Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

Kierunek: **INA**
Specjalność: **-**

Praca Dyplomowa Inżynierska

Algorytm Minimax dla gry w Warcaby

Janusz Witkowski

Opiekun pracy
dr Maciej Gębala

Sztuczna inteligencja, Algorytmika, Algorytm genetyczny, Teoria gier

Wrocław 2022

Streszczenie

W pracy zaimplementowano algorytm Minimax z cięciami alfa-beta do gry w warcaby w wariancie angielskim. Algorytm używa funkcji oceny heurystycznej rozpatrującej parametry danego stanu rozgrywki. Do wyznaczenia odpowiednich priorytetów parametrów wykorzystano algorytm genetyczny.

Abstract

The thesis is concentrated on implementing Minimax algorithm with alpha-beta-pruning for the game of English draughts (American Checkers). It involves a heuristic function which considers some number of parameters of given game state. To find suitable priorities for these parameters, a genetic algorithm was used.

Spis treści

Spis rysunków

Spis tabel

Wstęp

Tematem pracy jest prosty model sztucznej inteligencji do gry w warcaby w wariancie angielskim, oparty o algorytm decyzyjny Minimax oraz funkcję oceny heurystycznej stanu gry. Celem pracy było wyznaczenie jak najlepszej funkcji oceny stanowiącej intuicję algorytmu, oraz sprawdzenie różnic między możliwymi do obrania w Minimaksie perspektywami. Aby przeprowadzić takie badania, zaimplementowano dopasowany do problematyki algorytm genetyczny i uruchomiono kilka jego sesji na przestrzeni paru miesięcy.

Praca podzielona jest na 4 główne rozdziały:

1. Warcaby. Rozdział ten zawiera wprowadzenie do gry w warcaby, omówienie reguł gry oraz definicję wariantu angielskiego tej gry. Zawarto również krótką informację na temat dotychczasowych badań warcabów w tym wariancie.

2. Idea rozwiązania i algorytmy. Określone tu zostały wykorzystane w pracy algorytmy, jak i cele ich użycia. Omawiane są po kolei algorytm Minimax, funkcja oceny heurystycznej i algorytm genetyczny. Rozdział wzbogacony jest również o rysunki ilustrujące zachowania algorytmów.

3. Implementacja. Rozdział poświęcony jest wdrożeniu omawianych wcześniej algorytmów w rzeczywisty projekt napisany w konkretnym języku programowania. Podane są ogólne zadania i funkcjonalności najważniejszych części projektu. Na końcu rozdziału zapisano instrukcję obsługi programów wykonawczych.

4. Wyniki i rozszerzenia. W ostatnim rozdziale znajdują się opisy przeprowadzonych eksperymentów, wyniki tych eksperymentów oraz wnioski. Dodatkowo w osobnym podrozdziale zawarto pomysły na rozszerzenie projektu i badań.

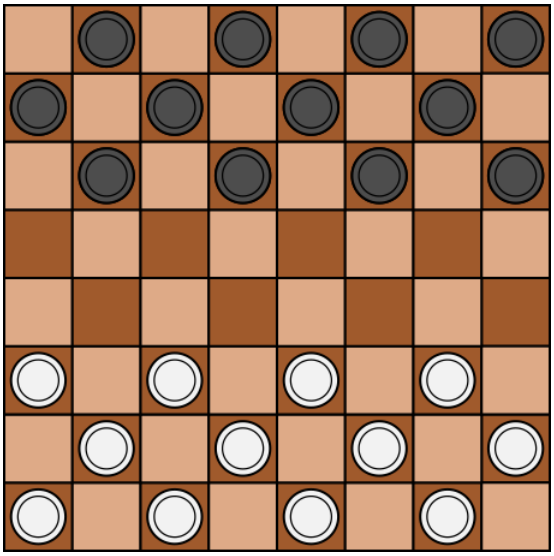
Rozdział 1

Warcaby

Warcaby to jedna z najpopularniejszych klasycznych gier dwuosobowych, zaliczanych do gier z doskonałą informacją i gier o sumie zerowej. Przyjmuje się, że gra ta zrodziła się w XII wieku, najprawdopodobniej na południu Francji lub w Hiszpanii, oraz że wywodzi się ona z dawnej arabskiej gry Alquerque [5]. Istnieją coroczne mistrzostwa i turnieje światowe w różnych odmianach warcabów, choć dzięki nieskomplikowanym zasadom są one popularne również w mniejszych kręgach.

1.1 Reguły warcabów standardowych

Pojedynczą partię warcabów rozgrywa się na szachownicy 8x8 o polach na zmianę pomalowanych na jasno lub ciemno (Rys. 1.1). W grze wykorzystywane są dwa rodzaje figur - piony i damki. Obydwaj gracze rozstawiają po 12 pionów na ciemnych polach w swoich pierwszych trzech rzędach. Dla rozróżnienia, piony pierwszego gracza są koloru białego, natomiast piony drugiego gracza - czarnego. Celem gry jest wyeliminowanie wszystkich figur przeciwnika lub zablokowanie go, poprzez serię naprzemiennych ruchów swoimi figurami. Zablokowanie gracza oznacza doprowadzenie do takiej sytuacji, w której gracz ten nie jest w stanie wykonać żadnego legalnego ruchu, w momencie gdy następuje jego kolej.



Rysunek 1.1: Stan początkowy planszy w warcabach.

Wszystkie figury w grze mogą poruszać się tylko i wyłącznie na ukos (przez co żadne jasne pole na planszy nie zostanie zajęte przez żadną figurę w trakcie rozgrywki). Piony z którymi zaczynają gracze poruszają się tylko o jedno pole w przód względem ich właściciela. Tzn. pion może skoczyć na pole ukośnie

sąsiadujące w kierunku oponenta, o ile pole to nie jest zajęte przez inną figurę. W grze istnieje również drugi typ figury - jeżeli pion gracza dojdzie do końca planszy znajdującego się po stronie jego oponenta (do tzw. rzędu awansu), pion zamieniany jest na damkę. Damka jest najpotężniejszą figurą w grze, jako że potrafi poruszać się we wszystkich czterech kierunkach na ukos, a na dodatek przebyć więcej niż jedno pole w linii w jednym ruchu. Pod tym względem damkę najłatwiej porównać z figurą gońca w szachach.

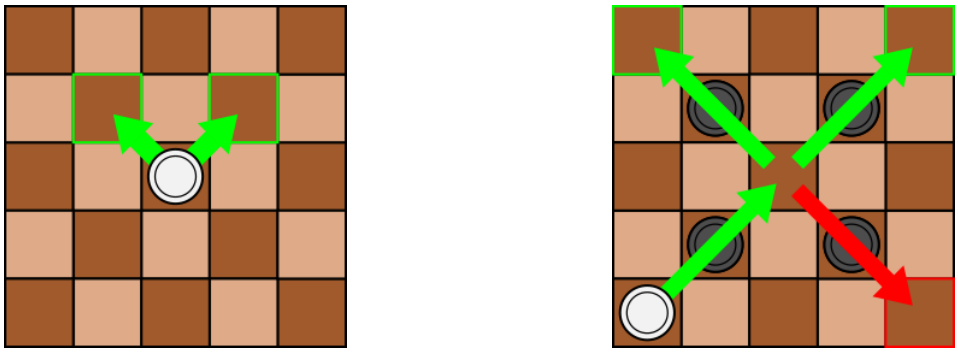
Każdą figurę w warcabach można zbijać, tj. usuwać z obecnej rozgrywki. Piony mogą zbijać sąsiednie figury przeciwnika, wykonując skok nad tą figurą na następne pole w linii prostej, o ile takie pole jest wolne. Piony mogą bić zarówno do przodu, jak i do tyłu. Damki w standardowych warcabach zbijają ze znacznie większego dystansu (można powiedzieć, że w momencie zbijania ich „sąsiadowanie" z przeciwnymi figurami nie jest ograniczone do jednego pola różnicy). Zbita figura zostaje zdjęta z planszy i nie bierze już udziału w rozgrywce. Nie można bić swoich figur.

Warcaby mają specjalne reguły bicia wyróżniające je spośród innych gier planszowych. Po pierwsze, w jednym ruchu jedna figura może wykonać wiele bić. Jeśli po jednym biciu figura wskoczyła w miejsce, z którego jest w stanie przeprowadzić kolejne bicie, można takie bicie wykonać w tej samej turze. W jednym ruchu nie można dwa razy zbić tej samej figury. Po drugie, bicia są obowiązkowe. Jeżeli gracz w swojej turze jest postawiony w sytuacji, w której co najmniej jedna z jego figur ma możliwość bicia, gracz ten musi wykonać taki ruch. Jeżeli więcej niż jedna figura gracza może wykonać bicie, gracz decyduje którą z tych figur się rusza. Dodatkowo, piony mają większą swobodę w ruchach bijących niż damki. Jeden pion może przeprowadzić dowolny z kilku różnych ruchów bijących, które jest w stanie przeprowadzić. Damki natomiast mają obowiązek maksymalnego bicia, tj. należy wykonać bicie o największej możliwej liczbie zbijanych figur w jednym ruchu.

1.2 Wariant angielski

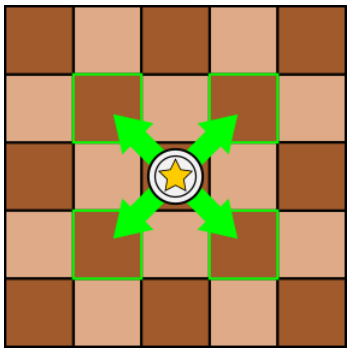
Praca skupiona jest na szczególnej wersji gry w warcaby, nazywanej na ogół wariantem angielskim lub w niektórych kręgach wariantem amerykańskim. Został on wybrany głównie ze względu na ograniczenie przestrzeni stanów, w jakich może znaleźć się rozgrywka - reguły gry dostosowane do tego wariantu znacznie zmniejszają liczbę możliwości, które rozgrywający algorytm musi rozpatrzeć.

Wariant ten wprowadza dwie zmiany do zasad gry względem wariantu standardowego (opierając się o [5]). Po pierwsze, piony nie mogą bić do tyłu. Po drugie, damkom ogranicza się możliwość ruchu do jednego sąsiedniego pola oraz do bicia wyłącznie sąsiadujących przeciwnych figur, lecz wciąż mogą poruszać się we wszystkich kierunkach na ukos. Jedyną przewagą damek nad pionkami w tym wariancie jest możliwość ruchu i bicia do tyłu.

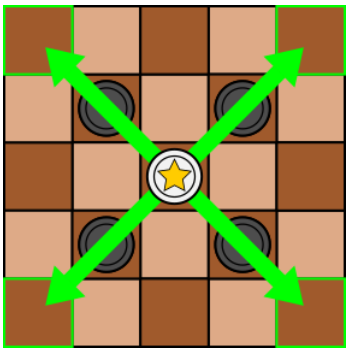


(a) Legalne ruchy (b) Legalne bicia

Rysunek 1.2: Zestaw możliwych ruchów dla piona w wariancie angielskim



(a) Legalne ruchy



(b) Legalne bicia

Rysunek 1.3: Zestaw możliwych ruchów dla damki w wariancie angielskim

Przydatne w implementacji jest rozpatrywanie remisów. W grach towarzyskich remis zazwyczaj następuje za obopólną zgodą graczy. Na arenie turniejowej istnieje parę kryteriów determinujących remis. W rozpatrywanej wersji wariantu angielskiego wykorzystywana będzie zasada 40 ruchów: rozgrywka kończy się remisem, gdy w 40 naprzemiennych ruchach obu graczy nie została zbita ani jedna figura.

1.2.1 Badania wariantu

Do rozwoju badań nad warcabami w wariancie angielskim najmocniej przyczynił się Jonathan Schaeffer, profesor Uniwersytetu Alberty w Kanadzie. Jego zespół opracował Chinooka, przeszukujący w głąb program do grania w warcaby angielskie, który w roku 1992 oraz 1994 stanął naprzeciw ówczesnego mistrza świata Mariona Tinsleya i ogłoszony został pierwszym komputerowym zwycięzcą mistrzostw [1]. Następnie, z pomocą programu, udowodnili poprzez słabe rozwiązanie (*weakly solved*, pojęcie omówione w [8]), że każda gra w warcabach angielskich kończy się remisem, pod warunkiem że gracze wykonują ruchy doskonale [7]. Pomimo uproszczonych zasad względem klasycznej wersji, wariant angielski posiada przestrzeń stanów wielkości rzędu 10^{20}, dlatego też rozwiązanie zajęło zespołowi Schaeffera około 18 lat na 200 równolegle liczących maszynach.

Rozdział 2

Idea rozwiązania i algorytmy

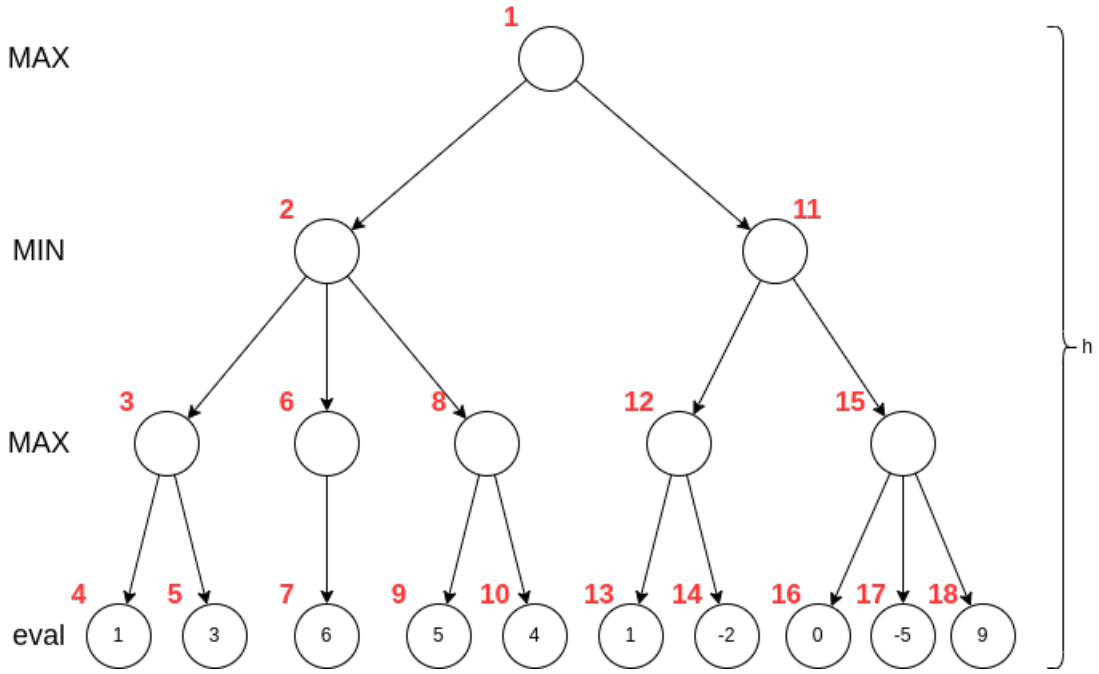
Głównym celem pracy jest stworzenie prostego modelu sztucznej inteligencji do grania w warcaby w wariancie angielskim z pewną strategią. Istnieją różne podejścia do tego zagadnienia, spośród których najpopularniejszymi są te stosujące sieci neuronowe (zestawem danych byłaby np. baza meczów rozegranych na mistrzostwach na przestrzeni kilkunastu lat). Praca skupia się na algorytmie Minimax połączonym z funkcją oceny heurystycznej. Do częściowego wyznaczenia funkcji oceny wykorzystano odmianę algorytmu genetycznego.

2.1 Minimax

Minimax jest szczególną wersją algorytmu przeszukującego w grafie. Jego idea jest bardzo prosta i zbliżona do ludzkiego rozumowania. Mając dany stan planszy oraz głębokość przeszukiwania, algorytm rekurencyjnie rozpatruje kolejne stany planszy symulując wykonanie jednego możliwego ruchu (Rys. 2.1). Kiedy już osiągnie maksymalną głębokość przeszukiwań na którymś stanie, funkcją oceny heurystycznej przypisuje wartość do tego stanu, po czym zwraca tę wartość do stanu-rodzica. Mając wartości oceny od każdego swojego dziecka, stan wybiera jedną z nich i przekazuje ją do swojego rodzica. Gdy wybór dojdzie do stanu będącego korzeniem drzewa przeszukiwań, algorytm wybierze jedną z dostępnych mu ocen i zwróci ruch do stanu, któremu ta ocena odpowiada.

Pseudokod 2.1: Prosty algorytm Minimax

```
Input: Stan gry state, flaga gracza perspective, głębokość przeszukiwań h, rozpatrujący gracz player
Output: Wartość funkcji oceny eval
1  if h = 0 then
2      eval ← evaluateState(state, player);
3  else
4      if perspective = MAX then
5          maxEval ← −∞;
6          foreach child ∈ getChildren(state) do
7              childEval ← minimax(child, MIN, h - 1, player);
8              if childEval ⩾ maxEval then
9                  maxEval ← childEval
10         eval ← maxEval
11     else
12         minEval ← +∞;
13         foreach child ∈ getChildren(state) do
14             childEval ← minimax(child, MAX, h - 1, player);
15             if childEval ⩽ minEval then
16                 minEval ← childEval
17         eval ← minEval
```



Rysunek 2.1: Kolejność rozpatrywania stanów przez Minimax dla h = 3

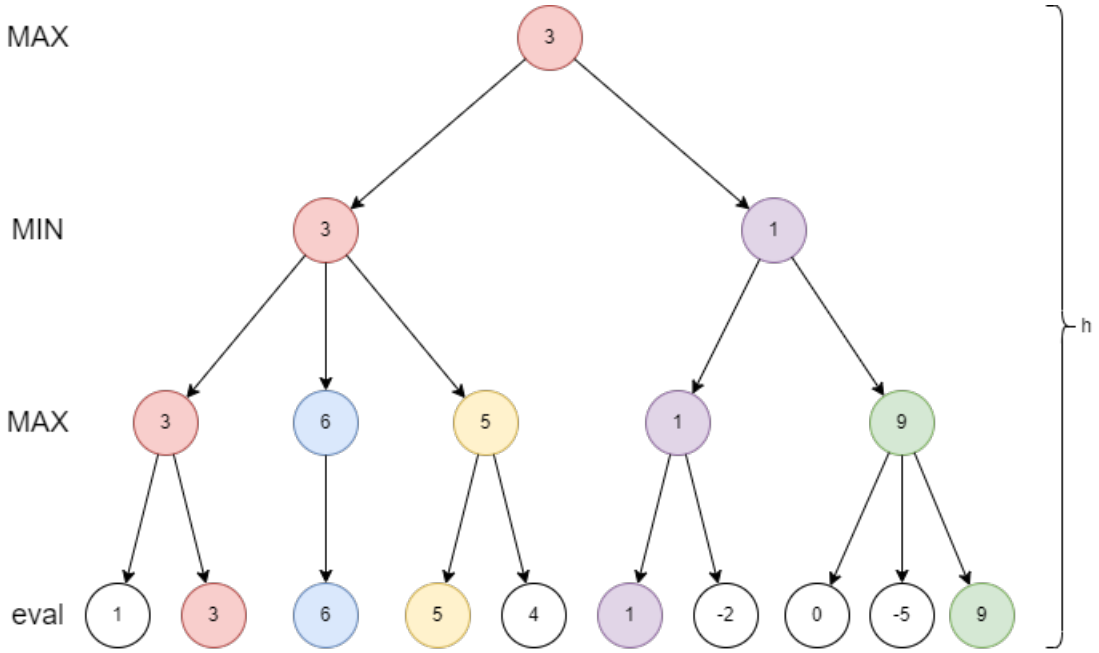
Swoją nazwę algorytm zawdzięcza naprzemiennemu rozpatrywaniu ocen heurystycznych w drzewie przeszukiwań. W momencie gdy dany gracz wywołuje procedurę Minimaxa rozpatrując możliwe do wykonania przez niego ruchy, oznaczany jest jako gracz MAX. W powstałych stanach gry gracz symuluje tok rozumowania jego przeciwnika, rozpatrując ruchy za niego i oznaczając go jako gracza MIN. Stany na kolejnym poziomie w drzewie przeszukiwań rozpatruje gracz MAX, i tak dalej. Gdy gracz MAX dokonuje wyboru oceny do przekazania do stanu-rodzica, wybiera maksymalną wartość. Analogicznie, gracz MIN wybiera ocenę o minimalnej wartości. (Rys. 2.2)

2.1.1 Alpha-Beta-pruning

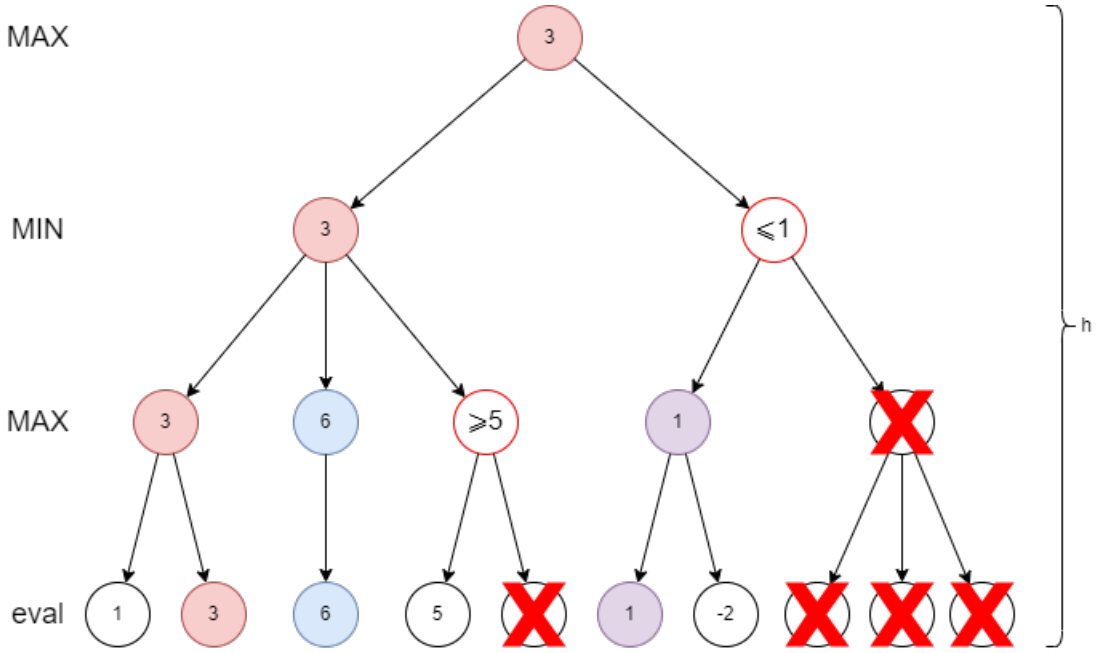
Patrząc jeszcze raz na rys. 2.1 i rys. 2.2 można zauważyć, że posiadając wiedzę o wartościach ocen odwiedzonych już stanów, nie ma potrzeby rozpatrywania niektórych następujących stanów. Na przykład w stanie numer 8, wiedząc że jego pierwsze dziecko otrzymało ocenę 5, można być pewnym że MAX dla tego stanu wybierze stan o wartości co najmniej 5, co sprawi że gracz MIN nad nim będzie miał do rozpatrzenia stany o wartościach dokładnie 3, dokładnie 6, co najmniej 5. Jako że gracz ten minimalizuje, wybierze stan o wartości 3. Wynika stąd, że sprawdzenie drugiego dziecka stanu numer 8 byłoby niepotrzebne, ponieważ nie wpłynęłoby na ostateczny wynik. Można też obciąć całe poddrzewo z korzeniem w stanie numer 15 - wybór gracza MIN będzie miał ocenę ograniczoną z góry przez 1, więc gracz MAX na pewno nie poprawi wyniku idąc tą ścieżką, wiedząc że ma obok stan o lepszej ocenie. Zaoszczędzono w ten sposób zasoby na przejrzenie 5 stanów, w tym jednego poddrzewa. (Rys. 2.3)

Taką optymalizację nazywa się cięciami alfa-beta [6]. Są to rezygnacje z rozpatrywania innych podgałęzi ze względu na brak możliwości poprawienia wyniku. Zoptymalizowany w ten sposób Minimax w trakcie działania operuje dwiema wartościami: α i β. α reprezentuje najwyższą ocenę znaną w poddrzewie, β natomiast - najniższą. Wartości α i β aktualizowane są w poddrzewach przez odpowiednio gracza MAX i gracza MIN. W momencie gdy $\beta \leqslant \alpha$, rekurencyjne wywołanie Minimaxa w poddrzewie zostaje zakończone i zwracana jest najlepsza (zdaniem gracza) wartość.

Wadą cięć alfa-beta jest ich zależność od kolejności rozpatrywania węzłów - gdyby algorytm w przykładzie 2.2 wpierw przeszedł przez poddrzewo numer 15, musiałby przejrzeć jeszcze poddrzewo numer 12, bo nie byłby pewny co do możliwości poprawienia wyniku dla rodzica. Mankament ten sprawia, że



Rysunek 2.2: Wynik przebiegu Minimaxa dla przykładu dla h = 3



Rysunek 2.3: Zastosowanie cięć alfa-beta na przykładzie

stosunku liczby optymalizowanych stanów do całego poddrzewa nie da się jednoznacznie określić.

2.2 Funkcja oceny heurystycznej

Ze względu na ogromny rozmiar przestrzeni stanów w warcabach, obecne komputery nie potrafią przeszukać jej całości w „rozsądnym" czasie. Jeśli jednym z celów budowania sztucznej inteligencji są w miarę szybkie decyzje prowadzące do zwycięstwa w grze, należy skrócić przeszukiwanie przestrzeni stanów i w jakiś sposób obejść się z niedoskonałą informacją posiadaną przez komputer. Ocena danego stanu na planszy ma wspomóc taki komputer w wyrobieniu „intuicji" poprzez analizę sytuacji na planszy.

Podejście w pracy do funkcji oceny heurystycznej polega na rozpatrzeniu wielu parametrów na planszy (np. liczba pionów, liczba damek przeciwnika, liczba ruchów), przemnożeniu wartości tych parametrów przez ustalone z góry wagi, a na koniec zsumowaniu powstałych iloczynów. Suma tych iloczynów to wartość funkcji oceny, którą przypisuje się pod dany stan gry.

Wartość funkcji oceny heurystycznej można określić następującym wzorem: $\sum_{i=1}^{n}(param_i * weight_i)$.

Funkcja oceny heurystycznej zależy oczywiście od podjętej strategii oraz od podejścia do problemu. Opisana wyżej funkcja ma parę zalet, na których opiera się praca. Po pierwsze, wyznaczenie oceny z wyliczonymi już wartościami parametrów odbywa się szybko, bo w czasie liniowym (nie uwzględnia to czasu potrzebnego do rozpatrzenia tychże parametrów). Po drugie, listowanie parametrów w ten sposób pozwala na przeprowadzenie eksperymentów i wyciągnięcie wniosków na temat teorii gry w warcaby, o czym w poniższym podrozdziale.

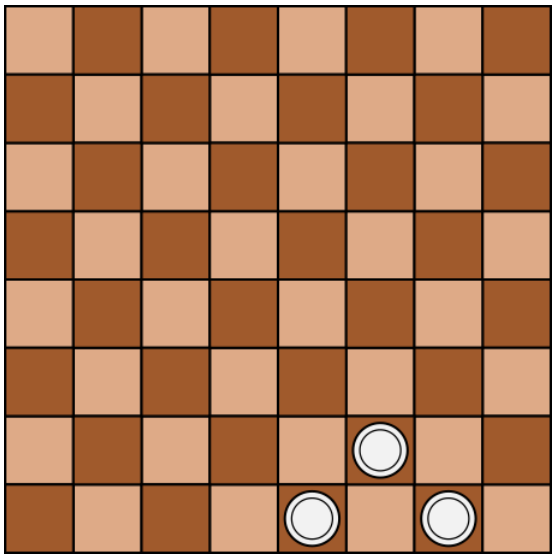
2.2.1 Wykorzystywane parametry

W tabeli 2.1 przedstawione są wszystkie wykorzystywane w pracy parametry. Są to liczby i flagi wyciągane z danego stanu rozgrywki i wykorzystywane do obliczenia wartości funkcji oceny heurystycznej dla tego stanu. Duża część tych parametrów to symetryczne odbicia innych parametrów względem figury (pion/damka) lub względem gracza (MAX/MIN). Poniżej znajduje się krótkie objaśnienie niektórych parametrów.

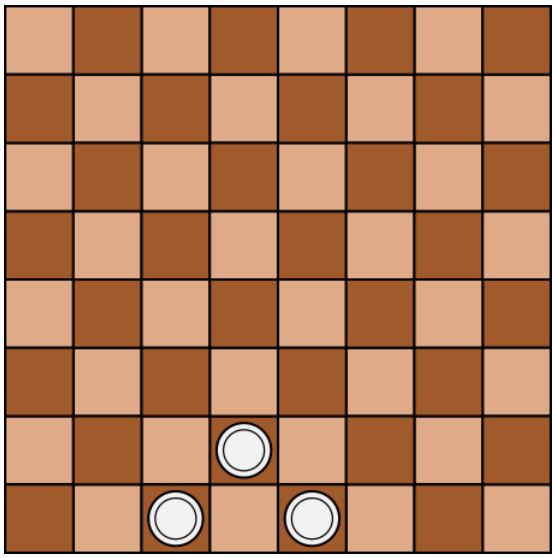
- **[9-12] Ruchome figury** to figury, które są w stanie wykonać co najmniej jeden ruch (w tym bicie).
- **[13-14] Możliwe ruchy graczy** są obliczane nawet jeśli w rozpatrywanym stanie tura nie należy do tego gracza.
- **[21-24] Rzędem awansu gracza** nazywamy rząd, w którym pion gracza awansuje do damki. **dystansem piona do rzędu** nazywamy liczbę pól w linii prostej od piona do rzędu.
- **[25-36] Dolne, środkowe i górne rzędy** gracza / przeciwnika rozważa się z perspektywy gracza / przeciwnika. **Dolne rzędy** gracza oznaczają dwa rzędy najbliżej gracza, **górne rzędy** gracza to trzy rzędy najbliższe jego przeciwnika, natomiast **środkowymi rzędami** nazywa się dwa rzędy w centrum planszy.
- **[37-40] Samotne figury** to takie, które nie mają w najbliższym sąsiedztwie żadnej figury.
- **[15, 18, 41-54]** Niektóre parametry są **parametrami binarnymi** i jako wartość przyjmują 0 lub 1.
- **[47-54] Każdy wspomniany „pattern"** jest objaśniony na ilustracjach 2.4.
- **[55-60] Blokujące figury** w tych parametrach to takie, które znajdują się za sojuszniczą figurą i tym samym blokują możliwość zbicia chronionej figury. **Linia bloku** składa się z co najmniej dwóch figur danego gracza ustawionych obok siebie na przekątnej.

Nr	Parametr	Nr	Parametr
1	Liczba sojuszniczych pionów	31	Liczba przeciwnych pionów w środkowych rzędach
2	Liczba sojuszniczych damek	32	Liczba przeciwnych damek w środkowych wierszach
3	Liczba przeciwnych pionów	33	Liczba sojuszniczych pionów w górnych rzędach
4	Liczba przeciwnych damek	34	Liczba sojuszniczych damek w górnych rzędach
5	Liczba sojuszniczych pionów przy ścianie	35	Liczba przeciwnych pionów w górnych rzędach
6	Liczba sojuszniczych damek przy ścianie	36	Liczba przeciwnych damek w górnych rzędach
7	Liczba przeciwnych pionów przy ścianie	37	Liczba samotnych sojuszniczych pionów
8	Liczba przeciwnych damek przy ścianie	38	Liczba samotnych sojuszniczych damek
9	Liczba ruchomych pionów gracza	39	Liczba samotnych przeciwnych pionów
10	Liczba ruchomych damek gracza	40	Liczba samotnych przeciwnych damek
11	Liczba ruchomych pionów przeciwnika	41	Czy pion gracza jest w kącie
12	Liczba ruchomych damek przeciwnika	42	Czy damka gracza jest w kącie
13	Liczba możliwych ruchów gracza	43	Czy gracz zajmuje dwa kąty
14	Liczba możliwych ruchów przeciwnika	44	Czy pion przeciwnika jest w kącie
15	Istnienie bijącego ruchu gracza	45	Czy damka przeciwnika jest w kącie
16	Liczba bijących ruchów gracza	46	Czy przeciwnik zajmuje dwa kąty
17	Rozmiar najdłuższego bijącego ruchu gracza	47	Obecność *Triangle pattern* u gracza
18	Istnienie bijącego ruchu przeciwnika	48	Obecność *Oreo pattern* u gracza
19	Liczba bijących ruchów przeciwnika	49	Obecność *Bridge pattern* u gracza
20	Rozmiar najdłuższego bijącego ruchu przeciwnika	50	Obecność *Dog pattern* u gracza
21	Suma dystansów pionów gracza do rzędu awansu	51	Obecność *Triangle pattern* u przeciwnika
22	Suma dystansów pionów przeciwnika do rzędu awansu	52	Obecność *Oreo pattern* u przeciwnika
23	Liczba niezajętych pól w rzędzie awansu gracza	53	Obecność *Bridge pattern* u przeciwnika
24	Liczba niezajętych pól w rzędzie awansu przeciwnika	54	Obecność *Dog pattern* u przeciwnika
25	Liczba sojuszniczych pionów w dolnych rzędach	55	Liczba blokujących sojuszniczych figur
26	Liczba sojuszniczych damek w dolnych rzędach	56	Liczba linii bloku gracza
27	Liczba przeciwnych pionów w dolnych rzędach	57	Wielkość najdłuższej linii bloku gracza
28	Liczba przeciwnych damek w dolnych rzędach	58	Liczba blokujących przeciwnych figur
29	Liczba sojuszniczych pionów w środkowych rzędach	59	Liczba linii bloku przeciwnika
30	Liczba sojuszniczych damek w środkowych rzędach	60	Wielkość najdłuższej linii bloku przeciwnika

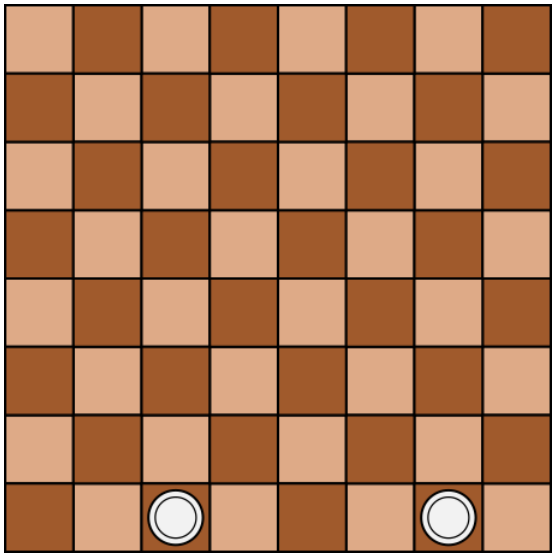
Tabela 2.1: Wszystkie parametry rozpatrywane w pracy. Część parametrów została zaczerpnięta z [4].



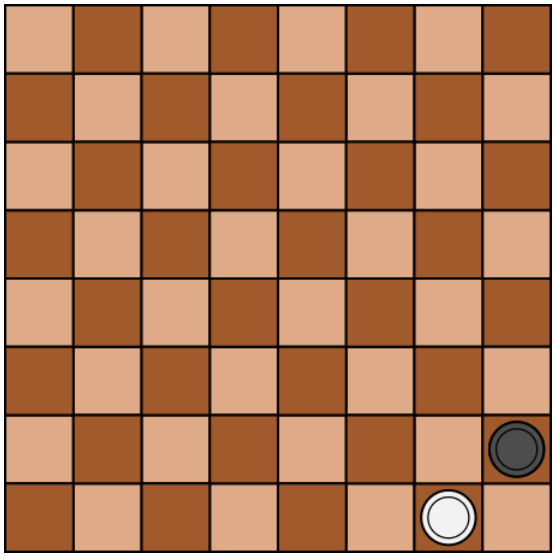
(a) Triangle pattern



(b) Oreo pattern



(c) Bridge pattern



(d) Dog pattern

Rysunek 2.4: Wzory na planszy względem gracza białego, wykorzystywane do niektórych parametrów

2.3 Algorytm genetyczny

Jednym z najważniejszych celów pracy jest znalezienie odpowiedniej strategii gry w warcaby w postaci dobrej funkcji oceny heurystycznej, co w tym przypadku sprowadza się do wyznaczenia jak najlepszego zestawu wartości wag, dalej zwanego ciągiem wag. Naiwnym podejściem do tego problemu byłoby przypisanie priorytetów każdego parametru stanu planszy przez człowieka bądź zespół ludzi. Może się jednak okazać, że pewne parametry nie będą tak wartościowe dla zwycięstwa jak podpowiedziałaby ludzka intuicja. Ponadto, zakładając że nie istnieje obiektywne spojrzenie na jakość danej strategii, należałoby przeprowadzić ogromną liczbę testów gry z człowiekiem w celu sprawdzenia poprawności wyznaczonych wag (tj. czy podane wartości ciągu wag są wystarczające by algorytm uznać za kompetytywnego gracza). Z pomocą tutaj przychodzi zastosowanie algorytmu genetycznego.

Algorytm genetyczny jest przedstawicielem klasy algorytmów metaheurystycznych. Konkretniej jest to odmiana algorytmu ewolucyjnego, który z kolei jest pochodną algorytmu populacyjnego. Metaheurystyka genetyczna symuluje zjawisko doboru naturalnego w przyrodzie, operując kolejnymi pokoleniami populacji osobników reprezentującymi potencjalne rozwiązania danego problemu, starając się odnaleźć rozwiązanie suboptymalne.

Najprostszy zarys algorytmu genetycznego można zobrazować w następujący sposób. W początkowej populacji losowo utworzonych osobników (rozwiązań), nazwaną pierwszym pokoleniem, dochodzi do selekcji, mającej na celu wyznaczenie populacji rodziców. W populacji rodziców dochodzi do wzajemnego krzyżowania i tworzenia się populacji dzieci, które dziedziczą po swoich rodzicach informacje genetyczne (tj. elementy rozwiązania). Wśród dzieci może z pewnym prawdopodobieństwem dojść do mutacji, która losowo zmienia jeden z elementów genotypu u osobnika. Dzieci, wraz z niektórymi rodzicami lub nowymi, losowymi osobnikami (zależnie od wersji), określa się nowym pokoleniem. W populacji nowego pokolenia powtarza się proces selekcji, krzyżowania i mutacji, aż do osiągnięcia z góry ustalonego warunku stopu (np. limitu czasowego). (Rys. 2.5)

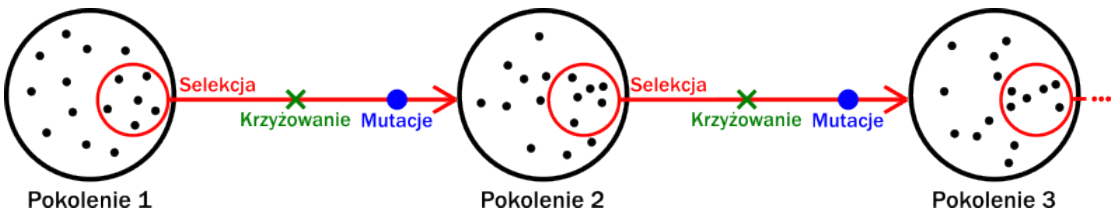
Biorąc pod uwagę fakt, że to od rodziców zależy jakość każdej populacji, szczególny nacisk należy położyć na fazę selekcji. Niezbyt intuicyjną, a ważną taktyką jest też wprowadzanie w pewne miejsca przebiegu algorytmu elementów losowości, aby w populacji nie dochodziło do tak zwanego zjawiska stagnacji (sytuacji, w której zbyt wiele osobników w populacji jest do siebie bardzo podobnych) oraz aby algorytm był w stanie znajdować inne, być może lepsze, rozwiązania.

2.3.1 Populacja i osobniki

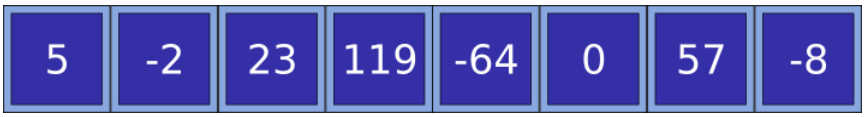
Osobnikiem populacji będzie ciąg wag funkcji oceny heurystycznej, reprezentowany jako tablica liczb całkowitych. Wagi będą mogły przyjmować zarówno dodatnie, jak i ujemne wartości. (Rys. 2.6)

2.3.2 Selekcja i ewaluacja

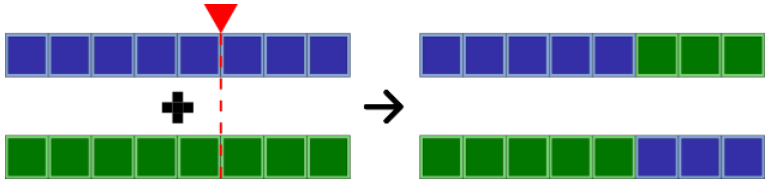
W wielu problemach, do których stosuje się algorytmy genetyczne, funkcja ewaluacji osobnika jest podana w treści problemu bądź łatwa do wyznaczenia. Tak jednak nie jest z problemem znalezienia najlepiej grającej sztucznej inteligencji. Dlatego też zastosowane zostało podejście lokalne, czyli zwracające uwagę na umiejętności osobników w danej populacji. W procesie selekcji każdy ciąg wag gra z każdym innym ciągiem wag w podwójnym pojedynku (białe/czarne i czarne/białe). Im więcej gier wygra dany ciąg wag, tym wyższe prawdopodobieństwo, że zostanie on wylosowany do populacji rodziców.



Rysunek 2.5: Ogólny przebieg algorytmu genetycznego



Rysunek 2.6: Uproszczony model osobnika (ciągu wag)



Rysunek 2.7: Model krzyżowania dwóch osobników w implementacji

2.3.3 Krzyżowanie i mutacja

W pracy zaimplementowano dobieranie rodziców w pary w których się wzajemnie krzyżują, produkując dwójkę dzieci. Aby zachować potencjalnie dobre informacje genetyczne, dzieli się ciągi wag rodziców w tym samym losowo wyznaczonym miejscu i zamienia się ze sobą części. Dzięki temu każde z dwójki dzieci przechowuje cechy po każdym swoim rodzicu. (Rys. 2.7)

Mutacja u nowo narodzonych osobników następuje z pewnym prawdopodobieństwem i wpływa na maksymalnie jedną wartość w ciągu wag. Mutowana waga zostaje zastąpiona losową wartością z odpowiedniego przedziału.

Rozdział 3

Implementacja

Kody źródłowe dołączone do niniejszej pracy (patrz Dodatek A) są wynikiem implementacji algorytmów opisanych w rozdziale 2. Przed ich omówieniem warto wspomnieć o dwóch niuansach, które miały wpływ na pisanie tej części pracy.

Po pierwsze, priorytetem w implementacji nie była szczegółowa optymalizacja. W założeniu czas przydzielony na przeprowadzenie sesji algorytmu genetycznego wynosił kilka miesięcy, natomiast ograniczenia maszynowe były zaniedbywane. Z tego powodu praca nie skupia się na wielu optymalizacjach, a projekt nie został napisany w języku bardzo wysokiej wydajności (jak C/C++, Rust).

Po drugie, interfejs użytkownika ograniczony jest do niezbędnego minimum. Interakcja z programami odbywa się z poziomu konsoli, nie ma też wielu elementów graficznych w projekcie. Mimo to, system posiada wszystkie funkcjonalności potrzebne do przeprowadzania gier i eksperymentów.

3.1 Język i środowisko

Projekt został napisany w języku Java. Jest to wygodne narzędzie pozwalające na naturalną strukturyzację projektu. Dzięki wirtualnej maszynie Javy (JVM) powstałe programy można uruchamiać na wszystkich wspieranych systemach operacyjnych, co ułatwia przeprowadzanie testów na większą skalę. Jest to też język dobrze zoptymalizowany - nie jest koniecznym przejmowanie się wydajnością „rozsądnie" napisanego kodu.

Struktura projektu tworzona była w duchu programowania obiektowego. Posiada ona klasy z podzielonymi funkcjonalnościami i odpowiedzialnościami. Zadbano również o przechwytywanie podstawowych wyjątków i błędów.

Projekt pisany był w środowisku Javy *OpenJDK 17.0.4 2022-07-19*, aczkolwiek wszystkie wykorzystywane funkcjonalności zawarte są w *OpenJDK 14*.

3.2 Struktura projektu

Struktura projektu składa się z kilkunastu klas, w tym z dwóch klas numerycznych (**HParam**, **StopCond**), dwóch klas statycznych (**RNG**, **Console**) i trzech programów wykonawczych (**Play**, **Find**, **Show**). Poniżej znajduje się ogólny opis odpowiedzialności i funkcjonalności ważniejszych encji w projekcie.

3.2.1 Klasa State

Reprezentacja pojedynczego stanu w przestrzeni stanów rozgrywki. Przechowuje informacje o planszy, numerze gracza do którego należy tura, liście możliwych ruchów tego gracza, ruchu poprzednim, ruchu który utworzył ten stan, oraz liczniku naprzemiennych ruchów bez zbicia (licznik potrzebny do określenia remisu). Wykorzystuje prywatne klasy rekordów **Pair** i **Node** do przekazywania odpowiednio współrzęd-

nych pól i drzewa możliwych bić. W kodach źródłowych instancje tej klasy często nazywa się po prostu „board".

Plansza w obiekcie **State** jest dwuwymiarową tablicą liczb całkowitych. Puste pola oznaczane są zerem, natomiast pola przynależne do gracza pierwszego lub gracza drugiego mają wartości odpowiednio dodatnie lub ujemne. Wartość bezwzględna pola z pionem wynosi 1, a pola z damką wynosi 2. Klasa obsługuje wykonywanie ruchów zarówno dla komputera jak i człowieka, wykorzystując do tego metody *makeMove* i *submitUserInput*. Każdy ruch aktualizuje potrzebne flagi i pola wewnątrz obiektu, dlatego też plansza jest bezpośrednio związana z tą klasą.

Kod źródłowy 3.1: Metoda w klasie State budująca drzewo możliwych bić

```
private void buildCaptureMove (Node parent, int height, ArrayList<ArrayList<Integer>> moves,
        int row, int col, int dr, boolean isKing) {
    Node next = new Node(coordinatesToNumber(row, col), height + 1, parent);
    int adjRow, adjCol, newRow, newCol;
    boolean nodeIsLeaf = true, checkOtherRow = !isKing;
    do {
        checkOtherRow = !checkOtherRow;
        for (int dc = -1; dc <= 1; dc += 2) {
            adjRow = row + dr;
            adjCol = col + dc;
            if (isInsideTheBoard(adjRow, adjCol)) {
                int owner = ownerOfField(adjRow, adjCol);
                if (owner != 0 && owner != ownerOfField(row, col)) {
                    newRow = adjRow + dr;
                    newCol = adjCol + dc;
                    if (isInsideTheBoard(newRow, newCol) && board[newRow][newCol] == 0) {
                        nodeIsLeaf = false;
                        board[newRow][newCol] = board[row][col];
                        board[row][col] = 0;
                        int capturedPiece = board[adjRow][adjCol];
                        board[adjRow][adjCol] = 0;
                        buildCaptureMove(next, next.height(), moves,
                            newRow, newCol, dr, isKing);
                        board[adjRow][adjCol] = capturedPiece;
                        board[row][col] = board[newRow][newCol];
                        board[newRow][newCol] = 0;
                    }
                }
            }
        }
        dr *= -1;
    } while (checkOtherRow);
    if (nodeIsLeaf) {
        ArrayList<Integer> move = getMoveFromTree(next);
        if (!move.isEmpty()) {
            moves.add(move);
        }
    }
}
```

Każdy obiekt klasy **State** może utworzyć listę stanów pochodnych. Do tego służy metoda *getChildren*, zwracająca listę obiektów **State** różniących się od rodzica pojedynczym ruchem.

3.2.2 Klasa Heuristic

Generuje obiekty różnych funkcji oceny heurystycznej. Zapożycza listę parametrów z klasy numerycznej **HParam** i przechowuje swój własny ciąg wag. Jej najważniejszymi metodami są *evaluate* obliczająca wartość oceny oraz *getParams* wstawiająca wartości pod parametry. Ogromną część klasy **Heuristic** stanowią metody wyciągające informacje o parametrach z obiektu klasy **State** z uwzględnieniem numeru gracza. Są to wrappery na metody z klasy stanu, dostosowane względem symetrii.

3.2.3 Klasa MinMax

Implementuje algorytm Minimax jako metodę *minimax*. Metoda jako parametry przyjmuje obiekt klasy **State**, obiekt klasy **Heuristic**, a także głębokość przeszukiwania, numer gracza uruchamiającego pierwsze wywołanie Minimaxa, flagę MIN/MAX oraz wartości α i β do wykonywania cięć. Wykorzystuje również statyczną klasę **RNG**, zawierającą logikę rachunków pseudolosowych.

Kod źródłowy 3.2: Implementacja algorytmu Minimax

```
public int minimax(State state, int depth, Heuristic heuristic, int alpha, int beta,
        int maximizingPlayer, boolean isPlayerMaximizing) {
    // Gdy osiągniemy maksymalną głębokość przeszukiwań,
    // zwracamy od razu wartość oceny danego stanu gry.
    if (depth == 0) return heuristic.evaluate(state, maximizingPlayer);

    State best = null;
    if (isPlayerMaximizing) {
        int maxEval = Integer.MIN_VALUE;
        double maxEps = 0.0;
        for (State child : state.getChildren()) {
            // Rekurencyjnie przeszukaj przestrzeń stanów które można osiągnąć z obecnego stanu.
            int eval = minimax(child, depth - 1, heuristic, alpha, beta, maximizingPlayer, false);
            if (maxEval < eval) {
                best = child;
                maxEval = eval;
            }
            else if (maxEval == eval) {
                /*
                Stanów o maksymalnej wartości funkcji oceny heurystycznej
                może być więcej niż jeden.
                Wówczas losujemy spośród tych stanów, aby strategia nie była
                zupełnie deterministyczna.
                Dla każdego stanu o maksymalnej wartości funkcji oceny heurystycznej
                losujemy epsilon i przyjmujemy ten największy,
                zatem losowanie stanu jest sprawiedliwe.
                 */
                double eps = RNG.randomDoubleFromZeroToOne();
                if (maxEps < eps) {
                    best = child;
                    maxEps = eps;
                }
            }
            if (alpha < eval) alpha = eval;
            // Nie musimy przeszukiwać reszty przesztrzeni stanów
            // jeśli możemy zastosować cięcie.
            if (beta <= alpha) break;
        }
        // Wykonaj najlepszy ruch na oryginalnej planszy.
        if (best != null) state.makeMove(best.creationMove());
        return maxEval;
    }
    else {
        int minEval = Integer.MAX_VALUE;
        double minEps = 1.0;
        for (State child : state.getChildren()) {
            int eval = minimax(child, depth - 1, heuristic, alpha, beta, maximizingPlayer, true);
            if (eval < minEval) {
                best = child;
                minEval = eval;
            }
            else if (eval == minEval) {
                double eps = RNG.randomDoubleFromZeroToOne();
                if (eps < minEps) {
                    best = child;
                    minEps = eps;
                }
            }
            if (eval < beta) beta = eval;
```

```
            if (beta <= alpha) break;
        }
        if (best != null) state.makeMove(best.creationMove());
        return minEval;
    }
}
```

3.2.4 GameHandler i rodzina klas Player

GameHandler odpowiedzialny jest za obsługę rozgrywek w warcaby angielskie. Mając obiekt klasy **State** oraz dwóch graczy **Player** ustala odpowiednią kolejność ruchów i kończy rozgrywkę w razie zwycięstwa. Aby uruchomić rozgrywkę, należy wywołać metodę *run*.

Player jest klasą abstrakcyjną z obowiązkową do zaimplementowania metodą *makeMove*. Dziedziczą po niej **PlayerComputer** oraz **PlayerHuman**. **PlayerComputer** odpowiada logice gracza komputerowego i implementuje *makeMove* jako wywołanie Minimaxa z posiadanej instancji **MinMax**, podając mu jako argument swój obiekt klasy **Heuristic**. **PlayerHuman** zawiera obsługę I/O potrzebną do komunikacji między graczem ludzkim a rozgrywką. Jego implementacja *makeMove* manipuluje listą dostępnych ruchów, a także łapie podstawowe błędy wejścia.

3.2.5 Klasa Genetic

W tej klasie znajduje się najważniejsza dla eksperymentów metoda *run*, która przeprowadza sesję algorytmu genetycznego. W swoich polach przechowuje argumenty dla takiej sesji, przez co jeden obiekt klasy **Genetic** jest w stanie przeprowadzić tylko jedną sesję (do odtworzenia sesji należy stworzyć nowy obiekt).

Genetic obsługuje selekcje i pojedynki metodami *selection* i *playDuels*, ustawiając ciągi wag w heurystykach graczy **PlayerComputer** i przeprowadzając rozgrywki obiektem **GameHandler**. Ostateczna selekcja w generacji działa na zasadzie ruletki, tj. osobniki mają tym wyższą szansę na przejście do populacji rodziców, im lepiej grały w pojedynkach. Do zsumowanego wyniku turniejowego każdego osobnika dodaje się losową wartość z przedziału od zera do wartości parametru ruletki, a następnie sortuje się osobniki względem nowych wyników.

Zarówno selekcja, jak i metody *crossover* oraz *mutation* korzystają ze statycznych funkcji losowania klasy **RNG**. Funkcje losowanie wykorzystuje się również w ogólnym przebiegu algorytmu genetycznego, by do świeżo powstających pokoleń wpuszczać też nowe, losowe osobniki.

Długość trwania sesji algorytmu genetycznego zależy od podanego kryterium stopu, obsługiwanego przez prywatną klasę **StopCondition**. Typy kryterium stopu definiowane są przez klasę numeryczną **StopCond**. W tej chwili dostępne są dwa typy - *TIME* (warunkiem stopu jest czas w sekundach) i *GENERATIONS* (warunkiem stopu jest liczba iteracji).

Kod źródłowy 3.3: Metoda selekcji w algorytmie genetycznym

```
private short[][] selection (short[][] population) {
    int popSize = population.length;
    // 0 - index; 1 - wygrane w ataku; 2 - wygrane w obronie; 3 - ogólny wynik.
    int[][] results = playDuels(population, popSize);
    // Znajdź najlepszy wynik
    for (int j = popSize - 1; j > 0; --j) {
        if (results[j-1][3] < results[j][3]) {
            int[] tmp = results[j];
            results[j] = results[j-1];
            results[j-1] = tmp;
        }
    }
    // Wybierz najlepszego.
    if (bestSoFar == null) bestSoFar = population[results[0][0]];
    else {  // Przeprowadź pojedynek gladiatorski
        short[] bestThisTime = population[results[0][0]];
        player1.changeHeuristicWeights(bestThisTime);
        player2.changeHeuristicWeights(bestSoFar);
```

```
            game1.resetBoard();
            game2.resetBoard();
            int result1 = game1.quickGame();
            int result2 = (-1) * game2.quickGame();
            if (result1 + result2 >= 0) bestSoFar = bestThisTime;
        }
        // Ruletka, czyli loteria osobników które przechodzą dalej.
        for (int i = 0; i < popSize; ++i) {
            results[i][3] += RNG.randomInt(selectionFactor);
        }
        // Insertion sort
        for (int i = 1; i < popSize; ++i) {
            for (int j = i; j > 0; --j) {
                if (results[j-1][3] < results[j][3]) {
                    int[] tmp = results[j];
                    results[j] = results[j-1];
                    results[j-1] = tmp;
                } else break;
            }
        }
        // W miarę możliwości dobieraj osobniki różne.
        boolean[] candidatesFree = new boolean[popSize];
        for (int i = 0; i < popSize; ++i) candidatesFree[i] = true;
        short[][] parents = new short[parentPopulationSize][genotypeSize];
        int numberOfParents = 0;
        for (int i = 0; i < popSize && numberOfParents < parentPopulationSize; ++i) {
            short[] candidate = population[results[i][0]];
            if (isGenotypeNotInPopulation(candidate, parents, 0, numberOfParents)) {
                parents[numberOfParents] = candidate;
                ++numberOfParents;
                candidatesFree[i] = false;
            }
        }
        // Dokończ populację rodziców duplikatami, aby nie było pustych miejsc.
        int index = 0;
        while (numberOfParents < parentPopulationSize) {
            short[] candidate = population[results[index][0]];
            if (candidatesFree[index]) {
                parents[numberOfParents] = candidate;
                ++numberOfParents;
                candidatesFree[index] = false;
            }
            ++index;
        }
        return parents;
    }
```

3.3 Instrukcja obsługi programów

Jak już wspominano we wcześniejszym podrozdziale 3.2, projekt obejmuje trzy dostępne dla użytkownika programy:

- **Play** - kierownik rozgrywek, prowadzi gry dla graczy zarówno ludzkich jak i komputerowych;
- **Find** - interfejs do uruchamiania sesji algorytmu genetycznego;
- **Show** - służy do przejrzystego drukowania ciągu wag z parametrami.

Programy uruchamia się przekazując argumenty z poziomu konsoli. Uruchomione programy przekazują informacje na standardowe wyjście (używając różnych kolorów, dostarczanych przez statyczną klasę **Console**). Sygnatury wywołań każdego z programów można wydrukować na standardowe wyjście, podając „–help” jako pierwszy argument.

3.3.1 Play

Dostępne sygnatury wywołania programu:

- Rozgrywka (standardowy model gry w warcaby, dostępny dla graczy ludzkich i/lub komputerowych; gracz komputerowy musi otrzymać ścieżkę do pliku z którego wczyta ciąg wag do swojej funkcji oceny heurystycznej)
  1. Typ pierwszego gracza [0 → gracz ludzki; liczba naturalna większa od zera → gracz komputerowy z głębokością przeszukiwań równą podanej liczbie]
  2. Ścieżka do pliku z ciągiem wag dla gracza pierwszego [ciąg znaków; argument omijany jeśli pierwszym graczem jest człowiek]
  3. Typ drugiego gracza [0 → gracz ludzki; liczba naturalna większa od zera → gracz komputerowy z głębokością przeszukiwań równą podanej liczbie]
  4. Ścieżka do pliku z ciągiem wag dla gracza drugiego [ciąg znaków; argument omijany jeśli drugim graczem jest człowiek]
- Szybka gra z komputerem o wylosowanej heurystyce (gra między graczem ludzkim a komputerowym; ciąg wag dla gracza komputerowego zostanie wylosowany)
  1. Gracz zaczynający [1 → zaczyna człowiek; -1 → zaczyna komputer]
  2. Głębokość przeszukiwań gracza komputerowego [liczba naturalna większa od zera]

Poprawnie wywołany program rozpocznie rozgrywkę w warcaby angielskie. Będzie na zmianę prosił graczy o wykonanie ruchu, lub, jeśli gracz jest komputerowy, obsłuży jego ruch. Po każdym ruchu drukowany będzie obecny stan planszy. Pokazywane są też ostatnio wykonane ruchy i lista możliwych ruchów dla gracza ludzkiego.

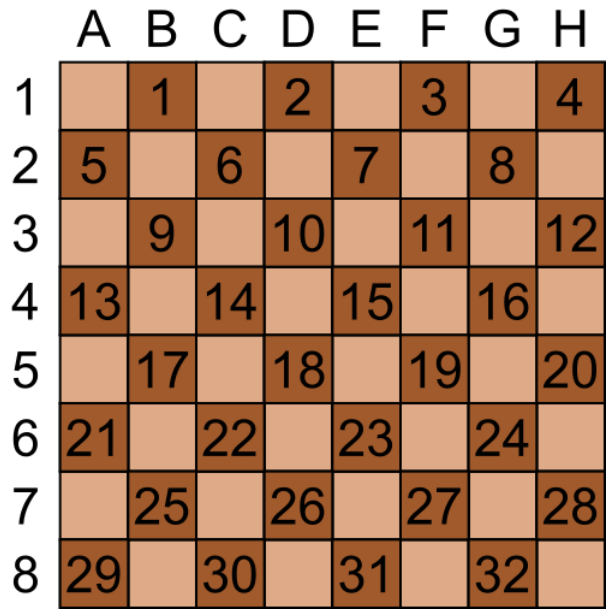
W trakcie wykonywania ruchu gracz ludzki musi podać pole z którego chce wykonać ruch, a następnie pole (pola) na które chce przeskoczyć wybraną figurą. Można się odnosić do pól na planszy na dwa sposoby: albo poprzez współrzędne (kolumna i wiersz), albo poprzez numerację pól (ukazaną na rys. 3.1).

W przypadku podania błędnego wejścia, program poprosi gracza o ponowne wprowadzenie ruchu od zera. W momencie gdy rozgrywka miałaby się skończyć, program informuje o zwycięzcy i kończy działanie.

3.3.2 Find

Dostępne sygnatury wywołania programu:

- Pierwsze uruchomienie algorytmu genetycznego (zalecane jeśli użytkownik chce utworzyć nową sesję algorytmu genetycznego od zera)
  1. Liczebność populacji osobników [liczba naturalna podzielna przez 4]
  2. Liczba pojedynków osobnika w procesie selekcji [liczba naturalna, przy czym 0 oznacza pojedynek z każdym innym osobnikiem]
  3. Głębokość przeszukiwania w Minimaksie [liczba naturalna większa od zera]
  4. Współczynnik losowej selekcji osobników [liczba całkowita]
  5. Szansa na mutację [liczba wymierna z przedziału od 0 do 1]
  6. Rodzaj kryterium stopu [0 → czas (w sekundach); 1 → liczba iteracji]
  7. Limit dla kryterium stopu [liczba naturalna]
- Reaktywacja algorytmu genetycznego z wybranego pliku populacji
  1. Nazwa pliku [ciąg znaków]
- Reaktywacja algorytmu genetycznego z ostatniego pliku populacji [BRAK ARGUMENTÓW]



Rysunek 3.1: Numeracja pól i współrzędne planszy

- Kontynuacja algorytmu genetycznego z nowymi parametrami
  1. Nazwa pliku z którego należy wczytać populację [ciąg znaków]
  2. Liczba pojedynków osobnika w procesie selekcji [liczba naturalna, przy czym 0 oznacza pojedynek z każdym innym osobnikiem]
  3. Głębokość przeszukiwania w Minimaksie [liczba naturalna większa od zera]
  4. Współczynnik losowej selekcji osobników [liczba całkowita]
  5. Szansa na mutację [liczba wymierna między 0 a 1]
  6. Rodzaj kryterium stopu [0 → czas (w sekundach); 1 → liczba iteracji]
  7. Limit dla kryterium stopu [liczba naturalna]

Bezbłędne uruchomienie rozpocznie sesję algorytmu genetycznego z podanymi argumentami. W trakcie działania program operuje na plikach w katalogu *heuristics* i jego podkatalogach: *population* (pliki zapisanych populacji, z których można reaktywować sesje; program na bieżąco aktualizuje plik populacji) oraz *output* (najlepiej przystosowany osobnik, tworzony w katalogu po osiągnięciu kryterium stopu). Oprócz tych podkatalogów istnieją również *single* (katalog na samodzielne tworzenie plików ciągów wag) i *archive* (archiwum w którym można bezpiecznie zapisywać pliki ciągów wag), lecz program na tych dwóch podkatalogach nie wykonuje żadnych operacji oprócz ich stworzenia.

Sesja algorytmu genetycznego na bieżąco informuje o postępie - w każdej iteracji drukuje numer porządkowy obecnie ewaluowanej generacji oraz stosunek postępu do limitu we wcześniej podanym kryterium stopu (np. 5/100 generacji lub 100/2000 sekund). W razie niespodziewanego wcześniejszego zakończenia programu, możliwym jest odczyt argumentów sesji i postępu z pliku ostatniej populacji.

3.3.3 Show

Dostępne sygnatury wywołania programu:

- Wydrukowanie ciągu wag z pliku osobnika
  1. Ścieżka do pliku osobnika

Program drukuje listę opisanych parametrów z przydzielonymi im wagami z podanego pliku. Jest to użyteczne narzędzie do wglądu w wyniki sesji algorytmu genetycznego.

Rozdział 4

Wyniki i rozszerzenia

Można wyróżnić 2 główne cele eksperymentów w pracy:

- **Sprawdzenie parametrów** - uruchomienie sesji algorytmu genetycznego w celu przypisania względnych wartości pod parametry w funkcji oceny heurystycznej;
- **Porównanie perspektyw MIN i MAX** - przeprowadzenie dwóch sesji algorytmu genetycznego z głębokościami różniącymi się od siebie o 1, a następnie porównanie wynikowych ciągów wag w celu ustalenia różnic między decyzjami gracza minimalizującego a decyzjami gracza maksymalizującego.

Do przeprowadzenia eksperymentów uruchomiono kilka sesji algorytmu genetycznego z różnymi argumentami. Poniżej znajdują się ich omówione wyniki.

4.1 Sprawdzenie parametrów

Przeprowadzono cztery różne sesje algorytmu genetycznego w celu znalezienia jak najlepszych priorytetów dla każdego parametru funkcji oceny heurystycznej. Ze względu na spory zakres wartości wag, losowość generowania wartości wag, jak i dużą liczbę parametrów, wprowadzono następujące oznaczenie priorytetów parametrów na podstawie wag:

- **Wysoki dodatni**→ duża wartość wagi; parametr bardzo korzystny lub kluczowy dla gracza
- **Średni dodatni**→ umiarkowana wartość wagi; parametr korzystny dla gracza
- **Niski**→ wartość wagi bliska zeru; algorytm nie zwraca uwagi na ten parametr
- **Średni ujemny**→ mała wartość wagi poniżej zera; parametr nieopłacalny dla gracza
- **Wysoki ujemny**→ bardzo niska wartość wagi; parametr staje się karą dla gracza

W tabeli 4.1 przedstawione są parametry funkcji oceny heurystycznej wraz z uśrednionymi wartościami wag przetłumaczonych na podane wyżej priorytety.

Wynikowe priorytety części z parametrów mogą wydawać się nieintuicyjne (np. wartościowanie sojuszniczych pionów niżej niż przeciwnych pionów). Należy jednak pamiętać, że są one rezultatem przeprowadzenia ogromnej liczby turniejów z wykorzystaniem przeróżnych strategii. Ponadto, różne sesje algorytmu genetycznego potrafią znajdować różne rozwiązania, oraz, jak wspomniano w opisie algorytmu genetycznego 2.3, nie można wykluczyć istnienia strategii rozmijających się z ludzkim pojmowaniem dobrych i złych decyzji w rozgrywce.

Należy zauważyć, że parametry liczby pionów powielają się w wielu innych parametrach, jak np. liczba pionów przy ściankach, liczba pionów w środkowych rzędach, liczba samotnych pionów. Mogło to sprawić,

Nr	Parametr	Priorytet
1	Liczba sojuszniczych pionów	**Wysoki ujemny**
2	Liczba sojuszniczych damek	**Średni dodatni**
3	Liczba przeciwnych pionów	**Wysoki dodatni**
4	Liczba przeciwnych damek	**Wysoki ujemny**
5	Liczba sojuszniczych pionów przy ścianie	**Wysoki dodatni**
6	Liczba sojuszniczych damek przy ścianie	**Średni ujemny**
7	Liczba przeciwnych pionów przy ścianie	**Średni ujemny**
8	Liczba przeciwnych damek przy ścianie	**Wysoki dodatni**
9	Liczba ruchomych pionów gracza	**Średni dodatni**
10	Liczba ruchomych damek gracza	**Wysoki dodatni**
11	Liczba ruchomych pionów przeciwnika	**Wysoki dodatni**
12	Liczba ruchomych damek przeciwnika	**Średni ujemny**
13	Liczba możliwych ruchów gracza	**Wysoki dodatni**
14	Liczba możliwych ruchów przeciwnika	**Średni dodatni**
15	Istnienie bijącego ruchu gracza	**Średni dodatni**
16	Liczba bijących ruchów gracza	**Wysoki ujemny**
17	Rozmiar najdłuższego bijącego ruchu gracza	**Wysoki ujemny**
18	Istnienie bijącego ruchu przeciwnika	**Wysoki ujemny**
19	Liczba bijących ruchów przeciwnika	**Średni dodatni**
20	Rozmiar najdłuższego bijącego ruchu przeciwnika	**Wysoki ujemny**
21	Suma dystansów pionów gracza do rzędu awansu	**Średni dodatni**
22	Suma dystansów pionów przeciwnika do rzędu awansu	**Wysoki ujemny**
23	Liczba niezajętych pól w rzędzie awansu gracza	**Niski**
24	Liczba niezajętych pól w rzędzie awansu przeciwnika	**Wysoki ujemny**
25	Liczba sojuszniczych pionów w dolnych rzędach	**Średni ujemny**
26	Liczba sojuszniczych damek w dolnych rzędach	**Średni dodatni**
27	Liczba przeciwnych pionów w dolnych rzędach	**Średni dodatni**
28	Liczba przeciwnych damek w dolnych rzędach	**Średni dodatni**
29	Liczba sojuszniczych pionów w środkowych rzędach	**Wysoki ujemny**
30	Liczba sojuszniczych damek w środkowych rzędach	**Wysoki dodatni**
31	Liczba przeciwnych pionów w środkowych rzędach	**Wysoki ujemny**
32	Liczba przeciwnych damek w środkowych wierszach	**Średni ujemny**
33	Liczba sojuszniczych pionów w górnych rzędach	**Średni dodatni**
34	Liczba sojuszniczych damek w górnych rzędach	**Wysoki dodatni**
35	Liczba przeciwnych pionów w górnych rzędach	**Średni ujemny**
36	Liczba przeciwnych damek w górnych rzędach	**Średni dodatni**
37	Liczba samotnych sojuszniczych pionów	**Niski**
38	Liczba samotnych sojuszniczych damek	**Wysoki ujemny**
39	Liczba samotnych przeciwnych pionów	**Średni ujemny**
40	Liczba samotnych przeciwnych damek	**Wysoki dodatni**
41	Czy pion gracza jest w kącie	**Wysoki ujemny**
42	Czy damka gracza jest w kącie	**Wysoki ujemny**
43	Czy gracz zajmuje dwa kąty	**Średni ujemny**
44	Czy pion przeciwnika jest w kącie	**Niski**
45	Czy damka przeciwnika jest w kącie	**Średni ujemny**
46	Czy przeciwnik zajmuje dwa kąty	**Wysoki dodatni**
47	Obecność *Triangle pattern* u gracza	**Wysoki ujemny**
48	Obecność *Oreo pattern* u gracza	**Wysoki dodatni**
49	Obecność *Bridge pattern* u gracza	**Średni ujemny**
50	Obecność *Dog pattern* u gracza	**Średni dodatni**
51	Obecność *Triangle pattern* u przeciwnika	**Średni dodatni**
52	Obecność *Oreo pattern* u przeciwnika	**Wysoki ujemny**
53	Obecność *Bridge pattern* u przeciwnika	**Średni ujemny**
54	Obecność *Dog pattern* u przeciwnika	**Wysoki dodatni**
55	Liczba blokujących sojuszniczych figur	**Niski**
56	Liczba linii bloku gracza	**Średni dodatni**
57	Wielkość najdłuższej linii bloku gracza	**Średni dodatni**
58	Liczba blokujących przeciwnych figur	**Wysoki dodatni**
59	Liczba linii bloku przeciwnika	**Wysoki ujemny**
60	Wielkość najdłuższej linii bloku przeciwnika	**Średni ujemny**

Tabela 4.1: Uśrednione wyniki kilku sesji algorytmu genetycznego.

Nr	Parametr	Waga
1	Liczba sojuszniczych pionów	-27848
3	Liczba przeciwnych pionów	22520
4	Liczba przeciwnych damek	-24912
5	Liczba sojuszniczych pionów przy ścianie	25644
8	Liczba przeciwnych damek przy ścianie	27337
10	Liczba ruchomych damek gracza	4708
11	Liczba ruchomych pionów przeciwnika	6119
13	Liczba możliwych ruchów gracza	21613
16	Liczba bijących ruchów gracza	-28692
17	Rozmiar najdłuższego bijącego ruchu gracza	-20989
18	Istnienie bijącego ruchu przeciwnika	-20551
20	Rozmiar najdłuższego bijącego ruchu przeciwnika	-13876
22	Suma dystansów pionów przeciwnika do rzędu awansu	-24871
24	Liczba niezajętych pól w rzędzie awansu przeciwnika	-30665
29	Liczba sojuszniczych pionów w środkowych rzędach	-24696
30	Liczba sojuszniczych damek w środkowych rzędach	26462
31	Liczba przeciwnych pionów w środkowych rzędach	-32375
34	Liczba sojuszniczych damek w górnych rzędach	27159
38	Liczba samotnych sojuszniczych damek	-20871
40	Liczba samotnych przeciwnych damek	11132
41	Czy pion gracza jest w kącie	-25605
42	Czy damka gracza jest w kącie	-32651
46	Czy przeciwnik zajmuje dwa kąty	25836
47	Obecność *Triangle pattern* u gracza	-25839
48	Obecność *Oreo pattern* u gracza	1326
52	Obecność *Oreo pattern* u przeciwnika	-618
54	Obecność *Dog pattern* u przeciwnika	14026
58	Liczba blokujących przeciwnych figur	32446
59	Liczba linii bloku przeciwnika	-28419

Tabela 4.2: Zestaw priorytetowych parametrów z wartościami wag

że licząc piony w wielu miejscach, algorytm musiał przypisać ujemną wagę ogólnej liczbie pionów, aby samo istnienie pionów nie przeważało w ostatecznej ocenie stanu. Oprócz tego, na późniejszych etapach gry piony mogą stać się dla gracza niekorzystne, choćby z tego powodu że lepiej jest mieć więcej damek. Warto sobie uświadomić, że obecna implementacja programu nie odróżnia etapu początkowego od etapu końcowego rozgrywki, przez co powstałą funkcję oceny heurystycznej można traktować jak uśrednienie taktyk opłacalnych na różnych stadiach gry.

Wynikowe priorytety parametrów 23, 37, 44 oraz 55 są niskiego rzędu, co sugeruje że nie mają one wpływu na jakość rozgrywki. Można je więc pominąć w ewaluacji stanów. Parametry o średnich priorytetach wywierają większą presję na ocenie heurystycznej i do pewnego stopnia warto zwracać na nie uwagę, jednak w ostatecznym rozrachunku może się okazać, że warto kierować się tylko parametrami o najwyższym priorytecie.

Na podstawie tych wyników i wniosków możemy zoptymalizować funkcję oceny heurystycznej, usuwając parametry których wpływ jest mniejszy w porównaniu do innych parametrów. Tabela 4.2 jest wynikiem wycięcia parametrów o niskim lub średnim priorytecie. Wprowadzono również wartości wag wyliczone w jednej z sesji eksperymentu.

Jak już wcześniej wspomniano, niska waga liczby sojuszniczych pionów może być wynikiem przewrażliwienia funkcji oceny ze względu na wysoką liczebność parametrów rozpatrujących piony, chociażby parametr 5 (obejmuje on piony przy ścianach, czyli takie których nie da się zbić). Wysoka waga liczby przeciwnych pionów (parametry 3, 11) może dodatkowo wskazywać, że na pewnym etapie gry w warcaby posiadanie pionów jest postrzegane jako słaby punkt na planszy. W tym przekonaniu utwierdza ujemna wartość wagi dla przeciwnych damek - dla gracza lepiej aby przeciwnik transformował jak najmniej pionów w damki.

Z obserwacji rezultatów wynika, że algorytm lubi mieć swobodę w wykonywaniu ruchów, lecz nie przepada za sytuacjami w których ma wymuszone bicie, najprawdopodobniej dlatego że jest poza jego

kontrolą. Nie jest to jednak własność symetryczna, ponieważ nie przepada też za tym, gdy jego oponent ma w danej chwili ruch bijący. Potencjalną hipotezą tłumaczącą to zjawisko jest różnica między traceniem figur a ich transformacją - algorytm woli zmniejszyć liczbę swoich pionów awansując je do damek, zamiast dać im się zbić przeciwnikowi.

Priorytet parametru 22 może wskazywać na to, że algorytm woli gdy piony przeciwnika znajdują się bliżej „bazy" gracza, być może dlatego, że gracz może je wówczas łatwiej zbić. Możliwym jest też, że parametr sumy dystansów pionów przeciwnika do rzędu awansu w pewien sposób preferuje stany w których przeciwnik posiada ogólnie mało pionów (wówczas wartość tego parametru też jest mała). Argumentem za tą hipotezą może być waga parametru 24, która silnie przeciwstawia się sposobności oponenta do awansu jego pionów.

Dalej można wnioskować że, według wyznaczonych wag, graczowi nie opłaca się zostawiać pionów na środku planszy, ale za to bardzo mu się opłaca przeprowadzać je wzdłuż ścian planszy i awansować je by umiejscowić damki w „bazie" przeciwnika. Równocześnie damki okazują się być figurami niewykorzystywanymi w pełni, jeżeli nie mają innych figur w pobliżu. Może to świadczyć o tym, że damka przynosi najlepsze rezultaty, gdy wywiera ciągłą presję na przeciwniku, lub że damki należy chronić innymi sojuszniczymi figurami, by nie zostały zbite.

Jeżeli chodzi o parametry binarne, można wnioskować że algorytm uważa figury siedzące w kątach za bezużyteczne i przynoszące niekorzyść ich właścicielowi, podobnie dla pionów wchodzących w skład *Triangle pattern*. Pozytywna waga *Oreo pattern* może być powiązana ze wcześniej przeanalizowanym parametrem 24. Warto zauważyć, że jeśli istnieje u przeciwnika *Dog pattern*, gracz posiada niemożliwego do zbicia piona, który albo blokuje jednego piona przeciwnika, albo ma prostą drogę do awansu. Możliwym jest, że to właśnie stąd wygenerowana została pozytywna waga takiej sytuacji.

Ostatnie dwa parametry w tabeli 4.2 można wytłumaczyć następującą hipotezą. Algorytm uważa, że blokujące figury nie są użyteczne ze względu na brak mobilności, być może dlatego że zwraca większą uwagę na ofensywną strategię awansowania pionów. Defensywne linie bloku oponenta przeszkadzają mu w tym jednak, dlatego też woli unikać takich stanów.

Powstałą strategię algorytmu można podsumować jako agresywną taktykę dążącą do jak najszybszego utworzenia damek, przy jednoczesnym uniemożliwieniu awansu przeciwnikowi. Algorytm widzi olbrzymi potencjał w damce, która notabene jest najpotężniejszą figurą w grze. Dzięki niej gracz może powybijać piony przeciwnika od wewnątrz.

4.2 Porównanie perspektyw MIN i MAX

W ramach eksperymentu przeprowadzono cztery różne sesje algorytmu genetycznego, z czego w dwóch sesjach głębokość przeszukiwań wynosiła 4, a w dwóch pozostałych głębokość wynosiła 5. Celem eksperymentu było sprawdzenie, czy funkcja oceny heurystycznej zależy od tego, kto podejmuje decyzje przy liściach drzewa przeszukiwań w Minimaksie - gracz MAX czy gracz MIN. W tabelach 4.3 oraz 4.4 znajdują się wyniki eksperymentu.

Można zaobserwować, że mimo wielu różnic, w obu przypadkach obie głębokości poskutkowały tymi samymi priorytetami dla około jednej trzeciej parametrów. Obie perspektywy (MIN i MAX) przykładają mniejszą uwagę do parametrów związanych z *patternami*. Można przypuszczać, że w takim razie istnieją uniwersalne i niezależne od perspektywy parametry, których priorytety są niezmienne.

Liczność par parametrów o różniących się priorytetach skłania jednak do wniosku, że gracz MIN inaczej rozpatruje stany gry od gracza MAX. Jeśli założymy że algorytm genetyczny dąży zawsze do tego samego rozwiązania z tych samych początkowych danych (co wcale nie musi być prawdą), to rezultaty eksperymentu mogą być empirycznym dowodem na to, że sam fakt wyboru innego ekstremalnego stanu z poddrzewa stanów na maksymalnej głębokości przeszukiwań prowadzi do konieczności obrania trochę innej strategii. Z ogólnej obserwacji wyników można zaryzykować stwierdzenie, że taktyka wygenerowana w sesji w której MAX decydował jako pierwszy jest bardziej ofensywna, natomiast w sesji w której to MIN decydował pierwszy - bardziej defensywna (wyższe wartościowanie liczby pionów oraz figur blokujących).

Nr	Parametr	Priorytet h = 4	Priorytet h = 5
1	Liczba sojuszniczych pionów	Wysoki dodatni	Wysoki dodatni
2	Liczba sojuszniczych damek	Wysoki dodatni	Wysoki dodatni
3	Liczba przeciwnych pionów	Średni ujemny	Wysoki ujemny
4	Liczba przeciwnych damek	Wysoki ujemny	Wysoki ujemny
5	Liczba sojuszniczych pionów przy ścianie	Niski	Wysoki dodatni
6	Liczba sojuszniczych damek przy ścianie	Średni dodatni	Wysoki dodatni
7	Liczba przeciwnych pionów przy ścianie	Średni ujemny	Niski
8	Liczba przeciwnych damek przy ścianie	Wysoki ujemny	Wysoki ujemny
9	Liczba ruchomych pionów gracza	Niski	Średni dodatni
10	Liczba ruchomych damek gracza	Wysoki dodatni	Wysoki dodatni
11	Liczba ruchomych pionów przeciwnika	Średni ujemny	Średni ujemny
12	Liczba ruchomych damek przeciwnika	Wysoki ujemny	Wysoki ujemny
13	Liczba możliwych ruchów gracza	Średni dodatni	Średni ujemny
14	Liczba możliwych ruchów przeciwnika	Średni dodatni	Średni ujemny
15	Istnienie bijącego ruchu gracza	Wysoki dodatni	Średni dodatni
16	Liczba bijących ruchów gracza	Wysoki ujemny	Wysoki dodatni
17	Rozmiar najdłuższego bijącego ruchu gracza	Wysoki dodatni	Średni dodatni
18	Istnienie bijącego ruchu przeciwnika	Średni ujemny	Wysoki ujemny
19	Liczba bijących ruchów przeciwnika	Średni ujemny	Niski
20	Rozmiar najdłuższego bijącego ruchu przeciwnika	Średni dodatni	Wysoki ujemny
21	Suma dystansów pionów gracza do rzędu awansu	Wysoki dodatni	Wysoki dodatni
22	Suma dystansów pionów przeciwnika do rzędu awansu	Wysoki ujemny	Wysoki ujemny
23	Liczba niezajętych pól w rzędzie awansu gracza	Średni dodatni	Średni ujemny
24	Liczba niezajętych pól w rzędzie awansu przeciwnika	Wysoki ujemny	Wysoki ujemny
25	Liczba sojuszniczych pionów w dolnych rzędach	Średni ujemny	Wysoki ujemny
26	Liczba sojuszniczych damek w dolnych rzędach	Wysoki ujemny	Wysoki dodatni
27	Liczba przeciwnych pionów w dolnych rzędach	Wysoki dodatni	Niski
28	Liczba przeciwnych damek w dolnych rzędach	Niski	Średni dodatni
29	Liczba sojuszniczych pionów w środkowych rzędach	Średni dodatni	Średni dodatni
30	Liczba sojuszniczych damek w środkowych rzędach	Wysoki dodatni	Wysoki dodatni
31	Liczba przeciwnych pionów w środkowych rzędach	Wysoki ujemny	Wysoki ujemny
32	Liczba przeciwnych damek w środkowych wierszach	Średni dodatni	Średni dodatni
33	Liczba sojuszniczych pionów w górnych rzędach	Wysoki dodatni	Wysoki dodatni
34	Liczba sojuszniczych damek w górnych rzędach	Wysoki dodatni	Wysoki dodatni
35	Liczba przeciwnych pionów w górnych rzędach	Wysoki ujemny	Wysoki ujemny
36	Liczba przeciwnych damek w górnych rzędach	Wysoki ujemny	Niski
37	Liczba samotnych sojuszniczych pionów	Wysoki dodatni	Wysoki dodatni
38	Liczba samotnych sojuszniczych damek	Średni dodatni	Średni dodatni
39	Liczba samotnych przeciwnych pionów	Wysoki ujemny	Średni ujemny
40	Liczba samotnych przeciwnych damek	Wysoki ujemny	Wysoki ujemny
41	Czy pion gracza jest w kącie	Wysoki ujemny	Wysoki ujemny
42	Czy damka gracza jest w kącie	Średni ujemny	Wysoki ujemny
43	Czy gracz zajmuje dwa kąty	Średni dodatni	Wysoki ujemny
44	Czy pion przeciwnika jest w kącie	Wysoki dodatni	Wysoki dodatni
45	Czy damka przeciwnika jest w kącie	Średni dodatni	Średni dodatni
46	Czy przeciwnik zajmuje dwa kąty	Średni dodatni	Średni dodatni
47	Obecność *Triangle pattern* u gracza	Wysoki dodatni	Wysoki dodatni
48	Obecność *Oreo pattern* u gracza	Wysoki ujemny	Średni dodatni
49	Obecność *Bridge pattern* u gracza	Średni dodatni	Wysoki dodatni
50	Obecność *Dog pattern* u gracza	Niski	Wysoki dodatni
51	Obecność *Triangle pattern* u przeciwnika	Niski	Wysoki ujemny
52	Obecność *Oreo pattern* u przeciwnika	Średni ujemny	Niski
53	Obecność *Bridge pattern* u przeciwnika	Niski	Niski
54	Obecność *Dog pattern* u przeciwnika	Wysoki dodatni	Wysoki dodatni
55	Liczba blokujących sojuszniczych figur	Wysoki dodatni	Wysoki dodatni
56	Liczba linii bloku gracza	Średni dodatni	Wysoki dodatni
57	Wielkość najdłuższej linii bloku gracza	Niski	Średni dodatni
58	Liczba blokujących przeciwnych figur	Wysoki ujemny	Średni ujemny
59	Liczba linii bloku przeciwnika	Niski	Niski
60	Wielkość najdłuższej linii bloku przeciwnika	Średni dodatni	Średni dodatni

Tabela 4.3: Porównanie priorytetów dla głębokości 4 oraz głębokości 5, sesja 1.

Nr	Parametr	Priorytet h = 4	Priorytet h = 5
1	Liczba sojuszniczych pionów	**Wysoki dodatni**	**Średni ujemny**
2	Liczba sojuszniczych damek	**Wysoki ujemny**	**Średni ujemny**
3	Liczba przeciwnych pionów	**Średni ujemny**	**Wysoki ujemny**
4	Liczba przeciwnych damek	**Niski**	**Wysoki dodatni**
5	Liczba sojuszniczych pionów przy ścianie	**Wysoki dodatni**	**Średni dodatni**
6	Liczba sojuszniczych damek przy ścianie	**Wysoki dodatni**	**Średni ujemny**
7	Liczba przeciwnych pionów przy ścianie	**Wysoki ujemny**	**Średni dodatni**
8	Liczba przeciwnych damek przy ścianie	**Wysoki ujemny**	**Średni dodatni**
9	Liczba ruchomych pionów gracza	**Wysoki dodatni**	**Wysoki dodatni**
10	Liczba ruchomych damek gracza	**Średni ujemny**	**Średni ujemny**
11	Liczba ruchomych pionów przeciwnika	**Średni ujemny**	**Wysoki ujemny**
12	Liczba ruchomych damek przeciwnika	**Średni dodatni**	**Wysoki ujemny**
13	Liczba możliwych ruchów gracza	**Średni dodatni**	**Średni dodatni**
14	Liczba możliwych ruchów przeciwnika	**Wysoki dodatni**	**Wysoki dodatni**
15	Istnienie bijącego ruchu gracza	**Wysoki ujemny**	**Średni ujemny**
16	Liczba bijących ruchów gracza	**Wysoki ujemny**	**Wysoki ujemny**
17	Rozmiar najdłuższego bijącego ruchu gracza	**Wysoki ujemny**	**Średni dodatni**
18	Istnienie bijącego ruchu przeciwnika	**Średni dodatni**	**Wysoki ujemny**
19	Liczba bijących ruchów przeciwnika	**Wysoki ujemny**	**Wysoki dodatni**
20	Rozmiar najdłuższego bijącego ruchu przeciwnika	**Wysoki ujemny**	**Wysoki ujemny**
21	Suma dystansów pionów gracza do rzędu awansu	**Wysoki dodatni**	**Średni dodatni**
22	Suma dystansów pionów przeciwnika do rzędu awansu	**Wysoki ujemny**	**Wysoki ujemny**
23	Liczba niezajętych pól w rzędzie awansu gracza	**Niski**	**Średni ujemny**
24	Liczba niezajętych pól w rzędzie awansu przeciwnika	**Wysoki dodatni**	**Wysoki dodatni**
25	Liczba sojuszniczych pionów w dolnych rzędach	**Wysoki dodatni**	**Niski**
26	Liczba sojuszniczych damek w dolnych rzędach	**Średni dodatni**	**Średni dodatni**
27	Liczba przeciwnych pionów w dolnych rzędach	**Niski**	**Niski**
28	Liczba przeciwnych damek w dolnych rzędach	**Średni ujemny**	**Średni ujemny**
29	Liczba sojuszniczych pionów w środkowych rzędach	**Wysoki dodatni**	**Wysoki ujemny**
30	Liczba sojuszniczych damek w środkowych rzędach	**Średni dodatni**	**Średni dodatni**
31	Liczba przeciwnych pionów w środkowych rzędach	**Niski**	**Średni ujemny**
32	Liczba przeciwnych damek w środkowych wierszach	**Średni ujemny**	**Średni dodatni**
33	Liczba sojuszniczych pionów w górnych rzędach	**Średni dodatni**	**Średni dodatni**
34	Liczba sojuszniczych damek w górnych rzędach	**Wysoki ujemny**	**Wysoki ujemny**
35	Liczba przeciwnych pionów w górnych rzędach	**Średni dodatni**	**Wysoki dodatni**
36	Liczba przeciwnych damek w górnych rzędach	**Wysoki dodatni**	**Niski**
37	Liczba samotnych sojuszniczych pionów	**Niski**	**Wysoki ujemny**
38	Liczba samotnych sojuszniczych damek	**Wysoki dodatni**	**Wysoki ujemny**
39	Liczba samotnych przeciwnych pionów	**Średni dodatni**	**Średni ujemny**
40	Liczba samotnych przeciwnych damek	**Wysoki dodatni**	**Średni dodatni**
41	Czy pion gracza jest w kącie	**Wysoki dodatni**	**Średni dodatni**
42	Czy damka gracza jest w kącie	**Średni ujemny**	**Wysoki ujemny**
43	Czy gracz zajmuje dwa kąty	**Średni ujemny**	**Niski**
44	Czy pion przeciwnika jest w kącie	**Średni ujemny**	**Średni dodatni**
45	Czy damka przeciwnika jest w kącie	**Wysoki ujemny**	**Średni ujemny**
46	Czy przeciwnik zajmuje dwa kąty	**Wysoki dodatni**	**Średni ujemny**
47	Obecność *Triangle pattern* u gracza	**Wysoki dodatni**	**Średni dodatni**
48	Obecność *Oreo pattern* u gracza	**Średni ujemny**	**Wysoki dodatni**
49	Obecność *Bridge pattern* u gracza	**Niski**	**Średni ujemny**
50	Obecność *Dog pattern* u gracza	**Średni dodatni**	**Średni ujemny**
51	Obecność *Triangle pattern* u przeciwnika	**Niski**	**Wysoki ujemny**
52	Obecność *Oreo pattern* u przeciwnika	**Średni dodatni**	**Średni dodatni**
53	Obecność *Bridge pattern* u przeciwnika	**Średni dodatni**	**Wysoki ujemny**
54	Obecność *Dog pattern* u przeciwnika	**Wysoki dodatni**	**Wysoki ujemny**
55	Liczba blokujących sojuszniczych figur	**Wysoki dodatni**	**Niski**
56	Liczba linii bloku gracza	**Niski**	**Niski**
57	Wielkość najdłuższej linii bloku gracza	**Niski**	**Wysoki dodatni**
58	Liczba blokujących przeciwnych figur	**Wysoki dodatni**	**Wysoki dodatni**
59	Liczba linii bloku przeciwnika	**Niski**	**Średni dodatni**
60	Wielkość najdłuższej linii bloku przeciwnika	**Niski**	**Średni ujemny**

Tabela 4.4: Porównanie priorytetów dla głębokości 4 oraz głębokości 5, sesja 2.

4.3 Możliwości rozwoju projektu

Mimo osiągnięcia zamierzonych celów wciąż pozostaje kilka aspektów pracy, które da się rozwinąć lub ulepszyć. Otwarcie kodów źródłowych na rozszerzenia może ułatwić dodanie nowych funkcjonalności bądź modyfikację kopii niektórych klas. Poniżej znajduje się kilka propozycji rozwinięcia projektu dalej.

4.3.1 Optymalizacje

Przeszukiwanie przestrzeni stanów w Minimaksie można usprawnić o bazę rozpoczęć i zakończeń - algorytm mógłby zaczynać gry lub odpowiadać jednym ze standardowych rozpoczęć turniejowych i, gdy nadarzy się sposobność, dążyć do jednego z zakończeń. Na podobnej zasadzie działał np. Deep Blue, komputer firmy IBM, który jako pierwsza maszyna na świecie zwyciężył w partii szachów z ówczesnym mistrzem świata Garri Kasparowem w 1996 roku [3].

Innym pomysłem na potencjalne skrócenie obliczeń jest zastosowanie Hashmapy przeszukanych stanów z przypisanymi im ocenami. Pomysł ten bazuje na obserwacji, że do niektórych stanów na planszy można dojść z kilku innych stanów. Rozpatrując stan, algorytm sięgałby do takiej Hashmapy i jeżeli znalazłby hash tego stanu, automatycznie przyznawałby ocenę bez konieczności schodzenia głębiej w drzewie. Warto jednak zaznaczyć, że operacje dodania i przeszukania w Hashmapie nie są stałe (zajmują złożoność logarytmiczną zależną od liczby stanów) i wykonywane są dla każdego stanu, a samych stanów może być w pewnym momencie bardzo dużo.

Jeszcze innym aspektem, którego optymalizacja mogłaby znacznie poprawić wydajność, szczególnie dla sesji algorytmu genetycznego, są funkcje analizujące planszę i wyliczające parametry do oceny heurystycznej. W momencie pisania pracy istnieje wiele funkcji wyliczających wartość jednego parametru w czterech wariantach (sojusznicze piony, sojusznicze damki, przeciwne piony, przeciwne damki). Warto jednak zwrócić uwagę na fakt, że prawie każda z tych funkcji musi przeanalizować całą planszę po każdym polu. Stąd pomysł na optymalizację: utworzyć specjalną klasę *StateAnalyzer* przechowującą informacje o wszystkich parametrach oraz booleanową flagę wskazującą, czy informacje te są aktualne. Przy każdym ruchu flagę ustawia się na *false*, a w razie konieczności (gdy zewnętrzny obiekt prosi o wartość parametru) ustawia się flagę na *true* i oblicza wartości każdego parametru od razu. Innym sposobem na optymalizację w tym zakresie jest obliczanie na nowo wartości tylko tych parametrów, które rzeczywiście zmieniają się w danym ruchu.

Dodatkowo można spróbować ulepszyć wydajność samego algorytmu genetycznego. Łatwo zauważyć, że do wybrania lepiej przystosowanej połowy populacji osobników (z czynnikiem losowym) nie potrzeba sortowania gorzej przystosowanej połowy populacji, ponieważ ta zostaje odrzucona. Sortowanie tylko części populacji miałoby szansę wywrzeć zauważalny wpływ na czas wykonania obliczeń. Można też eksperymentować z wprowadzaniem innych algorytmów sortujących, chociażby *QuickSort*.

4.3.2 Walka z efektem horyzontu

Efektem horyzontu nazywamy problem, w którym ograniczenie na wielkość przeszukiwanego kawałka przestrzeni stanów uniemożliwia dojście do potencjalnie lepszego rozwiązania znajdującego się krok dalej. Obecna implementacja Minimaxa w pracy jest podatna na problem horyzontu. Można częściowo temu zapobiec, zmuszając algorytm do rozpatrzenia dzieci stanu na maksymalnej głębokości, jeżeli jest on jedynym stanem pochodnym swojego rodzica (możliwość wykonania tylko jednego ruchu najczęściej oznacza wymuszone bicie, które jest poza kontrolą gracza). Można oszacować koszt takiego sprawdzenia jako większy zaledwie o jeden od średniego kosztu przejścia drzewa przeszukiwań - to tak jakby przenieść jedno poddrzewo o rząd niżej i umieścić w jego miejsce jeden stan. Ideę tę rozwija „przeszukiwanie uspokajające” (*Quiescence Search* [2]), które zatrzymuje przeszukiwanie poddrzew tylko na stanach spokojnych, czyli na takich które w najbliższych paru krokach nie zmienią się drastycznie (np. w warcabach stany tuż przed biciem lub awansem piona do damki nie są spokojne).

4.3.3 Analiza MINa i MAXa

Jak wspomniano we wnioskach podrozdziału 4.2, rezultaty badań dwóch perspektyw w Minimaksie sugerują, że różnica między ocenami których bezpośrednie wartości dla stanów są minimalizowane, a ocenami których wartości są maksymalizowane, jest niemała. Być może istnieją pewne niuanse związane ze spojrzeniem graczy na maksymalnej głębokości przeszukiwania. W ramach rozwinięcia pracy można by było przyjrzeć się temu zjawisku, przebadać je i zapisać nowe wnioski.

4.3.4 Interfejs

W obecnej chwili program prowadzący rozgrywki w warcaby uruchamiany jest z poziomu konsoli. Miłym rozszerzeniem byłoby stworzenie przyjaznego ludzkiemu użytkownikowi interfejsu do uruchamiania rozgrywek oraz intuicyjnego wprowadzania ruchów, chociażby poprzez kliknięcia. Nie zaszkodzą również dodatkowe opcje, np. możliwość cofnięcia ruchu, zapis i odczyt rozgrywki, przechowywanie interaktywnej historii rozgrywek. To wszystko można opakować w stronę webową i ją udostępnić.

Ciekawym pomysłem jest również napisanie adaptera, który pozwalałby sztucznej inteligencji stworzonej w pracy prowadzić rozgrywki z innymi dostępnymi graczami komputerowymi w internecie. Pozwoliłoby to na poznanie siły wyznaczonej sztucznej inteligencji.

Podsumowanie

W pracy przeprowadzono kilka sesji algorytmu genetycznego w ramach eksperymentów na algorytmie Minimax z funkcją oceny heurystycznej. Udało się poznać częściowe odpowiedzi na zadane pytania i sformułować z nich hipotezy.

Najważniejszym rezultatem testów jest uśredniony ciąg wag dla parametrów funkcji oceny. Dzięki niemu udało się określić parametry o większym wpływie na rozgrywkę w warcabach i odrzucić parametry niższego priorytetu. Podsumowano powstałą strategię jako agresywną taktykę dążącą do jak najszybszego awansu piona do damki. W przyszłości można zapuścić sesję algorytmu genetycznego ze skróconą listą parametrów, aby uzyskać jeszcze dokładniejsze wagi.

Kolejnym ważnym wnioskiem jest różnica w perspektywach graczy MIN i MAX. Okazuje się, że dobra strategia w warcabach (i prawdopodobnie innych grach dwuosobowych o sumie zerowej), opierająca się o Minimaxa, powinna zwracać uwagę na to który gracz dokonuje wyboru spośród liści drzewa przeszukiwań. Wstępne eksperymenty pokazały, że taktyka pod MAXa jest na ogół bardziej ofensywna, natomiast taktyka pod MINa - bardziej defensywna.

Jednak mimo osiągnięcia zamierzonych celów, praca ta jest zaledwie początkiem bardziej rozbudowanych badań i projektów. Z racji iż kody źródłowe otwarte są na rozszerzenia oraz umieszczone są w repozytorium **GitHub**, dalsze wsparcie projektu jest jak najbardziej możliwe.

Przyjrzenie się bliżej perspektywom MINa i MAXa to naturalne rozszerzenie pracy. Może ono stanowić ciekawy wgląd w algorytmy decyzyjne, a nawet w teorię gier. Innym pomysłem na rozwój projektu jest zastosowanie przeróżnych optymalizacji bądź mechanizmów poprawiających decyzyjność Minimaxa, takich jak *Quiescence Search*. Ponadto, można pomyśleć nad zintegrowaniem powstałego modelu sztucznej inteligencji z siecią internetową.

Bibliografia

[1] How checkers was solved. Web pages: https://www.theatlantic.com/technology/archive/2017/07/marion-tinsley-checkers/534111/.

[2] Quiescence search. Web pages: https://en.wikipedia.org/wiki/Quiescence˙search.

[3] I. Belda. *Umysł, maszyny i matematyka*. BUKA Books Sławomir Chojnacki, 2012.

[4] J. Mańdziuk, M. Kusiak, K. Walędzik. Evolutionary-based heuristic generators for checkers and give-away checkers. 2007.

[5] L. Pijanowski. *Przewodnik gier*. Iskry, 1978.

[6] S. Russel, P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, Inc., 2010.

[7] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, S. Sutphen. Checkers is solved. 2007.

[8] J. Schaeffer, R. Lake. Solving the game of checkers. 1996.

Załącznik A

Zawartość płyty CD

Na załączonej płycie CD znajdują się:

- **src** - folder z kodami źródłowymi;
- **doc** - dokumentacja projektu (JavaDoc);
- **README.txt** - instrukcja zbudowania projektu;
- **praca.pdf** - niniejsza praca dyplomowa, gotowa do wydruku.