównuje bieżącej predykcji z właściwą odpowiedzią dla zadanego wzorca, tylko porównuje kolejne predykcje aż do momentu uzyskania właściwej odpowiedzi – stąd też jego nazwa *temporal-difference-learning*. Zasadniczą różnicą w tej metodzie uczenia jest założenie, że właściwa odpowiedź systemu jest obserwowalna dopiero po określonej sekwencji wektorów wejściowych. Dlatego też pomiędzy kolejnymi wejściami wykorzystywane są tylko i wyłącznie predykcje uczącej się sztucznej sieci neuronowej. We wspomnianej pracy Sutton porównuje metodę uczenia TD(λ) z dotychczasowymi znanymi metodami i udowadnia jednocześnie jej wyższość pod względem obszaru stosowalności. Odpowiednie ustalenie wartości parametru l pozwala sprowadzać zachowanie tej metody do innych już znanych.

Metoda TD(λ) została bardzo pomyślnie wykorzystana w 1992 roku w programie TD-GAMMON grającym na bardzo wysokim poziomie w grę backgammon. Sukces tego programu doprowadził do zwiększenia zainteresowania tą metodą w zastosowaniu w różnych grach strategicznych.

Zastosowanie metody TD(λ) w szachach sprowadzać się będzie do porównywania predykcji wyniku gry w kolejnych posunięciach aż do zakończenia partii, a więc uzyskania ostatecznego rezultatu. Oczywiste jest, że po większości ruchów, a więc przy wielu różnych rozstawieniach figur, niemożliwe jest jednoznaczne odpowiedzenie na pytanie jak dana pozycja ma się do końcowego wyniku.

Każda procedura ucząca wyrażana jest na podstawie zasad aktualizacji wag połączeń pomiędzy neuronami. Zakładając zatem, że wagi w metodzie TD(λ) aktualizowane są po pełnej sekwencji obserwacji zakończonej zaobserwowanym wyjściem, można zapisać, że:

$$w \leftarrow w + \sum_{t=1}^{m} \Delta w_t \,, \tag{1}$$

gdzie zarejestrowano m obserwacji, a Δw_t jest wektorem zmian wektora wag w wynikającym z obserwacji w chwili t. Zakładając teraz, że jeśli pełna sekwencja obserwacji jest postaci $x_1, x_2, \ldots, x_m, z$, gdzie x_t jest wektorem wejść podanym na sieć w chwili t i z jest właściwą odpowiedzią systemu, to sztuczna sieć neuronowa generuje odpowiednią sekwencję predykcji $P_1, P_2, \ldots, P_m$ będących szacowaniem ostatecznej wartości z. Każda predykcja P_t jest funkcją zależną od wektora wejściowego x_t oraz wektora wag w. Wektor zmian wag wyliczany jest jako:

$$\Delta w_t = a\left(P_{t+1} - P_t\right)\sum_{k=1}^{t} l^{t-k} \nabla_w P_k \,, \tag{2}$$

gdzie a jest ustalonym współczynnikiem uczenia, l jest współczynnikiem metody TD(λ), $\nabla_w P_k$ jest gradientem funkcji k-tej predykcji względem wektora wag w oraz dla $t = m$ zachodzi $P_{m+1} = z$. Ekspotencjalne ważenie gradientów przeszłych predykcji ma swoją szczególną zaletę w procesie ich kumulacji (szereg z równania (2)). Zapisując równanie (2) w postaci:

$$\Delta w_t = a\left(P_{t+1} - P_t\right)e_t \,, \tag{3}$$

gdzie

$$e_t = \sum_{k=1}^{t} l^{t-k} \nabla_w P_k \,, \tag{4}$$

można w prosty przyrostowy sposób wyliczać kolejne wartości e_{t+1} jedynie na podstawie bieżącej informacji o wartości e_t:

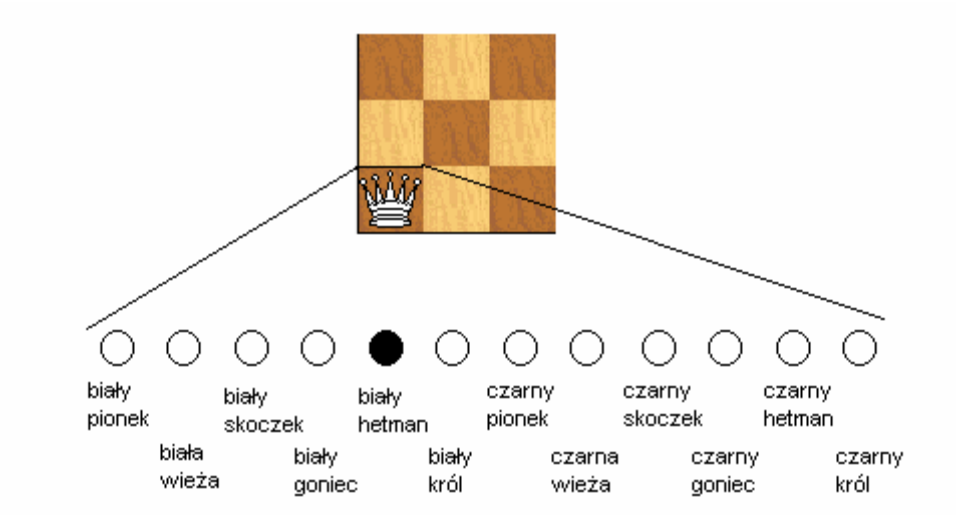
$$e_{t+1} = \sum_{k=1}^{t+1} l^{t+1-k} \nabla_w P_k = l\left(\sum_{k=1}^{t} l^{t-k} \nabla_w P_k\right) + \nabla_w P_{t+1} = l\, e_t + \nabla_w P_{t+1} \,. \tag{5}$$

Powyższy zabieg znacznie upraszcza implementację omawianej metody uczenia, ponieważ z jednej strony skraca niewygodny proces obliczeń, a z drugiej – unika ewentualnego „zaśmiecania” pamięci wyznaczanymi przyrostami.

Realizacja problemu

Jak zwykle największym problemem podczas projektowania sztucznej sieci neuronowej jest trafne zdefiniowanie struktury wejść oraz wyjść jak również samej topologii sieci. Od tego wyboru bardzo często zależy przyszły sukces wyuczonej sieci. W ramach realizacji projektu założono jednak stworzenie szkieletu takiej sieci, która będzie uczona metodą TD(λ).

Zaprojektowana sztuczna sieć neuronowa zbudowana jest z trzech warstw: wejściowej, ukrytej i wyjściowej. W warstwie wyjściowej znajduje się tylko jeden neuron, którego wyjście wyznaczać będzie wartość funkcji oceniającej dla zadanej na wejściu pozycji szachowej. W celu uproszczenia zadania przyjęto proste kodowanie pozycji jedynie na podstawie rozmieszczenia figur na szachownicy. Dla każdego pola szachownicy zarezerwowano 12 neuronów przyjmujących na wejściu dyskretne impulsy 0 lub 1 (Rysunek 2.). Każdy neuron pobudzany jest tylko i wyłącznie przez zdefiniowaną dla niego figurę. Przy założeniu, że dla każdego pola zestaw 12 neuronów jest identycznie uporządkowany, kodowanie rozmieszczenia figur na szachownicy jest całkowicie jednoznaczne (Kod 1.).



Rysunek 2. Przykład zakodowania białego hetmana przez 12 neuronów związanych z jednym polem szachownicy.

W warstwie wejściowej znajduje się ostatecznie $64x12 = 768$ neuronów, ponieważ szachownica składa się z 64 pól. Dla warstwy ukrytej założono, że w jej skład wchodzi $\sqrt{n_{in} n_{out}} = \sqrt{768} \approx 27$ neuronów, gdzie n_{in} wyznacza liczbę neuronów w warstwie wejściowej i n_{out} - liczbę neuronów w warstwie wyjściowej.

W topologii sieci wykorzystano połączenia pełne pomiędzy kolejnymi warstwami:

- $768x27 = 20736$ ważonych połączeń pomiędzy warstwą wejściową i ukrytą,
- $27x1 = 27$ ważonych połączeń pomiędzy warstwą ukrytą i wyjściową.

```
private int[] getInputs ( ChessBoard cb )
    {
      int[] inputs = new int[nOfInputs];
      for ( int j=0; j<8; j++ )
            for ( int i=0; i<8; i++ )
            {
                  Piece piece = cb.getPiece(i,j);
                  if ( piece!=null )
                  {
                        int offset = (piece.isWhite()?0:6);
                        if ( piece.isPawn() )
                              inputs[j*8+i+offset] = 1;
                        else if ( piece.isRook() )
                              inputs[j*8+i+offset+1] = 1;
                        else if ( piece.isKnight() )
                              inputs[j*8+i+offset+2] = 1;
                        else if ( piece.isBishop() )
                              inputs[j*8+i+offset+3] = 1;
                        else if ( piece.isQueen() )
                              inputs[j*8+i+offset+4] = 1;
                        else if ( piece.isKing() )
                              inputs[j*8+i+offset+5] = 1;
                  }
            }
      return inputs;
    }
```

Kod 1. Definicja metody getInputs klasy TDLambdaPlayer kodująca przekazaną przez obiekt typu ChessBoard pozycję szachową.

Jako funkcję aktywacji każdego neuronu przyjęto standardową funkcję sigmoidalną:

$$f(x)=\frac{1}{1+e^{-bx}}, \tag{6}$$

gdzie b jest współczynnikiem określającym stromość sigmoidy.

Dla zdefiniowanej równaniem (6) funkcji aktywacji można określić ostateczną postać wyjściowych predykcji sieci jako:

$$P_t(x_t,w)=f\left(\sum_{i=1}^{k} w_i^{ho} f\left(\sum_{j=1}^{l} w_{ji}^{ih} x_{tj}\right)\right), \tag{7}$$

gdzie

k - liczba neuronów w warstwie ukrytej,

l - liczba neuronów w warstwie wejściowej,

w_i^{ho} - waga połączenia i-tego neuronu warstwy ukrytej z jedynym neuronem warstwy wyjściowej,

w_{ji}^{ih} - waga połączenia j-tego neuronu warstwy wejściowej z i-tym neuronem warstwy ukrytej,

x_{tj} - j-te wejście w chwili t.

W celu wykorzystania równań (3) i (5) na korekcję wag po zakończonej partii konieczne będzie jeszcze wyznaczenie gradientu funkcji predykcji:

$$\nabla_w P_k = \left[\frac{\partial P_k}{\partial w_1^{ho}}, \frac{\partial P_k}{\partial w_2^{ho}}, \mathbf{K}, \frac{\partial P_k}{\partial w_k^{ho}}, \frac{\partial P_k}{\partial w_{11}^{ih}}, \frac{\partial P_k}{\partial w_{21}^{ih}}, \mathbf{K}, \frac{\partial P_k}{\partial w_{l1}^{ih}}, \frac{\partial P_k}{\partial w_{12}^{ih}}, \mathbf{K}, \frac{\partial P_k}{\partial w_{lk}^{ih}} \right], \tag{8}$$

gdzie

$$\frac{\partial P_k}{\partial w_i^{ho}} = f'\left(net^o\right) f\left(net_i^h\right), \tag{9}$$

oraz

$$\frac{\partial P_k}{\partial w_{ji}^{ih}} = f'\left(net^o\right) w_i^{ho} f'\left(net_i^h\right) x_{tj}, \tag{10}$$

przy czym

$net^o = \sum_{i=1}^{k} w_i^{ho} f\left(net_i^h\right)$ - pobudzenie w neuronie wyjściowym,

$net_i^h = \sum_{j=1}^{l} w_{ji}^{ih} x_{tj}$ - pobudzenie w i-tym neuronie warstwy ukrytej.

Wykorzystując ostatecznie równania (1), (3), (5), (9) oraz (10) można przejść do zaimplementowania funkcji akumulującej wyliczane po każdym ruchu przyrosty (Kod 2.).

```
public void accumulate ( double estimate )
{
      if ( lastOutput >= 0 )
      {
            for (int i = 0; i < nOfInputs; i++)
              for (int j = 0; j < nOfHiddens; j++)
                   dwIH[i][j] += alpha * (estimate - lastOutput) * ewIH[i][j];

                   for (int i = 0; i < nOfHiddens; i++)
                         dwHO[i] += alpha * (estimate - lastOutput) * ewHO[i];
      }

      for (int i = 0; i < nOfInputs; i++)
            for (int j = 0; j < nOfHiddens; j++)
                   ewIH[i][j]  =  fPrime(netO)  *  wHO[j]  *  fPrime(netH[j])  *
inputs[i] + lambda * ewIH[i][j];

      for (int i = 0; i < nOfHiddens; i++)
            ewHO[i] = fPrime(netO) * f(netH[i]) + lambda * ewHO[i];

      lastOutput = estimate;
}
```

Kod 2. Definicja metody accumulate klasy TDLambdaPlayer akumulująca przyrosty w tablicach ewIH, ewHO, dwIH, dwHO przy wykorzystaniu przekazanej przez parametr bieżącej predykcji oraz zapamiętanej ostatnio predykcji.

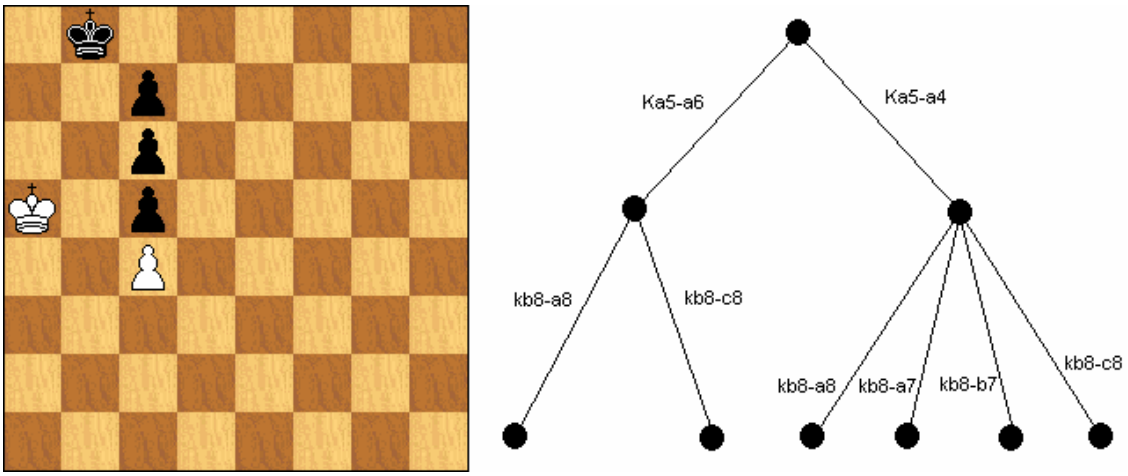
Wyjście sieci wyliczane jest oczywiście na podstawie równania (7) (Kod 3.).

```
public double getOutput ( ChessBoard cb )
    {
        inputs = getInputs(cb);
        for (int i = 0; i < nOfHiddens; i++)
        {
            netH[i] = 0;
            for (int j = 0; j < nOfInputs; j++)
                netH[i] += inputs[j] * wIH[j][i];
        }
        netO = 0;
        for (int i = 0; i < nOfHiddens; i++)
            netO += f(netH[i]) * wHO[i];

        return f(netO);
    }
```

Kod 3. Definicja metody getOutput klasy TDLambdaPlayer wyliczająca wartość wyjścia sieci na podstawie wprowadzonej na wejście pozycji szachowej.

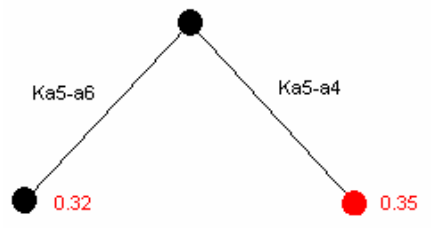
Po zaprojektowaniu i zaimplementowaniu sieci przystąpiono do przeprowadzania eksperymentów. Pamiętając, że zgodnie z założeniami wyjście sieci ma być wartością funkcji oceniającej konkretną pozycję szachową, ustalono, że na wejście sieci będzie podawane rozmieszczenie figur po kilku pół-ruchach naprzód. Pod pojęciem pół-ruch należy rozumieć jedno posunięcie białej bądź czarnej figury. Podejście to sprowadza się do generowania drzewa gry, w którym węzłami są kolejne pozycje szachowe, gałęziami są możliwe (prawidłowe!) do wykonania pół-ruchy, natomiast korzeniem jest pozycja startowa, dla której tworzone jest drzewo. Liście tego drzewa będą podlegać ocenie przez sztuczną sieć neuronową. Przykładowe drzewo gry zostało przedstawione na Rysunku 3.



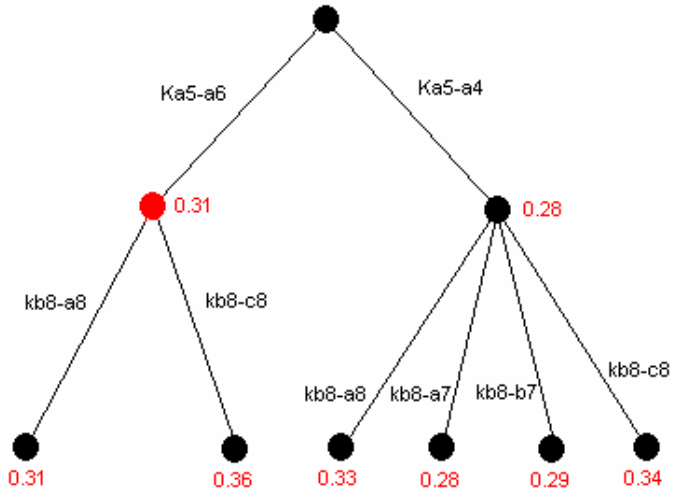
Rysunek 3. Drzewo gry wyprowadzane do drugiego pół-ruchu dla przykładowej pozycji szachowej, w której pierwszy ruch mają wykonać białe.

Z powodu braku czasu w implementacji ograniczono się do eksperymentów dla drzewa gry wyprowadzanego jedynie do 1. poziomu (tj. jeden pół-ruch do przodu) i do 2.poziomu (tj. jeden pełny ruch do przodu). We wszystkich partiach uczona sieć neuronowa będzie grała zawsze białymi figurami, a więc w każdej sytuacji będzie starała się wykonać ruch, który maksymalizuje wartość funkcji oceniającej (wyjścia).

Przeszukując drzewo gry w głąb wyjście sieci będzie wykorzystywane do wyszukiwania najlepszych pozycji dla koloru białego (maksymalizacja wartości) oraz wyszukiwania najlepszych pozycji dla koloru czarnego (minimalizacja wartości). Na Rysunku 4. oraz Rysunku 5. przedstawiono przykładowe wartościowanie liści dla drzewa gry (z Rysunku 3.) złożonego z jednego poziomu oraz z dwóch poziomów, jak również sposób wyboru najlepszego posunięcia. W obu przypadkach należy wybrać jeden z dwóch możliwych do wykonania ruchów. Na Rysunku 4. widać, że ocenie podlegały dwie pozycje, z których nieco lepiej została oceniona pozycja po wybranym posunięciu Ka5-a4. Na Rysunku 5. widać z kolei, że ocenie podlegało sześć pozycji, z których każda dotyczyła sytuacji po wykonaniu posunięcia przez czarne. Spośród tych pozycji wybrane zostały te, których wartość funkcji oceniającej była najmniejsza, a więc te, które uważane są za najlepsze dla czarnych z punktu widzenia tej funkcji. Wybrana wartość propagowana jest w górę do węzła rodzica, który reprezentuje w tym momencie rozmieszczenie figur zaistniałe po wykonaniu posunięcia przez białe. Spośród tych węzłów wybierane są analogicznie te, których wartościowanie jest najlepsze dla białych, a więc te, których wartość jest największa. W ten sposób okazuje się, że w drugim przypadku lepszym posunięciem jest Ka5-a6. Postępowanie z bardziej rozbudowanymi drzewami byłoby analogiczne, tzn. na każdym poziomie tego drzewa propagowano by w górę odpowiednie wartości minimalne bądź maksymalne.



Rysunek 4. Drzewo gry wyprowadzone do pierwszego pół-ruchu wraz z przykładowym wartościowaniem liści i wyborem najlepszego posunięcia białych.



Rysunek 5. Drzewo gry wyprowadzone do drugiego pół-ruchu wraz z przykładowym wartościowaniem liści i wyborem najlepszego posunięcia białych.

Wyniki badań

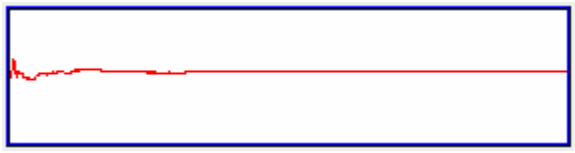
Korzystając z powyższych założeń wykonano eksperymenty dla dwóch przypadków:

- sieć neuronowa będąca na ruchu generuje jednopoziomowe drzewo,
- sieć neuronowa będąca na ruchu generuje dwupoziomowe drzewo.

W obu przypadkach sieć uczyła się grać z algorytmem wykonującym losowe ruchy, reprezentowanym przez klasę **ComputerPlayer**.

Eksperyment 1

W ramach pierwszego eksperymentu arbitralnie ustalono wszystkie stałe wartości współczynników: $a = 0.3$, $b = 5.0$ oraz $l = 0.8$. Sieć neuronowa miała do rozegrania 1000 partii, z czego 114 udało jej się wygrać, 842 zremisować i 44 przegrać, uzyskując tym samym 535 punktów na 1000 możliwych. Na Rysunku 6. przedstawiono progresję wyników zaobserwowanych podczas eksperymentu.

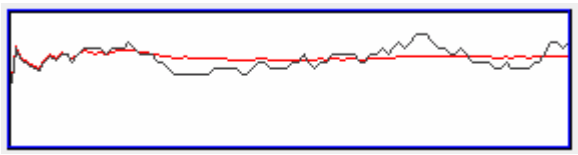


Rysunek 6. Wykres przedstawiający procentowy udział zdobytych punktów w trakcie rozgrywania kolejnych partii treningowych w ramach pierwszego eksperymentu.

Pomimo losowego charakteru gry sieci neuronowej zauważyć można, że ma ona tendencję do częstszego wygrywania aniżeli przegrywania z całkowicie losowym algorytmem. Wyniki na pewno nie są zadowalające, ale mogą przynajmniej w części utwierdzić w przekonaniu, że jest w tym wszystkim pozytywny wpływ zastosowanej metody uczenia. Nie można jednak mieć pewności czy aby 2.5 razy większa liczba zwycięstw nie jest zasługą jakichś innych przypadkowych czynników, np. rozpoczynanie gry przez białe, chociaż posługując się zdrowym rozsądkiem przy całkowicie losowej grze obu algorytmów nie powinno mieć to znaczenia. Nie zmienia to jednak faktu, iż większość partii kończyła się remisem, co w jednoznaczny sposób wskazuje na efekt losowości w grze obu zastosowanych strategii – większość tych partii doprowadzana była do „gołych króli”. Mimo wszystko bardzo interesujące wydaje się być spostrzeżenie, iż sieć neuronowa dość często potrafiła doprowadzać w grze do przewagi materialnej, której niestety nie potrafiła później wykorzystać.

Eksperyment 2

W ramach drugiego eksperymentu również arbitralnie dobrano stałe wartości współczynników: $a = 0.5$, $b = 1.0$ oraz $l = 0.8$. Sieć neuronowa miała tym razem do rozegrania jedynie 100 partii, z czego 37 udało jej się wygrać, 62 zremisować i tylko 1 przegrać, uzyskując tym samym 68 punktów na 100 możliwych. Na Rysunku 7. przedstawiono progresję wyników zaobserwowanych podczas eksperymentu.



Rysunek 7. Wykres przedstawiający procentowy udział zdobytych punktów w trakcie rozgrywania kolejnych partii treningowych w ramach drugiego eksperymentu.

Jak widać wyniki drugiego eksperymentu już znacznie się poprawiły. Niestety jest to znaczna zasługa rozszerzenia drzewa gry o jeden poziom wraz z umożliwieniem postawienia mata królowi przeciwnika w jednym posunięciu jeśli tylko jest taka możliwość. Ciężko w tej sytuacji w sposób pewny się

wypowiedzieć na temat wyników zastosowanej metody uczenia. Można by zaryzykować stwierdzenie, że efektem uczenia jest doprowadzanie do sytuacji, w której sieć może w jednym ruchu dać mata. Niestety z wykresów progresji uzyskanych przez sieć wyników nie można tego jednoznacznie osądzić, gdyż w końcowej fazie treningu procentowy udział zdobytych punktów w ostatnich dziesięciu partiach obniżył się praktycznie do 50%. Niemniej jednak udział ten potrafił utrzymywać się w górze dłużej niż na początku treningu. Ciekawym spostrzeżeniem wydaje się być tendencja sieci do szachowania króla przeciwnika, ale niestety dość często do szachowania „bezmyślnego", tzn. poprzez ustawienie takiego szacha, w którym jedyną obroną przeciwnika jest zbicie figury szachującej przez szachowanego króla.

Wnioski

Niestety w obu przeprowadzonych eksperymentach nie można mówić o jakimkolwiek sukcesie. Można mieć jedynie nadzieję, że nieco lepsze wyniki są faktycznie zasługą zastosowanej metody uczenia. Wyuczone strategie gry są mimo wszystko dalekie od doskonałości, a właściwie nadal pozostają one bardzo mocno losowe. W grze takiej sieci pojawiają się praktycznie same bezsensowne ruchy. Pewnym plusem wyuczonych sieci było wyrabianie pewnych stałych zachowań, np. szachowanie króla przeciwnika czy też doprowadzanie do przewagi materialnej. Na razie można jednak bardziej mówić o niepowodzeniu, chociaż główny cel projektu został zrealizowany, tzn. skonstruowano szkielet sieci neuronowej uczonej metodą TD(λ).

Wyniki tej części projektu utwierdzają jeszcze mocniej w przekonaniu, iż odnalezienie optymalnej strategii gry w szachy jest zadaniem bardzo trudnym. Na pewno jeszcze wiele rzeczy wymaga dopracowania. Na przykład największym wyzwaniem jest kwestia wejścia sieci, która powinna być lepiej przemyślana, tzn. oprócz rozstawienia figur należałoby uwzględnić również pewne własności strategiczne, tj. wartość materialna figur i idąca z nimi ewentualna przewaga materialna jednej ze stron, położenie króla w zależności od fazy gry (zabezpieczanie go w rogu szachownicy w trakcie początkowej i środkowej fazy gry i jego uaktywnianie w końcowej fazie), struktury pionowe (pionki izolowane i zdublowane), zdobywanie centrum, rozwijanie figur w początkowej fazie gry, uaktywnianie wież w środkowej fazie gry, itp. Należy zatem dobrze przemyśleć, które własności gry z punktu widzenia oceny zadanej pozycji powinno się zawrzeć w optymalizowanej funkcji oceniającej. Odrębną kwestią jest zakodowanie tych własności w neuronach warstwy wejściowej. W projekcie przyjęto prostą zasadę „jeden z wielu" dla każdego pola szachownicy, tj. spośród dwunastu neuronów związanych z określonym polem co najwyżej jeden neuron mógł być pobudzony. Warto również wypróbować impulsy bipolarne, aby wzmocnić różnicę pomiędzy szacowanym wynikiem gry, a jej ostatecznym rezultatem. Partia przegrana byłaby wówczas reprezentowana przez wartość -1, remis – przez wartość 0 i wygrana – przez wartość 1. Podejście to mogłoby nieco wzmocnić efekt uczenia.

Bardzo ważną kwestią jest również dobór odpowiedniego przeciwnika i dużej liczby gier podczas treningu. Raczej nie wydaje się możliwe, aby sieć neuronowa nauczyła się dobrze grać w szachy grając tylko i wyłącznie z zawodnikiem losowo wykonującym ruchy. Ciężko też wyobrazić sobie jak sieć neuronowa miałaby kształtować swoją grę przeciwko takiemu przeciwnikowi. W ramach realizacji projektu założono jednak, że powinna istnieć chociażby możliwość ukształtowania takiej strategii, która by częściej wygrywała.

W ramach dalszych prac dobrze byłoby zaopatrzyć się w pewien program szachowy, który umożliwiałby grę na wielu poziomach zaawansowania. Jeszcze lepiej byłoby gdyby istniał prosty interfejs komunikacji z takim programem, aby proces uczenia sieci przebiegał automatycznie. Mając już do wyboru strategie, które grają w pewien logiczny sposób (nie przypadkowy), można by pewniej oczekiwać widocznych efektów nauki.

Kolejnymi ulepszeniami, nie związanymi co prawda stricte ze sztuczną inteligencją, są rozwiązania algorytmiczne do generowania kolejnych ruchów i przeszukiwania drzewa gry. Zależałoby na tym, aby przeszukiwanie tego drzewa było zaimplementowane dosyć efektywnie, aby później w lepszym stopniu wykorzystać techniki sztucznej inteligencji do optymalizowania strategii gry. Jak wiadomo proces uczenia jest często procesem bardzo czasochłonnym i wypadałoby przynajmniej w minimalnym stopniu zwiększyć wydajność poszukiwania najlepszego w danej pozycji ruchu. Optymalną głębokością drzewa, przy której proces uczenia miałby większy sens, wydaje się być około 6 poziomów (6 pół-ruchów). W końcu każdy dobry szachista jest w stanie przeanalizować wiele wariantów gry na przestrzeni około trzech pełnych ruchów, a nawet dalej w pewnych intuicyjnie wybranych rozgałęzieniach. Podczas realizacji tego projektu skupiono się na dość skąpych pod względem wielkości drzewach. Analiza jednego pół-ruchu naprzód nawet przy świetnie wyuczonej strategii jest z pewnością niewystarczająca do dobrej gry. W jednym pół-ruchu prawie niemożliwe staję się przestrzeganie przed różnego rodzaju nawet najprostszymi groźbami. Analiza dwóch pół-ruchów naprzód też nie jest zachwycającym rozwiązaniem, chociaż może w części pozwala ustrzec się przed prostymi groźbami (np. podstawienie figury).

Pozyskiwanie wiedzy z szachowych baz danych

Niniejsza część projektu zostanie poświęcona kwestii wydobywania wiedzy z szachowych baz danych na przykładzie zestawu końcówek typu KRK (*King-Rook-King*), tzn. końcówek złożonych z trzech figur: białego króla, białej wieży oraz czarnego króla. Wydobyta wiedza zostanie wykorzystana w celu stworzenia strategii wygrywającej dla białych figur.

Wstęp

W pracy Johannes Fürnkranza [2] znajduje się bardzo ciekawy przegląd zastosowań maszynowego uczenia w szachach. W jednym z rozdziałów opisuje on między innymi ogólne metody indukcji klasyfikatorów końcówek szachowych. Okazuje się, że w ostatnich latach znacząco wzrosło zainteresowanie problemem wydobywania wiedzy szachowej z tego typu baz. Dobrze pozyskana wiedza z pewnej klasy końcówek przysługuje się nie tylko programom szachowym, ale również ludziom chcącym lepiej zrozumieć, a nawet lepiej poznać pewne własności dotąd nieznane i niedostrzegane. Najbardziej na tym polu wsławił się Ken Thompson, który w swoim laboratorium, korzystając m.in. z obliczeń rozproszonych, zdążył już wygenerować najlepsze warianty gry we wszystkich możliwych pozycjach końcowych złożonych z trzech, czterech oraz pięciu figur. Na tym jednak nie kończy się zabawa z „końcówkami”, ponieważ głównym dążeniem, wielu zafascynowanych szachami osób, jest uzyskanie jak najmniejszego zbioru reguł/zasad kompresującego niejako potężne bazy danych tych końcówek.

Typowym podejściem w maszynowym uczeniu końcówek szachowych jest ich klasyfikacja ze względu na wygraną bądź nie-wygraną, a więc na kształt klasy logicznej (tak/nie). Pewnego rodzaju końcówki zaczęto również klasyfikować ze względu na liczbę pół-ruchów bądź pełnych ruchów potrzebnych do wygrania jednej ze stron. Przykładem tego typu końcówek jest analizowana w ramach tego projektu końcówka król i wieża przeciwko królowi, która w przeważającej większości pozycji jest zawsze do wygrania przez stronę posiadającą wieżę. Trzy typy końcówek stały się nawet polem doświadczalnym do testowania algorytmów maszynowego uczenia. W repozytorium znajdującym się pod adresem http://www.ics.edu/~mlearn/databases w katalogu *chess* można znaleźć zbiory danych dotyczące końcówek KRK (*King-Rook-King* – król i wieża przeciwko królowi), KRKP (*King-Rook-King-Pawn* – król i wieża przeciwko królowi i pionowi) oraz KRKN (*King-Rook-King-Knight* – król i wieża przeciwko królowi i skoczkowi). Stąd też pobrane zostały dane końcówek KRK do dalszej realizacji projektu.

Podczas pozyskiwania wiedzy z końcówek szachowych nieodłącznym problemem staje się odpowiednia ich reprezentacja. Typowe metody indukcyjnego uczenia wymagają reprezentacji przykładów w postaci sekwencji par atrybut-wartość, przy czym każdy atrybut przykładu specyfikowany jest przez dokładnie jedną wartość ze skończonego zbioru możliwych wartości. Łatwo sobie wyobrazić, że informacje dotyczące rozstawienia figur na szachownicy na pewno nie będą wystarczające do stworzenia dobrze uogólnionych reguł. Dlatego też często proponuje się dołączanie do zbiorów uczących dodatkowych atrybutów, które dla każdej pozycji niosą ze sobą jednoznaczne informacje, które mogą z kolei przydać się w procesie wydobywania reguł.

Realizacja problemu

Ze wspomnianego wcześniej repozytorium ściągnięto zbiór końcówek szachowych typu KRK (Rysunek 8.).

```
d,4,h,8,f,1,nine
d,4,h,8,g,1,nine
d,4,h,8,h,2,nine
a,1,a,3,e,1,ten
a,1,a,3,f,1,ten
a,1,a,3,f,2,ten
```

Rysunek 8. Fragment zbioru przykładów uczących znajdujący się w pliku *krkopt.data*
w katalogu *Dane/king-rook-vs-king*

Zbiór ten składa się z 28056 przykładów, z których każdy zdefiniowany jest przez siedem atrybutów, przy czym ostatni z nich jest klasą, do której należy przykład. Każdy przykład dotyczy pozycji szachowej, w której czarne są na ruchu. Znaczenie oraz dziedziny tych atrybutów są następujące:

- atrybut 1. *KFile* – kolumna, w której znajduje się biały król: $\{a,b,c,d,e,f,g,h\}$,
- atrybut 2. *KRank* – wiersz, w którym znajduje się biały król: $\{1,2,3,4,5,6,7,8\}$,
- atrybut 3. *RFile* – kolumna, w której znajduje się biała wieża: $\{a,b,c,d,e,f,g,h\}$,
- atrybut 4. *RRank* – wiersz, w którym znajduje się biała wieża: $\{1,2,3,4,5,6,7,8\}$,
- atrybut 5. *kFile* – kolumna, w której znajduje się czarny król: $\{a,b,c,d,e,f,g,h\}$,

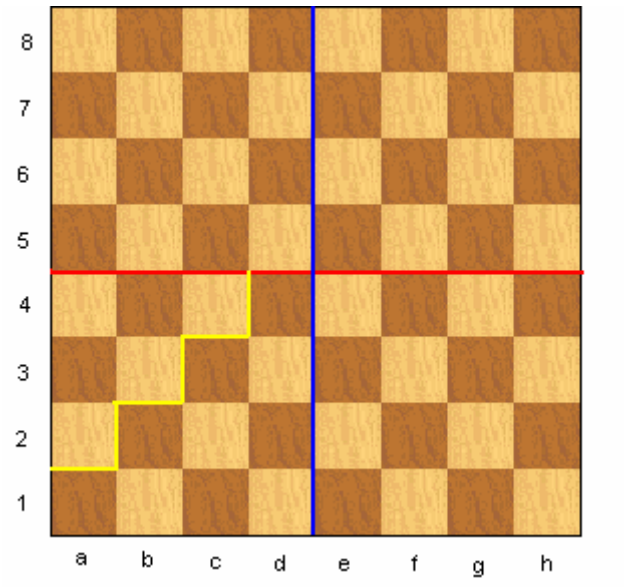
- atrybut 6. *kRank* – wiersz, w którym znajduje się czarny król: $\{1,2,3,4,5,6,7,8\}$,
- atrybut 7. *optimalDepth* – liczba ruchów potrzebna do wygrania końcówki przez białe przy optymalnej grze obu stron: *{draw, zero, one, two, three, four, five, six, seven, eight, nine, ten, eleven, twelve, thirteen, fourteen, fifteen, sixteen}.*

Dodatkowego wyjaśnienia wymaga atrybut 7., którego wartości *draw* oraz *zero* mogą nie być do końca zrozumiałe. Wartość *draw* oznacza, że dany przykład reprezentuje pozycję w której czarny król może w jednym ruchu zbić białą wieżę bądź pozycję patową czarnego króla, tzn. pozycję w której czarny król nie jest szachowany i jednocześnie nie może wykonać żadnego ruchu, a więc sytuację, która zakończyła się remisem. Z kolei wartość *zero* oznacza, że dany przykład reprezentuje pozycję matową, tzn. czarny król jest szachowany i jednocześnie nie może wykonać żadnego ruchu obronnego, a więc jest to sytuacja zakończona zwycięstwem białych figur.

Powyższy zbiór danych uczących został opisany i wykorzystany w eksperymentach maszynowego uczenia przeprowadzanych przez Baina w pracy [3]. Okazuje się bowiem, że powyższy zbiór nie zawiera wszystkich możliwych pozycji szachowych zawierających odpowiednio białego króla, białą wieżę oraz czarnego króla. Na zbiór ten składa się tzw. kanoniczna przestrzeń pozycji. Można łatwo obliczyć, iż zbiór wszystkich możliwych pozycji KRK składałby się z

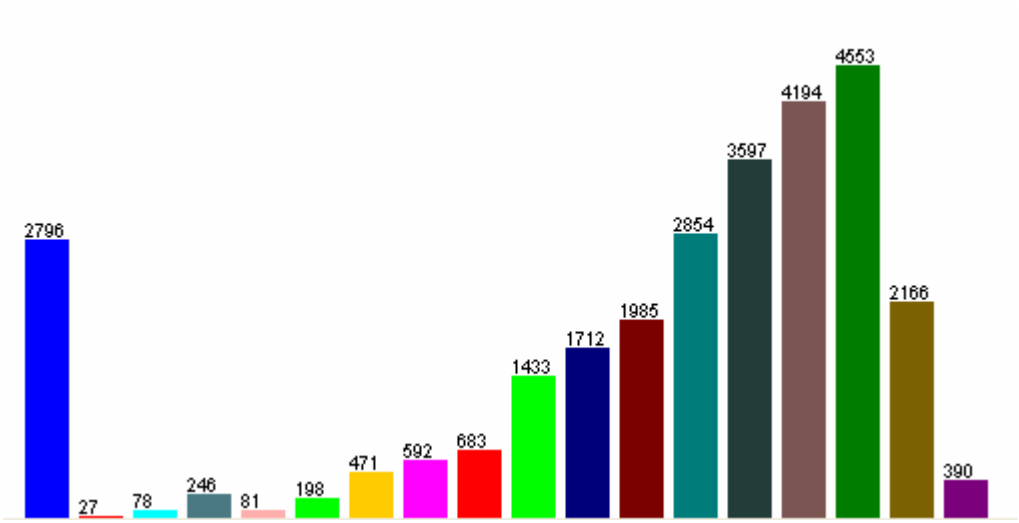
$$\binom{64}{3} \cdot 3! = 249984$$

pozycji (liczba 3-elementowych kombinacji pól ze zbioru 64-elementowego wymnożony przez liczbę permutacji trzech figur). Jednakże wykorzystując symetrie szachownicy okazuje się, iż kilka różnych pozycji można tak naprawdę sprowadzić do jednej z nich (poprzez obroty szachownicy bądź odbicia względem wyznaczonych osi symetrii). Sprawa byłaby nieco bardziej skomplikowana w sytuacji gdy na szachownicy znajdowałby się co najmniej jeden pionek. Natomiast w końcówkach typu KRK wykorzystanie tych symetrii pozwala zredukować liczbę przechowywanych pozycji do 28056. Na Rysunku 9. przedstawiono kluczowe elementy szachownicy, które pozwalają sprowadzić każdą pozycję typu KRK do jednej z pozycji kanonicznej przestrzeni. Każdą pozycję typu KRK można tak odbić i obrócić, aby biały król znalazł się w jednym z dziesięciu kluczowych dla niego pól *{a1, b1, b2, c1, c2, c3, d1, d2, d3, d4}* (poniżej żółtej krzywej). Przykładowo jeżeli biały król znajduje się powyżej czerwonej osi, to pozycję można odbić symetrycznie względem tej osi, tak aby biały król znalazł się poniżej niej. Jeżeli biały król znajduje się z prawej strony niebieskiej osi, to pozycję można odbić symetrycznie względem tej osi, tak aby biały król znalazł się po jej lewej stronie. Jeżeli biały król znajduje się powyżej diagonali *a1-h8*, to pozycję można odbić symetrycznie względem niej, tak aby biały król znalazł się ostatecznie w jednym z dziesięciu kluczowych dla niego pól. Jeżeli biały król wylądował na jednym z pól *a1, b2, c3, d4* i czarny król znajduje się powyżej diagonali *a1-h8*, to pozycję można odbić symetrycznie względem niej, tak aby czarny król znalazł się poniżej tej diagonali.



Rysunek 9. Symetrie szachownicy pozwalające doprowadzić każdą pozycję do jej kanonicznego odpowiednika.

Zakładając zatem, że wygenerowany zbiór uczący zawiera praktycznie wszystkie możliwe pozycje typu KRK, chciałoby się wydobyć z niego dość ogólną wiedzę klasyfikującą te pozycje ze względu na atrybut 7. Pod pojęciem dość ogólnej wiedzy należy tutaj rozumieć wykorzystanie własności innych niż bezpośrednia lokalizacja figur. Wygenerowany zbiór reguł klasyfikujących mógłby posłużyć do zdefiniowania wygrywającej strategii gry dla białych. Algorytm mający do dyspozycji zestaw dokładnie klasyfikujących reguł wykorzystywałby je do klasyfikacji pozycji powstających po każdym z możliwych do wykonania przez białe ruchu. Jako najlepsze byłoby wybierane to posunięcie, które zmniejsza liczbę ruchów potrzebnych do wygrania końcówki (atrybut 7. – klasa). Na Rysunku 10. przedstawiono rozkład przykładów w każdej z osiemnastu wartości klasy.

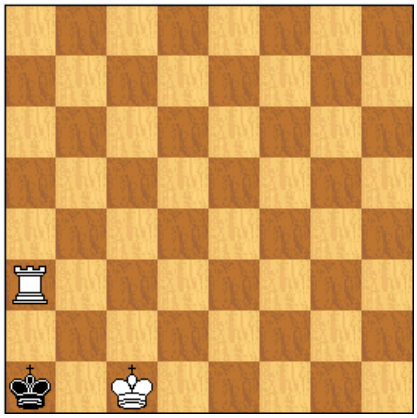


Rysunek 10. Liczba przykładów zbioru uczącego przypadająca na każdą wartość klasy w porządku *draw, zero, one, two, three, four, five, six, seven, eight, nine, ten, eleven, twelve, thirteen, fourteen, fifteen, sixteen*.

Mając zdefiniowane podłoże pracy i wytyczony cel, jakim jest wydobycie jak najbardziej ogólnej wiedzy na temat klasyfikacji końcówek typu KRK, przystąpiono do definicji dodatkowych atrybutów niosących ze sobą pewne dodatkowe informacje na temat każdego przykładu:

- *KMinDistanceFile* – minimalna odległość białego króla od dolnej bądź górnej krawędzi szachownicy $\{0,1,2,3\}$,
- *KMinDistanceRank* – minimalna odległość białego króla od lewej bądź prawej krawędzi szachownicy $\{0,1,2,3\}$,
- *RMinDistanceFile* – minimalna odległość białej wieży od dolnej bądź górnej krawędzi szachownicy $\{0,1,2,3\}$,
- *RMinDistanceRank* – minimalna odległość białej wieży od lewej bądź prawej krawędzi szachownicy $\{0,1,2,3\}$,
- *kMinDistanceFile* – minimalna odległość czarnego króla od dolnej bądź górnej krawędzi szachownicy $\{0,1,2,3\}$,
- *kMinDistanceRank* - minimalna odległość czarnego króla od lewej bądź prawej krawędzi szachownicy $\{0,1,2,3\}$,
- *KkFile* – określenie czy oba króle stoją w tej samej kolumnie $\{0,1\}$,
- *KkRank* – określenie czy oba króle stoją w tym samym wierszu $\{0,1\}$,
- *KRFile* – określenie czy biały król z białą wieżą stoją w tej samej kolumnie $\{0,1\}$,
- *KRRank* – określenie czy biały król z białą wieżą stoją w tym samym wierszu $\{0,1\}$,
- *kRFile* – określenie czy czarny król z białą wieżą stoją w tej samej kolumnie $\{0,1\}$,
- *kRRank* – określenie czy czarny król z białą wieżą stoją w tym samym wierszu $\{0,1\}$,
- *KkDistanceFile* – odległość białego króla od czarnego króla względem kolumny $\{0,1,2,3,4,5,6,7\}$,
- *KkDistanceRank* – odległość białego króla od czarnego króla względem wiersza $\{0,1,2,3,4,5,6,7\}$,
- *KRDistanceFile* – odległość białego króla od białej wieży względem kolumny $\{0,1,2,3,4,5,6,7\}$,
- *KRDistanceRank* – odległość białego króla od białej wieży względem wiersza $\{0,1,2,3,4,5,6,7\}$,
- *kRDistanceFile* – odległość czarnego króla od białej wieży względem kolumny $\{0,1,2,3,4,5,6,7\}$,
- *kRDistanceRank* – odległość czarnego króla od białej wieży względem wiersza $\{0,1,2,3,4,5,6,7\}$,
- *KkOppositionDistance* – odległość białego króla od czarnego króla jeżeli oba króle stoją w tym samym wierszu bądź kolumnie, w przeciwnym przypadku 0 $\{0,1,2,3,4,5,6,7\}$,
- *kOnEdge* – określenie czy czarny król znajduje się na krawędzi szachownicy $\{0,1\}$,
- *kInCorner* – określenie czy czarny król znajduje się w jednym z czterech rogów szachownicy $\{0,1\}$,
- *kInCheck* – określenie czy czarny król znajduje się w szachu $\{0,1\}$,
- *kCanMove* – określenie czy czarny król może wykonać posunięcie $\{0,1\}$,
- *ROnEdge* – określenie czy biała wieża znajduje się na krawędzi szachownicy $\{0,1\}$.

Na Rysunku 11. znajduje się przykładowa pozycja typu KRK wraz z odpowiednim wartościowaniem wszystkich atrybutów.



$KFile = c$ $KRank = 1$ $RFile = a$
$RRank = 3$ $kFile = a$ $kRank = 1$
$KMinDistanceFile = 0$ $KMinDistanRank = 2$
$RMinDistanceFile = 2$ $RMinDistanceRank = 0$
$kMinDistanceFile = 0$ $kMinDistanceRank = 0$
$KkFile = 0$ $KkRank = 1$ $KRFile = 0$
$KRRank = 0$ $kRFile = 1$ $kRRank = 0$
$KkDistanceFile = 0$ $KkDistanceRank = 2$
$KRDistanceFile = 2$ $KRDistanceRank = 2$
$kRDistanceFile = 2$ $kRDistanceRank = 0$
$KkOppositionDistance = 2$ $kOnEdge = 1$
$kInCorner = 1$ $kInCheck = 1$ $kCanMove = 0$
$ROnEdge = 1$ $optimalDepth = zero$

Rysunek 11. Przykładowa pozycja typu KRK oraz wartościowanie jej atrybutów.

Dla tak zdefiniowanych atrybutów zaimplementowano w klasie *KRKTranslator* (w katalogu *NeuralChess/auxilliary*) algorytm, który dla każdej pozycji ze zbioru uczącego wygenerował ten sam przykład opisany ostatecznie wartościami trzydziestu jeden atrybutów (*NeuralChess/auxilliary/tree03.arff*).

Zbiór uczący został już zdefiniowany i pozostało jedynie wybrać algorytm mający wyuczyć się jak najdokładniejszych reguł. Zdecydowano, iż najlepszym wyborem będzie algorytm ID3 generujący drzewo decyzyjne, które będzie można w miarę łatwo i efektywnie wykorzystać w dalszej implementacji. Rozwiązywany problem jest jak najbardziej wskazany dla drzew decyzyjnych, ponieważ wszystkie przykłady opisane są przez stały zbiór par atrybut-wartość i wszystkie atrybuty mają niewielkie zbiory możliwych wartości. Funkcja docelowa (klasyfikacja) przyjmuje dyskretne wartości. Wyprowadzone drzewo decyzyjne nie wymaga żadnego oczyszczania ani skracania, a zjawisko zbytniego dopasowania jest nawet jak najbardziej wskazane. Wygenerowane drzewo decyzyjne ma zapewnić 100% skuteczność klasyfikacji wszystkich pozycji ze zbioru uczącego.

Do wyprowadzenia drzewa decyzyjnego algorytmem ID3 wykorzystano darmowe oprogramowanie Weka 3.4.7. Przed uruchomieniem algorytmu usunięto ostatecznie sześć pierwszych atrybutów związanych z bezpośrednią lokalizacją figur, aby zapewnić wydobycie jak najogólniejszej wiedzy. Wygenerowane drzewo decyzyjne charakteryzowało się 100% dopasowaniem ze zbiorem uczącym (Rysunek 12.). Okazało się dodatkowo, że spośród wszystkich zdefiniowanych atrybutów tylko jeden był zbyteczny – *kInCheck*, który nie został wykorzystany. Jeden z głównych celi został już w tym momencie osiągnięty, tzn. uzyskano drzewo decyzyjne, które dość ogólnymi własnościami dobrze klasyfikowało zbiór uczący (*NeuralChess/auxilliary/tree03.id3*). Kolejnym krokiem miało być wygenerowanie na podstawie tego drzewa strategii gry, a więc zaimplementowanie algorytmu będącego translatorem pomiędzy drzewem decyzyjnym a kodem źródłowym w języku Java.

```
KMinDistanceFile = 0
|  KkDistanceFile = 0
|  |  KMinDistanceRank = 0
|  |  |  KRDistanceFile = 0
|  |  |  |  kRDistanceRank = 0: null
|  |  |  |  kRDistanceRank = 1
|  |  |  |  |  KRDistanceRank = 0: null
|  |  |  |  |  KRDistanceRank = 1: fifteen
|  |  |  |  |  KRDistanceRank = 2: draw
|  |  |  |  |  KRDistanceRank = 3: draw
|  |  |  |  |  KRDistanceRank = 4: draw
|  |  |  |  |  KRDistanceRank = 5: draw
|  |  |  |  |  KRDistanceRank = 6: draw
|  |  |  |  |  KRDistanceRank = 7: draw
|  |  |  |  kRDistanceRank = 2
|  |  |  |  |  KkDistanceRank = 0: null
|  |  |  |  |  KkDistanceRank = 1: null
|  |  |  |  |  KkDistanceRank = 2: fifteen
|  |  |  |  |  KkDistanceRank = 3
|  |  |  |  |  |  RMinDistanceRank = 0: null
|  |  |  |  |  |  RMinDistanceRank = 1: fifteen
|  |  |  |  |  |  RMinDistanceRank = 2: fourteen
```

Rysunek 12. Fragment wygenerowanego przez algorytm ID3 drzewa decyzyjnego.

Niestety w tym momencie zaczęły się pierwsze większe problemy, ponieważ wygenerowane drzewo składało się z ponad 50 tysięcy węzłów i dużą trudność sprawiło przełożenie go na kod źródłowy (definicja jednej metody w języku Java nie może przekraczać 65KB). Na szczęście część z tych węzłów była liśćmi zaetykietowanymi wartościami null, a więc wartościami nic nie znaczącymi. Dlatego też w pierwszej kolejności przystąpiono do usunięcia pustych liści (Rysunek 13., *NeuralChess/auxilliary/tree03a.id3*).

```
KMinDistanceFile = 0
|  KkDistanceFile = 0
|  |  KMinDistanceRank = 0
|  |  |  KRDistanceFile = 0
|  |  |  |  kRDistanceRank = 1
|  |  |  |  |  KRDistanceRank = 1: fifteen
|  |  |  |  |  KRDistanceRank = 2: draw
|  |  |  |  |  KRDistanceRank = 3: draw
|  |  |  |  |  KRDistanceRank = 4: draw
|  |  |  |  |  KRDistanceRank = 5: draw
|  |  |  |  |  KRDistanceRank = 6: draw
|  |  |  |  |  KRDistanceRank = 7: draw
|  |  |  |  kRDistanceRank = 2
|  |  |  |  |  KkDistanceRank = 2: fifteen
|  |  |  |  |  KkDistanceRank = 3
|  |  |  |  |  |  RMinDistanceRank = 1: fifteen
|  |  |  |  |  |  RMinDistanceRank = 2: fourteen
```

Rysunek 13. Fragment wyczyszczonego z pustych liści drzewa decyzyjnego

Powyższy zabieg pozwolił zredukować drzewo do około 25 tysięcy znaczących węzłów, co pozostawało nadal dość mocnym wyzwaniem do dalszego przełożenia go na kod źródłowy. Zadecydowano ostatecznie wygenerować kod źródłowy rozdzielony na kilka definicji funkcji i zaimplementowano algorytm, który je wygenerował w oddzielnych plikach z rozszerzeniem .code. Ostatecznie strategia gry została tak zaimplementowana, iż pierwsze dwa poziomy decyzyjne drzewa znajdowały się w jednym ciele funkcji *predictBoard* klasy *KRKPlayer*, a wszystkie następne były rozłożone w wywołaniach kolejnych funkcji rozpoczynających się od słowa kluczowego *node* (Kod 4.).

```
public int predictBoard ( ChessBoard cb )
{
	switch ( cb.KMinDistanceFile() )
	{
		case 0:
			switch ( cb.KkDistanceFile() )
			{
				case 0: return node00(cb);
				case 1: return node01(cb);
				case 2: return node02(cb);
				case 3: return node03(cb);
				case 4: return node04(cb);
				case 5: return node05(cb);
				case 6: return node06(cb);
				case 7: return node07(cb);
			}
			break;
		case 1:
			switch ( cb.KkDistanceFile() )
			{
				case 0: return node10(cb);
				case 1: return node11(cb);
				case 2: return node12(cb);
				case 3: return node13(cb);
				case 4: return node14(cb);
				case 5: return node15(cb);
				case 6: return node16(cb);
			}
			break;
		case 2:
			switch ( cb.kMinDistanceFile() )
			{
				case 0:	return node20(cb);
				case 1: return node21(cb);
				case 2: return node22(cb);
				case 3: return node23(cb);
			}
			break;
		case 3:
			switch ( cb.kMinDistanceFile() )
			{
				case 0: return node30(cb);
				case 1: return node31(cb);
				case 2: return node32(cb);
				case 3: return node33(cb);
			}
			break;
	}
	return -1;
}
```

Kod 4. Definicja metody predictBoard klasy KRKPlayer.

Wyniki badań

Po utworzeniu kodu źródłowego wygrywającej strategii gry przystąpiono do krótkiego eksperymentu mającego ocenić skuteczność wydobytych reguł. Korzystając z aplikacji testowej przeprowadzono testy dla 100 losowo wybranych końcówek typu KRK, w których białe zawsze zaczynają (a więc zawsze wygrywają). Niestety wyniki były niezadowalające. Okazało się, że program potrafił wygrać jedynie 72% pozycji (czasami w sposób nieoptymalny, tzn. nie najkrótszy), co w tej sytuacji było niedopuszczalne. Podczas rozgrywania kolejnych partii nasuwały się pewne spostrzeżenia, iż program najprawdopodobniej potrafi wygrywać wszystkie końcówki, w których do wygrania potrzeba około 4-5 optymalnych ruchów. W reszcie końcówek potrafił grać nieoptymalnie, doprowadzając czasami do wygrania, a czasami uzyskując tylko remis. Nie starczyło jednak czasu, aby dokładnie przetestować granicę maksymalnej liczby ruchów, do której program potrafi w sposób optymalny wygrywać te końcówki.

Wnioski

Można powiedzieć, iż druga część projektu zakończyła się niepowodzeniem, ponieważ najważniejszy cel stworzenia zawsze wygrywającej strategii nie został osiągnięty. Kroki, które zostały podjęte pozwoliły wygenerować jedynie strategię średnio-skuteczną i niech to zatem będzie połowiczny sukces projektu. Nieco mylące mogły okazać się atrybuty typu *MinDistanceFile* i *MinDistanceRank* dla każdej figury, ponieważ z jednej strony pozwoliły one uciec od bezpośredniego lokalizowania figur (tzn. na podstawie wiersza i kolumny), ale z drugiej strony mogły powodować pewne zamieszanie. Na przykład wartość 3 atrybutu *KMinDistanceFile* może oznaczać, że biały król stoi na czwartej albo na piątej linii i jeżeli dodatkowo atrybut *kMinDistanceFile* przyjmuje również wartość 3, to możliwe są cztery przypadki rozstawienia króli, tzn. oba stoją w wierszu piątym, albo oba stoją w wierszu czwartym, albo jeden stoi w wierszu czwartym, a drugi w wierszu piątym i na odwrót. Atrybut *KkDistanceFile* może jedynie zredukować powyższe cztery przypadki do dwóch, odpowiednio dla wartości 0 oraz 1.

W celu polepszenia uzyskanych wyników pozostaje do wykonania jeszcze kilka eksperymentów, z których co najmniej jeden wydaje się, że powinien doprowadzić do oczekiwanej najlepszej strategii. Oczywiście poza dodatkowymi przemyśleniami nad mocniejszymi atrybutami dla przykładów, można na pewno wypróbować jeden z następujących przypadków (co prawda nie najładniejszych):

- wygenerować wszystkie 250 tysięcy przykładów i na ich podstawie (wraz z zaproponowanymi tutaj atrybutami) starać się wygenerować drzewo decyzyjne. Ciężko stwierdzić jak długo trwałby proces generowania takiego drzewa algorytmem ID3 przy niemalże 10-krotnie większej bazie danych (w przypadku projektu generacja drzewa kończona była w przeciągu maksymalnie 10 sekund),
- wygenerować drzewo decyzyjne na podstawie siedmiu pierwotnych atrybutów, a więc *KFile, KRank, RFile, RRank, kFile, kRank* oraz *optimalDepth* i zaimplementować w programie algorytmy doprowadzające każdą pozycję KRK do jej kanonicznego odpowiednika i wówczas wybierać odpowiednio skonwertowany najlepszy ruch.

Literatura

[1] Richard S. Sutton. *Learning to Predict by the Methods of Temporal Differences*, Machine Learning 3: 9-44, 1988 Kluwer Academic Publishers, Boston – Manufactured in The Netherlands.

[2] Johannes Fürnkranz. *Machine Learning in Computer Chess: The Next Generation*, Austrian Research Institute for Artificial Intelligence.

[3] Michael Bain. *Learning Logical Exceptions in Chess*. Glasgow, 1994.