Samouczenie programów szachowych

Praca magisterska
pisana na Wydziale Matematyki, Informatyki i Mechaniki UW
pod kierunkiem dr Jerzego Cytowskiego

Tomasz Michniewski [1]

Warszawa, 16 czerwca 1995 r.

[1] Obecny adres e–mail: tmichniewski@skrzynka.pl

Spis treści

1 Wprowadzenie **3**
1.1 Dlaczego warto pisać programy szachowe 3
1.2 Jak dotychczas pisano programy szachowe 4
1.3 Uczenie programów szachowych — dotychczasowe osiągnięcia . 7
1.4 Metoda uczenia 9

2 Przeszukiwanie drzewa gry **12**
2.1 Generator posunięć i ukryte w nim heurystyki 12
2.2 Metoda przeszukiwania i możliwości usprawnień 14
2.3 Funkcja oceniająca — wykorzystanie wiedzy eksperta 15
2.3.1 Dwadzieścia cztery parametry 16
2.3.2 Ocena materialna 16
2.3.3 Rozpoznawanie fazy gry 16
2.3.4 Ocena pozycyjna pionków 16
2.3.5 Matowanie 17
2.3.6 Ocena króla 17
2.3.7 Rozwój 18
2.3.8 Ocena pozycyjna skoczków 18
2.3.9 Ocena pozycyjna gońców i wież 18
2.3.10 Ocena pozycyjna hetmanów 18

3 Algorytmy genetyczne **19**
3.1 Opis ogólny 19
3.2 Zdefiniowanie problemu 21
3.3 Użyte operatory 22
3.3.1 Zakodowanie parametrów 22
3.3.2 Funkcja przystosowania 22
3.3.3 Selekcja 23
3.3.4 Krzyżowanie 24
3.3.5 Mutacja 24
3.3.6 Przebieg ewolucji 25
3.4 Opis implementacji algorytmu genetycznego 25

4 Wyniki eksperymentów **28**
4.1 Przebieg uczenia 28
4.2 Mecz sprawdzający 35
4.3 Mecz ze „średnią” 36
4.4 Superturniej 37
4.5 Wybrane parametry 38

5 Heurystyki **41**
5.1 Siła heurystyk 41
5.2 System klasyfikujący 42
5.2.1 Sieć Kohonena jako system klasyfikujący 43
5.2.2 Opis implementacji sieci Kohonena 44
5.2.3 Wnioski z przeprowadzonych testów 46
5.3 Podsumowanie 47

A Zbieżność wag neuronu **49**

B Załączona dyskietka **50**

C Słownik terminów **51**

1 Wprowadzenie

1.1 Dlaczego warto pisać programy szachowe

Napisanie silnego programu szachowego pociągało ludzi od dawna. Początkowo, w latach pięćdziesiątych, głównym celem było pokazanie, że komputer też potrafi robić to, co człowiek. Pierwszy krok został zrobiony w 1950 roku przez Claude'a Shannona [20]. Zaobserwował on, że szachy są grą skończoną: dla każdej pozycji istnieje skończona liczba możliwych posunięć i po skończonej liczbie ruchów partia musi się zakończyć. Można zatem dać kompletny opis strategii gry dla gracza. Znając drzewo gry oraz oceny liści (jako oceny końcowych pozycji) można przypisać pozostałym węzłom wartości ze zbioru $\{0, \frac{1}{2}, 1\}$. Strategia ta, pod nazwą strategii minimaksowej, jest podstawą wszystkich dotychczasowych prób zaprogramowania gry w szachy. Dokładny jej opis można znaleźć np. w [4, 16, 18, 21].

Należy tu podkreślić, że gdyby dało się przejrzeć całe drzewo gry, to gra przestałaby być ciekawa — partia byłaby zakończona już przed rozpoczęciem.

W 1953 roku A.Turing opierając się na propozycjach Shannona ułożył program, który był tak prosty, że mógł być wykonywany ręcznie, bez pomocy maszyny cyfrowej [23]. Widać było już wtedy, że podstawowe problemy zostały przezwyciężone.

Dzisiaj na programy szachowe patrzymy zupełnie inaczej. W graniu przez komputer w szachy nie ma już nic ciekawego (małą zachętą jest nagroda w wysokości 100 tys. $ dla pierwszego komputera, który pokona mistrza świata). Dziś interesują nas analogie między różnymi dziedzinami intelektualnej działalności człowieka oraz próby wykorzystania rozwiązań jednych problemów do rozwikłania innych. Dziedziną wiedzy, która się tym zajmuje jest — szeroko pojmowana — sztuczna inteligencja. Ogólnie można powiedzieć, że szachy są poligonem, na którym można wypróbować różne metody postępowania i rozwiązywania dowolnych problemów decyzyjnych. Pozwalają one dostrzec siłę heurystyk i ich wpływ na czas znajdowania rozwiązania. Pośrednio przyczyniają się także do zrozumienia sposobu działania ludzkiego mózgu.

Znane są próby wykorzystania teorii gier do gry na giełdzie (gra bez pełnej informacji). Były mistrz świata w szachach, Michaił Botwinnik, dostrzegł pewne analogie między szachami i systemami kierowania złożonymi procesami. Udowodnił możliwość wykorzystania pomysłów z dziedziny programów szachowych do automatyzacji sterowania. Pod jego kierunkiem powstał program planowania remontów obiektów elektrotechnicznych.

Moim zdaniem przed dzisiejszymi szachami komputerowymi stoją dwa cele:

1. Opracowanie metod polepszania siły gry (bez udziału programisty).

2. Stworzenie metod wykorzystania silnych heurystyk.

Jeśli chodzi o punkt 1, to wydaje się, że programy uczące się (niekoniecznie gry w szachy) stopniowo wyprą pozostałe. Już dziś słyszy się o programach antywirusowych opartych na sieciach neuronowych, które są w stanie rozpoznać nowe, nieznane im wirusy na podstawie podobieństwa do innych, znanych. Innym przykładem mogą być np. inteligentne strategie obsługi dysku w systemach wielozadaniowych, gdzie użyta strategia zależna byłaby od zgłoszonych uprzednio żądań. Wreszcie systemy doradcze, gdzie miarą jakości jest właśnie zdolność do przyswajania nowej wiedzy. Dlatego wydaje mi się, że samouczące się programy szachowe są bardzo ważne zarówno dla szachistów, jak i sztucznej inteligencji.

Natomiast, jeśli chodzi o punkt 2, to jest on pewnego rodzaju alternatywą dla punktu 1. Jak dotychczas wszystkie programy szachowe wykorzystywały pewne heurystyki, jednakże nie były one tak silne, aby można było naśladować proces myślenia człowieka przy grze w szachy. Nie słyszałem również, aby komukolwiek udało się tego dokonać (tzn. zasymulować działanie ludzkiego mózgu). Tym niemniej jest to bardzo interesujące. Skoro człowiek, dzięki heurystycznemu przeszukiwaniu, analizując nie więcej niż 100 pozycji potrafi pokonać komputer, który przegląda miliony pozycji na sekundę, to coś w tym musi być.

1.2 Jak dotychczas pisano programy szachowe

Większość znanych programów szachowych jest napisana w ten sam sposób: program analizuje drzewo gry wykorzystując alfa-beta obcięcie [4, 11, 16]. Wymienić tu można tak świetne programy jak: Deep Thought, Hitech czy Cray Blitz [16]. Prekursorem w tej dziedzinie był program CHESS 4.X (w wielu wersjach) napisany w latach siedemdziesiątych przez dwóch studentów (Slate, Atkin) [7]. Napisany przez nich program wygrał mistrzostwa USA programów szachowych w latach 1970–73 i 1975 oraz zajął drugie miejsce w 1974 roku.

Program ten był pierwszym, w którym — z powodzeniem — wykorzystano tyle różnych ciekawych pomysłów. Był to program napisany zgodnie z „regułami sztuki". Użycie tablic transpozycyjnych, tablica ruchów morderców (ang. killer table), iteracyjnie pogłębiana alfa-beta, biblioteka debiutów, programowanie w assemblerze — wszystko to stało się standardem dla późniejszych programów szachowych [7, 14, 16]. Późniejsze programy takie jak: Deep Thought, Hitech czy Cray Blitz były tylko ulepszeniami tego schematu.

Programowanie w assemblerze zastąpiono układami scalonymi i programowaniem za pomocą mikrorozkazów (Deep Thought i Hitech), dzięki czemu program pracujący na stacji roboczej Sun mógł wygrać z programem Cray Blitz działającym na najszybszym komputerze świata.

Technika tablic transpozycyjnych (jako uczenie się na pamięć) pozwalała szybko uczyć się debiutów (samodzielnie!) oraz — słabiej — końcówek. Natomiast w grze środkowej, z uwagi na ogromną liczbę możliwych pozycji (między 10^{40} a 10^{60}) była nieprzydatna. Próbowano temu zaradzić przy pomocy tzw. „brutalnej siły". Już Slate i Atkin początkową wersję programu napisaną w Fortranie przepisali na assembler komputera CDC 6000/Cyber. Znane już były nieudane próby dokonywania selektywnego przeszukiwania drzewa gry. Zamiast w skomplikowany sposób odrzucać złe ruchy, generator posunięć generuje wszystkie możliwe posunięcia. Dzięki takiemu uproszczeniu generatora, można było zwiększyć analizę o 2–3 półruchy. Pozwalało to na znaczną poprawę siły gry programu.

Wydaje mi się jednak, że to nie te programy były krokiem milowym w tej dziedzinie. Już bowiem w 1959 roku w IBM Journal of Research and Development ukazał się artykuł A.L.Samuela, w którym była zawarta większość powyższych idei [19]. Była to praca poświęcona uczeniu się maszyn na przykładzie gry w warcaby amerykańskie (pionki chodzą i biją tylko do przodu, damki chodzą i biją do przodu i do tyłu, ale mogą przemieszczać się tylko o jedno pole — a więc jest to uproszczona wersja warcabów znanych w Polsce).

Program napisany był na maszynę IBM 704. Wykorzystano w nim możliwości komputera do maksimum, np. zakodowano pozycje na bitach 36 bitowych słów maszyny, dzięki czemu do wygenerowania wszystkich posunięć pionkami z danej pozycji należało wykonać 4 (słownie cztery) rozkazy procesora. Komputer współpracował z taśmą magnetyczną (nie było jeszcze dysków), program wprowadzało się na kartach perforowanych, a wyniki były drukowane na drukarce. Dziś można powiedzieć, że to archaiczny komputer.

Na taśmie była przechowywana baza pozycji wraz z ocenami w kilku listach odpowiednio posortowanych i pokatalogowanych tak, aby pozycje wcześniej potrzebne, były na początku. Pozycje były dzielone na grupy według ilości materiału na szachownicy, kolejności posunięcia, itd. Pozycje identyczne z dokładnością do symetrii były traktowane jak jedna (nasuwa się porównanie do programu Kena Thompsona do bezbłędnego rozgrywania końcówek opisanego między innymi w [2], [16, str. 13–16]). Baza pozycji, nazywana przez autorów spisem pozycji, nie była niczym innym, jak właśnie tablicą transpozycyjną. Podręczna pamięć taśmowa mogła zawierać 53 tys. pozycji. Dzięki temu program nauczył się świetnie rozgrywać debiuty oraz rozpoznawać większość wygranych i przegranych pozycji w grze końcowej.

Jednakże w grze środkowej uczenie na pamięć okazało się mało skuteczne.

Aby zmienić tą sytuację autorzy zastosowali metodę losowych zmian współczynników wielomianu oceniającego. Został zaprojektowany liniowy wielomian oceniający, którego wyrazy zostały wybrane w mniej lub bardziej przypadkowy sposób — wypróbowano dla oceny pozycji pewną liczbę abstrakcyjnych schematów takiego wielomianu, nie mających nic wspólnego z pojęciami warcabowymi. Jedną z bardziej interesujących takich prób była ocena pozycji warcabowej za pomocą pierwszego i wyższych momentów osobno białych i czarnych bierek względem prostopadłych osi szachownicy. Ostatecznie został wybrany schemat uwzględniający rozwój, opanowanie centrum, blokadę, ruchliwość, opanowanie centrum przez damki, itp. (w sumie kilkadziesiąt parametrów wśród których od 4 do 16 było wykorzystywanych podczas gry). Sposób doboru tych parametrów zostanie opisany w rozdziale 1.3.

Przeszukiwanie drzewa gry było wykonywane na ustaloną (niedużą) głębokość (nie używano wtedy procedury cięć alfa-beta), po czym program przechodził w obszar selektywnego przeszukiwania — rozpatrując tylko ruchy, które są biciami, są odpowiedzią na bicie lub możliwe jest doprowadzenie do wymiany. Zatem widać tu podobieństwo do schematu B Shannona, w którym zamiast wszystkich rozpatruje się tylko „sensowne" ruchy. W dzisiejszej terminologii takie przeszukiwanie nosi nazwę wariantu forsownego. Największy wkład w ten rodzaj przeszukiwania wnieśli Rosjanie Adelson-Wielskij, Arłazarow i Donskoj. Dostrzegli oni, że liczba odpowiedzi na szacha, groźbę mata, czy bicie będące wymianą jest znacznie mniejsza, niż liczba wszystkich ruchów w danej pozycji. W takich sytuacjach gra nosi forsowny charakter. Pozostałe ruchy można odrzucić przy pomocy prostych reguł. I tak — w odpowiedzi na szacha należy odejść królem, przesłonić linię szacha lub zbić szachującą bierkę. Pozostałe ruchy dają bowiem możliwość zbicia króla, (aby to sprawdzić nie trzeba analizować drzewa gry). W przypadku groźby mata, jeśli wybierzemy zły ruch, to po prostu dostaniemy mata. W przeciwnym przypadku (tym pożądanym) gra toczy się dalej. W odpowiedzi na bicie najczęściej też należy coś zbić. W pozycjach, w których wszystkie bicia są złe, dopuszcza się tzw. pusty ruch (ang. null move). Dokładną analizę wariantu forsownego można znaleźć między innymi w [1].

Ostatnim prezentowanym tu podejściem do gry w szachy będzie pomysł eks-mistrza świata Michaiła Botwinnika [1, 5]. Botwinnik chciał skonstruować algorytm do gry w szachy, który analizowałby tylko ruchy służące pewnym konkretnym celom; (a więc jest to selektywna analiza wybranych ruchów od samego początku — w przeciwieństwie do wariantu forsownego, który stosuje się dopiero na pewnej głębokości). Zdaniem Botwinnika — szachisty, który przez 15 lat był mistrzem świata — każdą operację na sza-

chownicy można sprowadzić do wieloruchowego przemieszczenia figur w celu zbicia pewnych bierek przeciwnika. Jeśli nawet przeciwnik zdoła się obronić, to w procesie walki o zdobycie przewagi materialnej można dostrzec inny cel.

Botwinnik porównuje grę do funkcjonowania wielopoziomowego systemu sterowania. „Cele" niskiego poziomu, to zwykłe przejścia bierek po krótkich (dwuruchowych) trajektoriach. Cele wyższego poziomu, to ułożenie w drzewo celów niskiego poziomu w taki sposób, aby osiągnięcie celów będących niżej w drzewie gwarantowało osiągnięcie celów leżących wyżej. Intensywnie korzystając z liczb zespolonych, sprowadził grę do „odpowiedniej wymiany". To walczące strony muszą dokonać oceny, czy podczas wymiany (niekoniecznie bierek, ale również wartości pozycyjnych) więcej zyskują niż tracą. Jednak na dzień dzisiejszy program Botwinnika nie jest ukończony. W szachach komputerowych, podobnie jak w życiu, swoje racje należy udowadniać przy szachownicy.

1.3 Uczenie programów szachowych — dotychczasowe osiągnięcia

Przed przejściem do omówienia uczenia programów szachowych należy zdefiniować sam proces uczenia. Przez uczenie rozumieć tu będziemy zarówno nabywanie nowej wiedzy np. $6! = 720$ (zapamiętane, a nie obliczone), jak i nabywanie nowych umiejętności, w szczególności nabywanie umiejętności obliczania pewnych funkcji.

Uczenie na pamięć — jako prostsze — zostało dokładniej przebadane. Z uczeniem tego typu wiąże się nierozerwalnie pojęcie pojemności pamięci. Nawet, jeśliby była bardzo duża, to i tak kiedyś się zapcha. Wtedy można zakończyć proces uczenia. Jednak we wszystkich danych istnieją informacje mniej (rzadziej) przydatne. Dlatego lepsze efekty uzyskuje się dopuszczając do zapominania. Należy tylko pamiętać, jak często korzystamy z danej informacji. Ten schemat uczenia na pamięć został z powodzeniem zastosowany w programie Samuela [6, 19]. Program nauczył się rozgrywać debiuty tak, jakby przeczytał wszystkie podręczniki. Później ten schemat był wielokrotnie użyty w programach szachowych (tablice transpozycyjne). Z pewną „lokalną" odmianą tego typu uczenia mamy do czynienia przy heurystyce tablicy ruchów morderców. Jest to również uczenie się na pamięć, ale na bardzo krótki („lokalny") okres. Jesteśmy przy tym świadomi tego, że po kilku ruchach (a już na pewno po zakończeniu partii) informacja przestaje być aktualna.

Zatem w przypadku, gdy zbiór sytuacji, w których należy podejmować decyzje nie jest zbyt duży, można skutecznie stosować zapamiętywanie na pamięć. Ale wystarczy niewielki wzrost tego zbioru, a sytuacja się pogar-

sza. Przypomnijmy, program Samuela nauczył się rozpoznawać większość wygranych i przegranych końcówek, podczas gdy w końcówkach szachowych, aby uzyskać podobne rezultaty (lub lepsze) należało dla końcówki o danej konfiguracji materiału na szachownicy (np. KHKW) zapamiętać wszystkie pozycje, co wymagało ponad 100 MB miejsca na dysku (po spakowaniu) (słynna baza Kena Thompsona [2], [16, str. 13–16]). Zatem jak widać, w trudnych przypadkach, komputer sam nie zdoła się wszystkiego nauczyć i potrzebna jest metoda systematycznego nagromadzenia informacji.

Dużo ciekawszym i bardziej obiecującym rodzajem uczenia się jest uogólnianie. Pewnego rodzaju pionierskie rozwiązanie było zaprezentowane w programie Samuela. Otóż, jak pamiętamy z poprzedniego rozdziału, funkcja oceniająca była liniowym wielomianem. Zostało zdefiniowanych 38 wyrazów rozpoznających pewne cechy pozycji na szachownicy. Uczenie miało polegać na wyborze spośród nich 16 najbardziej użytecznych i takim dobraniu współczynników, aby otrzymać najlepszą funkcję oceniającą. Przy czym użyte tu pojęcie najlepsza oznacza najlepszą przy danej metodzie przeszukiwania drzewa gry, danej głębokości, danych (zdefiniowanych) wyrazach wielomianu, itd. W rzeczywistości robimy pewne uproszczenie, ponieważ zakładamy, że najlepsza funkcja (jako zbiór 16 wyrazów i odpowiadających im współczynników) jest najlepsza także dla innych głębokości analizy.

Program Samuela został napisany tak, aby podczas uczenia zachowywał się jak dwaj różni gracze, nazwani przez autora Alfa i Beta (nie należy tego mylić z procedurą cięć alfa-beta). Beta posługiwała się stałym wielomianem oceniającym, natomiast Alfa miała zmieniany wielomian co pewien czas (wymiana jednego bądź kilku współczynników). Jeżeli Alfa wygrywała z Betą, to jej system oceny był przekazywany Becie. Jeżeli natomiast Alfa przegrała kilka (zwykle trzy) partii, to przyjmowano, że jest na niewłaściwej drodze i dokonywano bardziej drastycznej zmiany w wielomianie oceniającym (zmieniając wartość największego współczynnika na zero). Ta czynność była konieczna, ponieważ cały proces uczenia się był poszukiwaniem najwyższego punktu w wielowymiarowej przestrzeni ocen przy istnieniu wielu lokalnych maksimów, na których program mógł utknąć. Zatem mieliśmy tu do czynienia z algorytmem zachłannym (ang. hill-climbing).

W programie Samuela połączenie uczenia na pamięć z uogólnianiem dało znakomite rezultaty. 12 lipca 1962 roku w Yorktown rozegrano partię między bardzo silnym graczem warcabowym i programem Samuela. Robert W. Nealey był wtedy jednym z najlepszych graczy amerykańskich. Partia zakończyła się zwycięstwem programu.

Oto co powiedział po partii R.Nealey:

„Nasza partia ... miała kilka przełomowych momentów. Cały po-

> czątek partii, aż do 31 posunięcia był już poprzednio publikowany, przy czym kilka razy unikałem posunięć najczęściej podawanych w książkach, bezskutecznie próbując sprowadzić maszynę na nieznane jej tory. O ile mogłem sprawdzić, począwszy od przegrywającego 32 posunięcia, cała reszta partii jest całkowicie oryginalna. Wydaje mi się godne podkreślenia, że maszyna, aby uzyskać zwycięstwo, musiała wykonać kilka błyskotliwych posunięć i że gdyby ich nie zrobiła osiągnąłbym remis. Dlatego przedłużałem grę. Jednakże maszyna rozegrała zakończenie partii bezbłędnie. Jeśli chodzi o końcówkę, to nie spotkałem wśród ludzi tak silnego partnera od 1954 roku, kiedy przegrałem partię po raz ostatni."

Opis programu i przebieg partii można znaleźć w [6].
Tak oto rozpoczęła się era programów samouczących się.

1.4 Metoda uczenia

Jak widzieliśmy w warcabach program komputerowy uzyskał poziom mistrza. Szachy są jednak bardziej interesujące, ponieważ są bardziej zbliżone do życia. Większy niż w warcabach stopień komplikacji, większa liczba wyjątków. Wszystko to sprawia, że są pewnego rodzaju wyzwaniem dla człowieka. Szachiści uczą się grania, a informatycy piszą programy, które będą mogły z nimi rywalizować. I jak na razie, w przeciwieństwie do warcabów, są grą żywą (w warcabach dominuje pogląd, że przy prawidłowej grze z obydwu stron, nikt nie może wygrać).

W takim stanie opublikowanej wiedzy na temat programów grających w gry postanowiłem wypróbować inne podejście. Metoda uczenia, którą chcę zaproponować jest oparta na mechanizmach ewolucji i ich komputerowej symulacji — algorytmie genetycznym (dokładniejszy jego opis znajduje się w rozdziale 3).

Pomysł narodził się w czerwcu 1994 roku. Przeszukiwanie drzewa gry, ze względu na złożoność, może przebiegać tylko do skończonej i dość ograniczonej głębokości. Powstałe w ten sposób pozycje w liściach drzewa poszukiwań należy poddać ocenie. I tu pojawia się problem. Otóż, z góry nie wiadomo, które czynniki należy brać pod uwagę. Które są ważniejsze, a które nieistotne, kiedy należy brać pod uwagę tylko ich podzbiór. Nie wiadomo nawet, czy wyjściowa pozycja jest remisowa, czy nie. Dlatego została zaprojektowana metoda uczenia, której celem jest stworzenie (wyprodukowanie) funkcji oceniającej.

Na czym ona polega? Przy wykorzystaniu wiedzy eksperta została zaprogramowana funkcja oceniająca, uwzględniająca podstawowe pojęcia szachowe

takie jak: wartość materialna bierek, aktywność i ruchliwość figur, bezpieczeństwo króla, otwarcie centrum, zaawansowanie pionków, itp. Dokładny jej opis znajduje się w rozdziale 2.3. Jednak w odróżnieniu od standardowego podejścia, gdzie parametry wpływające na działanie funkcji ustalone są z góry, tutaj nie zostały zainicjalizowane w ogóle, a dokładniej zostały zainicjalizowane losowo. Przy tradycyjnym podejściu myślimy w ten sposób: „końcówka rozpoczyna się wtedy, gdy wartość materiału na szachownicy nie przekracza ...". No właśnie — ile? Tutaj rozpoczynają się próby, by po wielu godzinach testów ustalić, że taki parametr wynosi ok. 1500 (przykładowo). Podejście z algorytmem genetycznym jest zupełnie inne. Mówimy: „końcówka rozpoczyna się wtedy, gdy wartość materiału na szachownicy nie przekracza wartości zmiennej *MateriałWKońcówce*". Natomiast wartość tej zmiennej zostanie ustalona przez algorytm genetyczny.

Takich zmiennych jest w obecnej postaci 24. Zatem jeden zestaw takich zmiennych, to jedna funkcja oceniająca. Inny zestaw, to inna funkcja. Rozpatrując pewien zbiór takich zestawów (czy jak kto woli — funkcji) można zadawać pytanie, które są lepsze, a które gorsze. Informację taką można uzyskać np. rozgrywając między nimi turniej „każdy z każdym", czy w dowolny inny sposób (pod warunkiem, że sprawiedliwy i miarodajny). Następnie z tych lepszych można wyprodukować potencjalnie jeszcze lepsze. Można też wyprodukować gorsze. Zadaniem algorytmu genetycznego jest odrzucenie złych i wydobycie istotnych parametrów z tych lepszych.

Chcę tutaj zaznaczyć, że począwszy od zaprogramowania funkcji oceniającej oraz operatorów genetycznych, dalsza część poszukiwań odbywa się bez udziału programisty i eksperta szachowego. Nie trzeba godzinami rozgrywać partii z programem. Włączamy komputer i czekamy na wyniki (cierpliwie).

Nasuwa się tu porównanie do programu Samuela. Tam jednak mieliśmy do czynienia z jednym programem i jedną jego mutacją. Zatem była to próba losowego poprawiania rozwiązania (całkiem skuteczna). Tutaj mamy możliwość jednocześnie poszukiwać optymalny zestaw parametrów w kilku miejscach przestrzeni parametrów na raz.

Pojęcie optymalny należałoby w zasadzie ująć w cudzysłów. Optymalny zestaw jest bowiem najlepszy tylko dla jednej konkretnej funkcji oceniającej. Jeśli okaże się, że funkcja oceniająca nie uwzględnia pewnej własności pozycji, (która powinna być uwzględniona) to „optymalny" zestaw parametrów nie będzie już optymalny.

Dla uproszczenia założymy, że taka sytuacja nie zachodzi, to znaczy funkcja oceniająca jest wystarczająco ogólna. Lub inaczej mówiąc, skupimy się na poszukiwaniu optymalnego zestawu parametrów dla danej, konkretnej funkcji. Zwłaszcza, że modyfikować funkcję, zdefiniowaną językiem abstrakcyjnym i pojęciami szachowymi, jest znacznie łatwiej niż zbiór liczb. Nasuwa

się tu porównanie do problemu regulacji telewizora.

Działanie algorytmu genetycznego może być trochę zaburzone faktem, że funkcja przystosowania (wynik w turnieju) jest obarczona błędem. Algorytm genetyczny może być oszukiwany np. jeśli p_1, p_2 i p_3 są zestawami parametrów, $f(p_1)$, $f(p_2)$, $f(p_3)$ odpowiadającymi im funkcjami oceniającymi i $f(p_1)$ jest lepsza od $f(p_2)$, a $f(p_2)$ jest lepsza od $f(p_3)$, to z tego wcale nie wynika, że $f(p_1)$ jest lepsza od $f(p_3)$. Może się bowiem zdarzyć, że słabe strony $f(p_1)$ będą bardziej się dawały odczuć właśnie przy grze z $f(p_3)$. Dokładniej te i inne problemy zostaną omówione w rozdziale 3.

2 Przeszukiwanie drzewa gry

2.1 Generator posunięć i ukryte w nim heurystyki

Jako metoda przeszukiwania drzewa gry została wybrana procedura cięć alfa-beta [4, 11, 14, 16, 21]. Wiadomo, że działa ona tym szybciej im w procesie przeszukiwania będzie więcej cięć. Dlatego bardzo ważne jest, aby ruchy potencjalnie lepsze, były sprawdzane wcześniej. W przypadku, gdy ruchy są generowane od najgorszego do najlepszego (w sensie oceny minimaksowej), to w drzewie poszukiwań nie ma żadnych cięć i alfa-beta zachowuje się jak minmax.

Jako język programowania została wybrana z uwagi na czytelność języka i dobrą jakość generowanego kodu obiektowa TopSpeed Modula w implementacji firmy JPI. Programowanie w C zostało odrzucone od samego początku. Celem wszystkich prac programistycznych było bowiem przeprowadzenie testów i eksperymentów, które pozwoliłyby pokazać, że program szachowy może się uczyć. Natomiast wyprodukowanie komercyjnego programu szachowego nie było nigdy poważnie rozważane.

Szachownica jest reprezentowana jako jednowymiarowa tablica indeksowana od 1 do 64, gdzie pole $a1 = 1$, $b1 = 2$, ..., $g8 = 63$, $h8 = 64$ jak na rysunku.

57	58	59	60	61	62	63	64
49	50	51	52	53	54	55	56
41	42	43	44	45	46	47	48
33	34	35	36	37	38	39	40
25	26	27	28	29	30	31	32
17	18	19	20	21	22	23	24
9	10	11	12	13	14	15	16
1	2	3	4	5	6	7	8

Generowanie ruchów odbywa się zawsze ze strony białych. Jeśli program rozpatruje pozycję ze strony czarnych, to wstępnie następuje obrócenie szachownicy. Wiąże się to z pewnymi nakładami czasowymi podczas obliczeń, ale za to upraszcza procedury, w których nie trzeba za każdym razem uwzględniać koloru lub pisać symetrycznych procedur dla białych i czarnych.

Przed rozpoczęciem przeszukiwania drzewa gry wyznaczane są i zapamiętywane pozycje, na których znajdują się bierki jednej i drugiej strony. Dzięki temu w późniejszej analizie nie trzeba przeglądać całej szachownicy. Przydaje się to zwłaszcza w końcówce, gdy szachownica jest już prawie pusta.

Aby zwiększyć liczbę cięć generator posunięć, jako pierwsze, zwraca ruchy prowadzące do zysku materialnego (np. bicie, promocja) oraz ruchy szachu-

jące króla. Każdy ruch ma określony atrybut o nazwie zysk. Generator zna wartości bierek:

- król = 20000
- hetman = 900
- wieża = 500
- goniec = 330
- skoczek = 300
- pionek = 100

Są to jego wewnętrzne dane, które nie są przekazywane do funkcji oceniających. Wpływają one tylko na szybkość przeszukiwania drzewa gry. Dla funkcji, które podobnie oceniają wartości bierek, będzie zwiększał liczbę cięć. Dla pozostałych może natomiast wydłużyć analizę. Niezależnie jednak od kolejności generowanych ruchów alfa-beta zawsze zwraca ten sam wynik — wartość minimaksową pozycji (zależną od funkcji oceniającej, a nie od powyższych wartości bierek). Dla ruchu, który jest zbiciem np. skoczka atrybut zysk równa się 300, dla promocji w hetmana zysk wynosi $900 - 100$, dla promocji w gońca w połączeniu ze zbiciem hetmana przeciwnika zysk równa się $900 + 330 - 100$. Wynika stąd, że dla ruchu, który jest zbiciem króla, zysk $\geqslant 20000$. Dodatkowo, za każdy ruch, który jest szachem bezpośrednim, (a nie z odsłony), doliczana jest premia w wysokości 1000 punktów (szachy z odsłony nie są tu już uwzględniane z uwagi na koszt wykrywania ich). Uzyskane w ten sposób „wartości" ruchów pozwalają uporządkować je od najlepszego (w sensie atrybutu zysk). Nie są sortowane wszystkie ruchy, lecz tylko ta ich część, dla której atrybut zysk > 0. Sortowanie powyższe często prowadzi do analizy ruchów, które są bez sensu, np. bicie hetmanem bronionego pionka. Takie ruchy są jednak błyskawicznie obalane w następnym ruchu właśnie dzięki sortowaniu.

Aby przyśpieszyć generowanie posunięć została zaprojektowana metoda szybkiego generowania ruchów przy wykorzystaniu dużej ilości przygotowanych wcześniej danych (tzw. look-up table). Dla każdego pola i każdej bierki, oprócz hetmana i pionka, zostały przygotowane tablice, w których znajdują się wszystkie możliwe ruchy z danego pola szachownicy. Dla pionków, z uwagi na małą liczbę ruchów, nie trzeba było tego robić. Natomiast wygenerowanie ruchów hetmana z danego pola jest równoważne wygenerowaniu ruchów gońca i wieży.

Przykładowo wygenerowanie wszystkich ruchów wieży z pola skąd wygląda następująco:

```
FOR kierunek:=1 TO 4 DO
   k:=RuchyWieży[skąd][kierunek][1];
   i:=1;
   WHILE (k ≠ 0) AND (pozycja[k] = NIC) DO (* zero jest strażnikiem *)
       DopiszRuch(skąd,k); (* zapamiętanie ruchu do pola o numerze k *)
       INC(i);
       k:=RuchyWieży[skąd][kierunek][i];
   END;

   (* jeśli na pozycji k znajduje się czarna bierka, to próbujemy ją zbić *)
   IF pozycja[k] < 0 THEN DopiszRuch(skąd,k); END;
END;
```

Analogicznie generowane są ruchy dla gońca. Dla skoczka i króla jest jeszcze prościej, ponieważ nie ma kierunku. Generator nie sprawdza, czy nastąpiło zbicie nieprzyjacielskiego króla. Robi to dopiero moduł przeszukujący.

Dzięki takim usprawnieniom generator posunięć jest bardzo szybki. Na komputerze 386DX 40 MHz w ciągu sekundy możliwe jest wygenerowanie wszystkich posunięć dla ponad 5000 pozycji.

2.2 Metoda przeszukiwania i możliwości usprawnień

Obszar przeszukiwania został podzielony na dwa fragmenty. Pierwsza faza jest pełnym przeszukiwaniem na pewną głębokość, będącą parametrem. Jeśli w procesie analizy program zejdzie w drzewie na taką głębokość, to nie kończy, lecz rozpoczyna fazę drugą — wariant forsowny. Został on zaprogramowany niezależnie od opisu wariantu forsownego w [1]. W fazie drugiej — wariancie forsownym są analizowane tylko bicia oraz szachy na ustaloną głębokość, będącą również parametrem programu. Jako wynik zwracana jest wartość najlepszego wariantu lub wartość bieżącej pozycji, jeśli okaże się, że wszystkie bicia i szachy prowadzą do pogorszenia pozycji. W odpowiedzi na szacha, który pojawi się w obszarze selektywnego przeszukiwania, rozpatrywane są wszystkie ruchy (zwiększa się głębokość pierwszego obszaru o 1 (z zera, które oznacza wejście w wariant forsowny)). Z uwagi na to, że alfa-beta zwraca ocenę pozycji, a nas interesuje najlepszy ruch, została zaprogramowana procedura, która wykonuje pierwsze ruchy i dalej korzysta z alfa-bety. W tej procedurze, nazwanej NajlepszyRuch, głębokość pierwszego obszaru przeszukiwania jest również zwiększana o 1, jeśli biały król (król strony będącej na posunięciu) jest szachowany w początkowej pozycji.

Takie zwiększanie głębokości jest istotne, ponieważ szach może radykalnie

zmienić ocenę pozycji, a odpowiedzi na szacha jest mało. W przypadku, gdy głębokość pierwszego obszaru przeszukiwania wynosi n_1, a głębokość wariantu forsownego wynosi n_2 i w pozycji początkowej król jest szachowany oraz w wariancie forsownym połowa ruchów jest szachem (jedna strona ciągle szachuje), głębokość może się zwiększyć do $G = 1 + n_1 + 2n_2$, np. $n_1 = n_2 = 3 \rightarrow G = 10$. Dodatkowo głębokość tą może zwiększyć sytuacja, gdy odpowiedzią na szacha jest szach (np. przy wymianie ciężkich figur w pobliżu jednego i drugiego króla).

W trakcie analizy, gdy zostanie wykryta możliwość zbicia króla, alfa-beta zwraca wartość równą 20000 minus odległość od korzenia w drzewie. W ten sposób, jeśli program zobaczy, że może dać mata, to wybierze najkrótszą drogę.

Moduł przeszukujący nie wykorzystuje techniki tablic transpozycyjnych, ponieważ funkcja oceniająca jest ciągle zmieniana w procesie uczenia i jako taka jest parametrem procedury NajlepszyRuch. Nie została również wykorzystana możliwość myślenia na czasie przeciwnika, gdyż najczęściej przeciwnikiem jednego programu jest inny program i procesor jest cały czas zajęty.

Aby przyśpieszyć rozgrywanie partii między programami można by przechowywać w pamięci drzewo gry z wygenerowanymi ruchami dla każdej pozycji, ale bez ocen (lub z ocenami obydwu przeciwników). Wykonanie bowiem jednego półruchu powoduje, że nadal jesteśmy w tym samym drzewie, tylko o jeden poziom niżej. Mogłoby to zlikwidować wielokrotne (zależnie od głębokości analizy) generowanie ruchów dla tej samej pozycji.

Można by również zaimplementować iteracyjne pogłębianie. Wtedy miarą sprawiedliwego wykorzystania procesora byłby czas, a nie głębokość.

Jeśli chodzi o grę w debiucie, to równoważnikiem tablic transpozycyjnych mogłaby być biblioteka debiutów (niezbędna w profesjonalnym programie). Jednakże takie rozwiązanie było konsekwentnie odrzucane jako niezgodne z ideą całego przedsięwzięcia (nie chodzi o naśladownictwo, ale o uczenie).

2.3 Funkcja oceniająca — wykorzystanie wiedzy eksperta

Zawsze się zastanawiałem, dlaczego w różnych programach szachowych wartości figur w stosunku do pionka są różne. Np. wartość skoczka to 2.9, 3 lub 3.5 wartości pionka; wartość gońca to 3.1, 3.3, 3.5, itd. Jak zobaczymy później, te wartości zależą w bardzo istotny sposób od konstrukcji funkcji oceniającej. W tym rozdziale nazwy parametrów będą pisane pochyloną czcionką.

2.3.1 Dwadzieścia cztery parametry

Funkcja oceniająca została skonstruowana w taki sposób, aby nie zaszywać w niej stałych (poza kilkoma wyjątkami). Jeśli tylko pojawiały się wątpliwości, co do pewnych stałych, to automatycznie stałe te stawały się zmiennymi. W ten sposób powstały 24 parametry.

Funkcja zwraca jedną liczbę typu INTEGER będącą oceną pozycji z punktu widzenia białych. W rzeczywistości niezależnie oblicza się ocenę białych i czarnych, po czym w wyniku zwraca się ich różnicę.

2.3.2 Ocena materialna

Wartość każdego pionka została ustalona na 100. Wartości pozostałych bierek (oprócz króla) są określone przez zmienne *Skoczek*, *Goniec*, *Wieża*, *Hetman*. Można przyjąć, że wartość króla wynosi 0 (pamiętamy, że w module przeszukującym miał on wartość 20000, ale tylko po to, żeby wykryć ewentualne jego zbicie).

2.3.3 Rozpoznawanie fazy gry

Zostały zdefiniowane trzy fazy gry. Pozycja jest klasyfikowana jako debiut, jeśli co najmniej 3 lekkie figury stoją na 1 lub 8 linii. W takiej sytuacji pozycja nie jest końcówką ani matowaniem. W rozdziale poświęconym sieciom nueronowym poznamy inną metodę rozpoznawania debiutu.

Jeśli pozycja nie jest debiutem, to za końcówkę uznajemy ją, gdy wartość materiału na szachownicy (obliczona j.w.) nie przekracza zmiennej *M*ateriałWKońcówce lub też na szachownicy jest co najmniej jeden hetman i wartość materiału na szachownicy nie przekracza 2 * *MateriałWKońcówce*.

Z matowaniem mamy do czynienia, jeśli pozycja została zaklasyfikowana jako końcówka i jedna ze stron ma przewagę większą niż *P*rzewagaPrzyMatowaniu.

Poniższy opis dotyczy oceny białych.

2.3.4 Ocena pozycyjna pionków

Za każdego izolowanego pionka (tj. takiego, który w sąsiedniej kolumnie nie ma sąsiada) jest odejmowana: *KaraZaIzolowanePionki*. Za każdego zdublowanego pionka (drugiego w kolumnie) jest odejmowana: *K*araZaZdwojonePionki (jeśli są trzy pionki w kolumnie, to mamy dwie kary). Jeśli na polach d2 i e2 stoją białe pionki, to od oceny białych jest odejmowana: *K*araZaD2E2.

Jeśli na co najmniej jednym z pól d2 i d3 stoi biały pion, to jest odejmowana: *KaraZaD2iD3* (jedna). Analogicznie uwzględnia się pionki na e2 i e3 (*KaraZaE2iE3*).

Powyższe kary za centralne pionki mają wymusić otwarcie centrum i ułatwienie rozwoju.

Jeśli jesteśmy w końcówce, to dla każdego pionka doliczamy do oceny pozycji premię za zaawansowanie pionka równą numerowi linii, w której pionek stoi * *PremiaZaZaawansowaniePionka* .

2.3.5 Matowanie

Przy matowaniu funkcja oceniająca traci swoją symetrię. Obliczana jest bowiem tylko dla strony matującej. Załóżmy, że tą stroną są białe. Do ich oceny doliczane jest 10 * odległość czarnego króla od środka szachownicy. Odległość od środka jest zdefiniowana przez poniższą tabelkę:

5	5	4	3	3	4	5	5
5	4	3	2	2	3	4	5
4	3	2	1	1	2	3	4
3	2	1	0	0	1	2	3
3	2	1	0	0	1	2	3
4	3	2	1	1	2	3	4
5	4	3	2	2	3	4	5
5	5	4	3	3	4	5	5

Następnie za każdą białą bierkę odejmujemy: 2 * jej odległość od czarnego króla. Odległość dwóch pól (x_1, y_1) i (x_2, y_2) to $\max(|x_1 - x_2|, |y_1 - y_2|)$ $(x_1, y_1, x_2, y_2$ ze zbioru $\{1, 2, 3, 4, 5, 6, 7, 8\})$

Jeśli mieliśmy do czynienia z matowaniem, to funkcja oceniająca kończy w tym momencie działanie.

2.3.6 Ocena króla

Poniższy opis znów dotyczy białych. Król białych jest bezpieczny, jeśli stoi na polu g1 za swoimi pionkami na f2 i g2 lub stoi na h1 za białymi pionkami na g2 i h2. Po drugiej (lewej) stronie szachownicy król też może być bezpieczny, jeśli zachodzą symetryczne przypadki.

Jeśli jesteśmy w końcówce, to odejmujemy od oceny białych wartość: *K*rólWCentrum * odległość białego króla od środka szachownicy. W pozostałych fazach gry (debiut lub gra środkowa (= nie debiut i nie końcówka)) dodajemy powyższą wartość do oceny. Dodatkowo bezpieczny król dostaje nagrodę pod nazwą: *PremiaZaBezpiecznegoKróla* . Zmienne te mają za zadanie wymusić ruch króla do centrum w końcówce oraz zabezpieczenie króla w debiucie i grze środkowej.

2.3.7 Rozwój

Niezależnie od fazy partii, (o ile nie jest to matowanie), za każdą lekką figurę na 1 linii odejmowana jest kara: *KaraZaBrakRozwoju*.

2.3.8 Ocena pozycyjna skoczków

W debiucie za każdego skoczka na 1 linii od oceny białych odejmuje się zmienną: *KaraZaBrakRozwojuSkoczka*. W końcówce za każdego białego skoczka odejmuje się od oceny białych wartość: *K*araZaOdległośćSkoczkaOdKróla * odległość tego skoczka od czarnego króla. W debiucie lub grze środkowej od oceny białych odejmuje się wartość: *K*araZaOdległośćSkoczkaOdCentrum * odległość skoczka od środka szachownicy.

2.3.9 Ocena pozycyjna gońców i wież

Za każdego białego gońca dolicza się do oceny białych wartość równą liczbie możliwych ruchów tego gońca * *PremiaZaIlośćRuchówGońca*. Analogicznie dla każdej wieży (używając zmiennej *PremiaZaIlośćRuchówWieży*). Dodatkowo, dla każdej wieży na 7 linii dolicza się *PremięDlaWieżyNa7*.

2.3.10 Ocena pozycyjna hetmanów

Jeśli jesteśmy w końcówce, to za każdego hetmana od oceny białych odejmuje się wartość: *KaraZaOdległośćHetmanaOdKróla* * odległość tegoż hetmana od czarnego króla. W przeciwnym przypadku od oceny białych odejmuje się wartość: *KaraZaOdległośćHetmanaOdCentrum* * odległość hetmana od środka szachownicy. Dodatkowo, jeśli czarny król nie jest bezpieczny, to hetman dostaje premię: *PremiaDlaHetmanaZaOsłabienieKrólaPrzeciwnika*.

Oceny ze strony czarnych są obliczane analogicznie. Oceną analizowanej pozycji jest różnica oceny białych i czarnych.

Zdaję sobie sprawę, że powyższy schemat funkcji oceniającej nie jest doskonały. Jest wielce prawdopodobne, że czytelnik o większym doświadczeniu szachowym ode mnie zdoła wskazać słabe punkty w tej funkcji i usprawnienia. Jednak do eksperymentów taka funkcja jest wystarczająca.

3 Algorytmy genetyczne

3.1 Opis ogólny

Algorytmy genetyczne [8, 10, 17] są probabilistycznymi metodami przeszukiwania. W przeciwieństwie do innych metod, pozwalają one poszukiwać ekstremum funkcji celu w wielu miejscach przestrzeni jednocześnie. Programy wykorzystujące algorytmy genetyczne osiągają swój cel stosując strategię doboru naturalnego. Na początku generuje się początkową populację, tzn. możliwe rozwiązania. W analogii do żywych organizmów, różne rozwiązania wymieniają między sobą „geny" poprzez mechanizm krzyżowania, a dzięki mutacji pojawiają się losowe zmiany w osobnikach. „Przeżywają" tylko rozwiązania najlepiej (lub stosunkowo dobrze) przystosowane, a ich „potomstwo" jest podstawą do szukania kolejnych rozwiązań.

Algorytmy genetyczne wymagają, aby zbiór parametrów w problemie optymalizacji był kodowany na łańcuchy skończonej długości (osobniki), w określonym, skończonym alfabecie. Każdy osobnik reprezentuje jeden punkt w przestrzeni parametrów. Populacja łańcuchów reprezentuje zbiór testowanych punktów. W każdej chwili znane są tylko wartości funkcji przystosowania, (którą może być sama funkcja celu) w aktualnie testowanych punktach przestrzeni parametrów. Preferowanie „lepszych" osobników i ich powielanie z możliwymi losowymi zmianami powoduje powstawanie nowych osobników, a zatem określenie nowych punktów w przestrzeni parametrów, które będą testowane w następnej chwili czasowej. Ponieważ algorytmy genetyczne przeszukują dziedzinę funkcji w wielu punktach równocześnie, jest szansa, że zostanie znalezione globalne ekstremum funkcji. Możliwe jest znalezienie kilku ekstremów lokalnych — różne szczyty mogą być znalezione przez różne łańcuchy w tej samej populacji.

Algorytmy genetyczne stosują losowe reguły przechodzenia w przestrzeni parametrów, co nie oznacza jednak, że jest to losowe szukanie (błądzenie). Algorytmy genetyczne preferują lepsze rozwiązania na podstawie których, przy pomocy losowych zmian, produkują nowe, potencjalnie jeszcze lepsze rozwiązania.

Działanie algorytmu genetycznego można opisać następującą procedurą:

```
PROCEDURE GeneticAlgorithm;
BEGIN
   t:=0;
   Initialize P(t); (* start z losowej populacji *)
   Evaluate P(t); (* obliczenie współczynników przystosowania populacji *)
   WHILE NOT StopCondition DO
      t:=t+1;
      Select P'(t) from P(t-1); (* wybranie osobników do reprodukcji *)
      Recombine P'(t); (* wyprodukowanie nowych osobników *)
      Evaluate P'(t); (* obliczenie współczynników przystosowania *)
      Generate P(t) from P'(t) and P(t-1); (* wygenerowanie nowej populacji *)
   END;
END;
```

Działanie algorytmu genetycznego rozpoczyna się od zainicjalizowania zbioru możliwych rozwiązań (populacji). Jeśli nie dysponujemy żadną wiedzą o rozwiązaniu optymalnym, to można początkową populację wybrać losowo. Następnie, dopóki w naszej populacji nie znajdziemy wystarczająco dobrego osobnika, tworzymy kolejne, nowe populacje. Każda następna powstaje tylko na podstawie poprzedniej oraz informacji o przystosowaniu poszczególnych osobników.

Pierwszą operacją, jaką wykonujemy jest wybranie (wyselekcjonowanie) „lepszych" osobników. Liczba potomków każdego elementu zależy od jego przystosowania. Najczęściej stosuje się regułę „ruletki" opisaną w rozdziale 3.3.

Następnie otrzymane osobniki kojarzy się w pary i stosuje się operatory krzyżowania, mutacji i inne (np. operator inwersji nie opisany tutaj). Operator krzyżowania działa na parze osobników. Jego zadaniem jest wymiana fragmentów kodów genetycznych obu osobników. Dzięki temu pewne własności rozwiązań mogą się zmienić nie niszcząc innych. Operator mutacji, działając na pojedynczych osobnikach, zmienia ich losowo wybrane fragmenty. W ten sposób w populacji mogą pojawić się osobniki o cechach, które wcześniej nie występowały. Dlatego mutację należy stosować rzadko, (aby poszukiwania nie stały się zwykłym losowym błądzeniem).

Po utworzeniu potomków można wybrać elementy do nowej populacji. W niektórych implementacjach stosuje się wybór najlepszych elementów spośród potomków (cała populacja „wymiera", „żyje" tylko najlepsza część potomstwa); w innych implementacjach — do następnej generacji są wybierane najlepsze elementy spośród zbioru „rodziców" i „potomków". Czasem jest stosowana strategia pośrednia — do następnej populacji bezwarunkowo przechodzi najlepszy element z populacji „rodziców", a resztę stanowi „po-

tomstwo".

W literaturze można spotkać opis wielu różnych wersji powyższych operatorów. Działanie operatorów genetycznych zależy w dużym stopniu od sposobu kodowania pojedynczych rozwiązań w genotyp. W zasadzie dla każdego kodowania można zdefiniować osobne wersje operatorów.

3.2 Zdefiniowanie problemu

Chcemy znaleźć najlepszą postać funkcji oceniającej. Postać funkcji oceniającej jest ustalona i optymalizacji podlegają jedynie 24 parametry opisane w rozdziale 2.3.

Należy tutaj podkreślić specyfikę tego problemu. Otóż, w dotychczasowych zastosowaniach, algorytm genetyczny, poruszając się po przestrzeni rozwiązań, próbował znaleźć punkt, dla którego wartość funkcji przystosowania byłaby dostatecznie duża. Funkcją przystosowania mogła być sama funkcja celu, (czyli funkcja, dla której chcieliśmy znaleźć ekstremum globalne) lub jej pewne uproszczenie. Nawet, jeśli nie była podana w postaci wzoru matematycznego, to zawsze można było obliczyć jej wartość dla każdego punktu jej określoności. W moim przypadku coś takiego, jak wartość funkcji celu (nie należy jej mylić z funkcją oceniającą) w ogóle nie istnieje — a dokładniej nikt jej nie zna. Tym bardziej nie można podać dla niej wzoru matematycznego. Mało tego, dla dwóch punktów z dziedziny rozwiązań nie można nawet jednoznacznie powiedzieć, który z nich jest lepszy. Wiąże się to z tym, że jedyną operacją, jaką można wykonać na osobnikach populacji (będących zestawami parametrów) jest rozegranie między nimi partii. Wynik takiej partii nie przesądza jednak o tym, który z nich jest lepszy. Tak, jak w życiu, aby wyłonić zwycięzcę, należy rozegrać większą liczbę partii. Wtedy można oczekiwać, że uzyskany wynik lepiej ocenia grę danego programu (rozumianego tutaj jako zestaw parametrów).

Z uwagi na takie zaburzenia w pojedynczych partiach, nie można mówić o przechodniości. Jeżeli p_1, p_2 i p_3 są zestawami parametrów i p_1 jest „lepszy" od p_2, a p_2 jest lepszy od p_3, to wcale p_1 nie musi być lepszy od p_3 (w pojedynczych partiach). Dlatego rozgrywając partie między programami, należy rozegrać taką ich ilość, aby wynik był bardziej wiarygodny. Wiąże się to jednak ze wzrostem czasu obliczeń. Jako rozwiązanie kompromisowe zostało wybrane rozgrywanie turnieju między programami metodą „każdy z każdym" po jednej partii. Wynik w takim turnieju jest wartością funkcji przystosowania dla danego programu. Sposób obliczania zostanie omówiony w rozdziale 3.3. Funkcja celu nie jest potrzebna. Interesuje nas bowiem tylko zestaw parametrów, a nie wartość jakiejś abstrakcyjnej funkcji celu obliczonej w punkcie optymalnym.

Kolejną różnicą w stosunku do tradycyjnego podejścia jest to, że wartość funkcji przystosowania dla danego osobnika może być różna w różnych pokoleniach, (bo wynik w turnieju zależy od przeciwników — a więc od samej populacji). Może się zatem okazać, że zamiast zbieżności algorytm genetyczny wpadnie w pewnego rodzaju „cykle" — z uwagi na to, że nie ma jednoznacznie określonej funkcji przystosowania. Jednakże takie użycie algorytmu genetycznego nie przeczy zdrowemu rozsądkowi. Algorytm genetyczny powstał bowiem jako próba naśladowania natury. W przypadku wyłaniania lepszego szachisty, (a ogólniej sportowca), również rozgrywa się turnieje i mecze. W życiu często mamy do czynienia z sytuacjami, w których pojęcie „lepszy" nie musi być dobrze określone, a mimo to może funkcjonować.

3.3 Użyte operatory

3.3.1 Zakodowanie parametrów

Funkcja oceniająca została zaprogramowana przy udziale eksperta (szachisty) i dlatego wszystkie parametry mają jasno określone znaczenie. Dlatego łatwo można stwierdzić, że pewne parametry mają bezsensowne wartości — np. taką bezsensowną wartością może być 0 jako wartość parametru skoczek. Aby uniknąć poszukiwania w obszarach, które są w oczywisty sposób nieinteresujące, dla każdego parametru została zdefiniowana dziedzina dopuszczalnych wartości. Np. dla parametru skoczek dziedzina ta, to przedział $[200, 500]$ — 301 możliwych wartości (parametry są liczbami typu INTEGER). Dzięki temu rozmiar przestrzeni poszukiwań został zredukowany z około 10^{115} do około 10^{45}. W trakcie ewolucji algorytmu genetycznego parametry są reprezentowane jako liczby typu INTEGER, a wszystkie operatory są tak zdefiniowane, aby nie wykroczyć poza dopuszczalne dziedziny.

3.3.2 Funkcja przystosowania

Jeden krok algorytmu to rozegranie turnieju „każdy z każdym". Jeśli rozmiar populacji wynosi N (zawsze parzysty), to wymaga to rozegrania $\frac{N(N-1)}{2}$ partii. Programy grają na przemian białym i czarnym kolorem. Partie trwają ustaloną liczbę ruchów (zwykle 70). Za zwycięstwo przez zamatowanie przeciwnika do oceny zwycięzcy dolicza się 20000 minus numer ruchu, w którym wygrał. Do oceny przeciwnika dolicza się tę samą wartość, ale przemnożoną przez -1. W przypadku nie zakończenia partii przed upływem limitu posunięć partia podlega ocenie. Uwzględnia się tylko wartości materialne bierek:

- hetman = 900

- wieża = 500
- goniec = 300
- skoczek = 300
- pionek = 100

Następnie różnica wartości materialnych jest oceną obu stron (dla strony słabszej wartość jest ujemna). Np. jeśli białe mają przewagę pionka, to ich ocena w tej partii wynosi 100, a ocena czarnych -100. Ocena danego programu to suma ocen ze wszystkich jego partii.

Był też sprawdzany inny schemat, w którym w momencie oceniania niezakończonych partii strona mająca przewagę dostawała ujemną ocenę za to, że nie wygrała mając przewagę. Jednak metoda ta została odrzucona, ponieważ programy analizując drzewo gry nic nie wiedziały o limicie posunięć.

Widać, że taka funkcja przystosowania wyraźnie preferuje programy, które potrafią zamatować. Natomiast ocena partii niezakończonych jest co najmniej o rząd wielkości mniejsza.

3.3.3 Selekcja

Po rozegraniu turnieju można przystąpić do tworzenia nowej populacji. W tradycyjnym algorytmie genetycznym mamy dwukrotnie do czynienia z selekcją. Po raz pierwszy dokonujemy selekcji przy wybieraniu osobników ze zbioru „rodziców", aby skojarzyć ich w pary, skrzyżować i dokonać mutacji, tworząc w ten sposób zbiór „potomków". Następnie po raz drugi dokonujemy selekcji tworząc nową populację (ze zbioru „rodziców" i „potomków"). Aby stosować taki schemat musimy znać wartość funkcji przystosowania dla „potomków". W przypadku uczenia programów szachowych obliczenie wartości funkcji przystosowania jest bardzo kosztowne. Dlatego algorytm genetyczny został zmodyfikowany pod kątem problemów z kosztowną funkcją przystosowania.

Główna idea polega na tym, że rezygnujemy z drugiej fazy selekcji. Zamiast tego w pierwszej fazie selekcji wybieramy bezwarunkowo dwa najsilniejsze programy oraz pozostałe $N-2$ programy stosując ruletkę. Wyróżnione dwa programy są traktowane w sposób szczególny — nie stosujemy na nich operatorów krzyżowania i mutacji, ponieważ mogą to być najlepsze programy ze zbioru wszystkich możliwych (elitaryzm). Dwa z uwagi na brak przechodniości — pojęcie lepszy nie jest jednoznacznie zdefiniowane. Pozostałe $N-2$

programy wybieramy ze zbioru „rodziców" losując z powtórzeniami z rozkładem prawdopodobieństwa:

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$$

(jest to tak zwana reguła ruletki). W szczególności, w tej grupie mogą znaleźć się dwa najsilniejsze programy (lub jeden z nich — nawet $N-2$ razy, jeżeli jest dużo lepszy od pozostałych). Grupa ta będzie następnie podlegała krzyżowaniu i mutacji. Po zastosowaniu operatora krzyżowania i mutacji (opisanych niżej) dostajemy nową populację: dwa niezmodyfikowane programy (kopia ze zbioru „rodziców") oraz $N-2$ programy zmodyfikowane przez operatory genetyczne.

3.3.4 Krzyżowanie

Programy z drugiej grupy kojarzymy w pary zgodnie z kolejnością wylosowania: 3 z 4, 5 z 6, ..., $N-1$ z N. Następnie każda para podlega krzyżowaniu z prawdopodobieństwem p_c (równym 0.9). Krzyżowanie polega na wylosowaniu punktu przecięcia i zamianie odpowiednich fragmentów:

$$(a_1, a_2, \ldots, a_{i-1} \mid a_i, \ldots, a_{24}) \to (a_1, a_2, \ldots, a_{i-1}, \tfrac{a_i+b_i}{2}, \ldots, \tfrac{a_{24}+b_{24}}{2})$$
$$(b_1, b_2, \ldots, b_{i-1} \mid b_i, \ldots, b_{24}) \to (\tfrac{a_1+b_1}{2}, \ldots, \tfrac{a_{i-1}+b_{i-1}}{2}, b_i, \ldots, b_{24})$$

Zatem mamy tu do czynienia z krzyżowaniem arytmetycznym opisanym w [17]. Jedyna różnica polega na tym, że tam po skrzyżowaniu dwóch pojedynczych parametrów x i y otrzymywaliśmy dwie wartości $ax + (1-a)y$ oraz $(1-a)x + ay$ (dla pewnego a z przedziału $(0,1)$), a tutaj dostajemy jedną wartość $\frac{x+y}{2}$. Dzielenie przez 2 należy rozumieć jako DIV 2, ponieważ parametry są liczbami całkowitymi.

3.3.5 Mutacja

Mutacji podlega jedynie $N-2$ osobników. Podczas mutacji następuje zmiana reprezentacji parametrów z INTEGER na ciąg bitów. Następnie parametry podlegają kodowaniu Gray'a [9]. Ideą tego kodowania jest to, aby sąsiednie liczby miały kody różniące się dokładnie jednym bitem. Np. binarne kody zbioru $\{0, 1, \ldots, 7\}$ to zbiór $\{000, 001, 010, 011, 100, 101, 110, 111\}$, podczas, gdy odpowiadające im kody Gray'a to $\{000, 001, 011, 010, 110, 111, 101, 100\}$. Dzięki takiemu kodowaniu mutacja może zmienić parametr zarówno o $+1$ jak i o -1 (przy mutacji na jednej pozycji, która zdarza się najczęściej). Zatem mamy pełną symetrię. W przypadku mutacji na większej liczbie pozycji możemy dostać dowolną wartość. Procedury konwertujące są następujące:

$$\begin{aligned}\mathsf{Gray}(j) &= j \text{ XOR } (j \text{ DIV } 2)\\ \mathsf{Gray}^{-1}(j) &= \text{IF } (j=0) \text{ OR } (j=1) \text{ THEN } j\\ &\quad \text{ELSE } j \text{ XOR } \mathsf{Gray}^{-1}(j \text{ DIV } 2)\end{aligned}$$

Po zamianie na kody Gray'a każdy bit każdego parametru jest zamieniany na przeciwny z prawdopodobieństwem p_m (zwykle 0.05 lub 0.01). Po mutacjach na poszczególnych bitach, każdy parametr jest zamieniany ponownie na liczbę typu INTEGER. Jeśli należy do dziedziny, to znaczy, że mutacja poskutkowała. W przeciwnym przypadku mutacja była tak silna, że parametr wyszedł poza dopuszczalny przedział. Wtedy losujemy dowolną liczbę z dziedziny danego parametru (każdą z jednakowym prawdopodobieństwem).

W tej wersji algorytmu genetycznego są użyte dwie różne reprezentacje osobników: do krzyżowania reprezentacja numeryczna, do mutacji — binarna. Wydaje mi się, że w ten sposób można wykorzystać zalety obu operatorów: krzyżowania arytmetycznego oraz mutacji na kodach Gray'a.

3.3.6 Przebieg ewolucji

Specyficzną cechą rozwiązywanego tu zadania jest to, że funkcja przystosowania jest bardzo kosztowna. Jeśli rozegranie jednej partii trwa około 5 minut, to jeden krok algorytmu genetycznego trwa $\frac{N(N-1)}{2} * 5$ minut. Przykładowo dla $N = 8$ daje to 2 godziny i 20 minut, zatem w ciągu nocy mogą być wykonane maksymalnie 4 kroki algorytmu genetycznego. Jasne jest, że podczas całego procesu uczenia nie można liczyć na wykonanie więcej niż 1000 iteracji, choć i ta liczba wydaje się zawyżona. Sytuację może poprawić fakt, że czas trwania pojedynczej partii jest tym krótszy, im płytsza jest głębokość analizy. Jednocześnie rośnie wtedy znaczenie funkcji oceniającej. Zatem w początkowej fazie uczenia głębokość może być mała (np. $2+1$ tzn. głębokość pełnego przeszukiwania = 2, wariant forsowny = 1), natomiast w późniejszej fazie głębokość można zwiększyć do $(2+2)$, $(3+1)$, $(3+2)$, $(3+3)$, $(4+1)$ lub $(4+2)$.

Przy tym problemie przydaje się szybka zbieżność algorytmu genetycznego do suboptymalnego rozwiązania. Dzięki temu szybko dostajemy akceptowalne rozwiązanie i wtedy, w zależności od ilości czasu i mocy obliczeniowej, możemy kontynuować bądź przerwać obliczenia.

3.4 Opis implementacji algorytmu genetycznego

Po tym, co do tej pory zostało powiedziane, pozostaje jeszcze podać zmodyfikowany algorytm genetyczny — w takiej postaci był przeze mnie używany. Jest on opisany przy pomocy abstrakcyjnej notacji, z którą zetknęliśmy się już wcześniej.

Osobniki są kodowane jako wektory 24 liczb całkowitych typu INTEGER. Dziedziny są ustalane przed uczeniem, aczkolwiek w trakcie działania algorytmu mogą być zmieniane — wymaga to ręcznej zmiany niektórych parametrów, jeśli po zmianie nie należą do dziedziny.

Podczas ewolucji algorytmu genetycznego są stosowane dwie reprezentacje:

- jedna podczas krzyżowania — kodowanie w postacji 16-bitowych liczb całkowitych,
- druga podczas mutacji — reprezentacja w postaci 24 ciągów binarnych, po 16 bitów na jeden parametr.

```
PROCEDURE ModifiedGeneticAlgorithm;
BEGIN
   t:=0;
   pop_size:=8;
   (* tu może być dowolna, większa stała (najlepiej parzysta) *)
   Initialize P(t);
   (* jeden osobnik zdefiniowany, pozostałe wygenerowane losowo —
   losowane z dziedzin z jednakowym prawdopodobieństwem *)

   Tournament P(t);
   (* rozegranie turnieju „każdy z każdym" *)
   Evaluate P(t);
   (* obliczenie współczynników przystosowania populacji *)

   WHILE NOT StopCondition DO
      t:=t+1;
      Remember2Winners from P(t-1) as W(t);
      (* zapamiętanie dwóch zwycięzców ostatniego turnieju *)

      Select P'(t) from P(t-1);
      (* wybranie pop_size−2 osobników do reprodukcji *)

      Recombine P'(t);
      (* wyprodukowanie nowych osobników —
      krzyżowanie arytmetyczne, mutacja na kodach Gray'a *)

      P(t):=P'(t) + W(t);
      (* stworzenie nowej populacji *)

      Tournament P(t);
      (* rozegranie turnieju „każdy z każdym" *)

      Evaluate P(t);
      (* obliczenie współczynników przystosowania *)
   END;
END;
```

4 Wyniki eksperymentów

Po około dwóch miesiącach (z przerwami) pracy algorytmu genetycznego proces uczenia został zakończony. W tym czasie udało się przeprowadzić niewiele ponad 70 turniejów, co odpowiada rozegraniu około 2 tysięcy partii. Uczenie odbywało się bez udziału człowieka (programy grały same ze sobą), dzięki czemu uczenie można było przeprowadzać w nocy — obowiązki „nauczyciela" — człowieka sprowadzały się tylko do włączania i wyłączania komputera o odpowiedniej porze.

4.1 Przebieg uczenia

Uczenie nie jest całkowicie zakończone. W algorytmie genetycznym nigdy nie wiemy, czy istnieje jeszcze lepsze rozwiązanie (wyjątkiem może być optymalizacja funkcji ograniczonej, dla której algorytm genetyczny znalazł punkt z dziedziny, dla którego funkcja celu przyjmuje wartość równą ograniczeniu).

Tym bardziej interesujące jest, do jakiego rozwiązania doszliśmy w procesie uczenia. Przypomnijmy, że algorytm genetyczny poruszał się w przestrzeni 24 parametrów, liczba wszystkich możliwych zestawów parametrów to około 10^{45}. Początkowo było nawet rozważane ograniczenie liczby parametrów. Jednak wtedy dramatycznie pogarszała się jakość gry. Przykładowo, po ograniczeniu zestawu parametrów tylko do oceny materialnej funkcja oceniająca przestała rozróżniać pozycje o tym samym stosunku sił materialnych, choć niektóre z nich były wyraźnie lepsze od innych.

Można było nawet dodatkowo dodać pewne parametry, które nie byłyby wykorzystywane przez funkcję oceniającą (pewnego rodzaju redundancja). Wykazanie skuteczności uczenia w takim przypadku pozwoliłoby zdjąć z projektanta funkcji oceniającej obowiązek uwzględniania tylko szachowych pojęć (przypomnijmy definicję funkcji oceniającej przy pomocy momentów względem przekątnych w programie Samuela). Podczas uczenia programów szachowych częściowo mieliśmy do czynienia z taką sytuacją z uwagi na pewne niedoskonałości funkcji oceniającej.

W poniższych dwóch grupach tabel zostały umieszczone zestawy parametrów zwycięzców ostatnich 63 turniejów — w pierwszej grupie w kolejnych wierszach znajdują się parametry o numerach 1–12, w drugiej o numerach 13–24. Numery kolumn oznaczają wartości odpowiednich parametrów i tak (dodatkowo podaję dopuszczalne dziedziny):

1. *Skoczek* [200, 500]
2. *Goniec* [200, 500]

3. *Wieża* [400, 600]
4. *Hetman* [800, 1100]
5. *MateriałWKońcówce* [1000, 2000]
6. *PrzewagaPrzyMatowaniu* [400, 1000]
7. *KrólWCentrum* [0, 100]
8. *PremiaZaBezpiecznegoKróla* [0, 100]
9. *KaraZaBrakRozwoju* [0, 100]
10. *KaraZaIzolowanePionki* [0, 100]
11. *KaraZaZdwojonePionki* [0, 100]
12. *KaraZaD2E2* [60, 300]
13. *KaraZaD2iD3* [0, 150]
14. *KaraZaE2iE3* [0, 150]
15. *PremiaZaZaawansowaniePionka* [0, 100]
16. *KaraZaBrakRozwojuSkoczka* [0, 100]
17. *KaraZaOdległośćSkoczkaOdCentrum* [0, 100]
18. *KaraZaOdległośćSkoczkaOdKróla* [0, 100]
19. *PremiaZaIlośćRuchówGońca* [0, 10]
20. *PremiaZaWieżęNa7* [0, 100]
21. *PremiaZaIlośćRuchówWieży* [0, 10]
22. *KaraZaOdległośćHetmanaOdCentrum* [0, 20]
23. *KaraZaOdległośćHetmanaOdKróla* [0, 20]
24. *PremiaDlaHetmanaZaOsłabienieKrólaPrzeciwnika* [0, 100]

Grupa parametrów o numerach 1–12. W pierwszym wierszu znajdują się numery parametrów, w pierwszej kolumnie znajduje się numer turnieju.

Lp.	1	2	3	4	5	6	7	8	9	10	11	12
1	479	500	534	1065	1000	612	23	40	35	18	28	264
2	395	333	575	900	1682	813	8	92	53	0	25	268
3	395	333	575	900	1682	813	8	92	53	0	25	268
4	395	333	575	900	1682	813	8	92	53	0	25	268
5	482	339	576	973	1580	598	77	82	73	0	78	180
6	459	232	516	1069	1525	569	64	19	49	0	5	281
7	459	232	516	1069	1525	569	64	19	49	0	5	281
8	458	232	516	1042	1925	569	65	20	49	0	5	299
9	459	231	516	1069	1525	959	64	19	49	66	5	217
10	201	231	516	1062	1625	569	64	19	49	0	5	290
11	201	231	532	1062	1625	765	64	19	49	0	5	290
12	201	231	516	1062	1625	569	64	19	49	0	26	290
13	201	231	516	1062	1625	569	64	19	49	0	26	290
14	201	231	512	1062	1799	765	64	19	49	0	5	290
15	201	231	512	1062	1799	765	64	19	49	0	5	290
16	224	244	512	1062	1799	765	72	19	49	0	5	290
17	201	231	512	883	1799	852	64	19	49	0	5	290
18	201	231	512	883	1799	852	64	19	49	0	5	290
19	201	231	512	883	1799	852	64	19	49	0	5	290
20	214	232	513	928	1805	763	50	19	49	36	13	122
21	210	224	544	958	1729	757	52	22	49	9	9	206
22	201	208	572	1017	1568	745	7	39	43	39	8	290
23	329	201	561	1062	1492	710	53	46	52	52	65	291
24	372	270	561	1082	1813	816	7	72	71	42	34	206
25	374	232	561	941	1771	727	30	33	89	38	18	293
26	329	273	561	815	1895	745	68	95	43	56	52	108
27	440	273	566	1072	1892	911	22	34	4	71	58	268
28	440	362	531	861	1683	911	23	72	6	71	58	268
29	440	273	566	1072	1892	911	22	34	4	71	58	268

Lp.	1	2	3	4	5	6	7	8	9	10	11	12
30	294	465	457	958	1557	904	41	34	62	71	42	228
31	440	273	514	1010	1892	880	20	80	22	71	58	268
32	367	438	474	969	1525	919	47	26	48	63	31	248
33	227	344	505	961	1742	907	22	40	33	72	34	289
34	216	425	401	822	1294	919	15	5	58	54	24	248
35	405	384	442	891	1317	631	18	3	73	63	33	289
36	287	366	545	822	1377	802	8	20	41	73	34	223
37	337	378	514	849	1456	830	24	5	42	69	83	224
38	333	412	497	956	1312	914	16	12	93	38	88	220
39	445	434	521	943	1484	646	24	5	62	7	29	179
40	445	434	521	943	1484	646	24	5	62	7	29	179
41	406	357	571	1031	1330	470	46	60	81	7	96	179
42	441	321	478	920	1955	834	40	56	53	12	92	224
43	426	492	560	1031	1795	542	53	14	12	37	63	242
44	391	405	549	1031	1795	542	74	14	19	37	0	242
45	391	405	549	1031	1795	542	74	14	19	37	0	242
46	391	405	549	1031	1795	542	74	14	19	37	0	242
47	273	238	424	1087	1852	542	74	15	19	37	0	291
48	496	371	569	805	1741	437	31	83	19	28	68	117
49	274	251	461	1024	1804	452	27	68	16	4	100	129
50	268	297	461	804	1148	958	27	67	16	74	92	129
51	499	218	403	1012	1652	608	60	56	33	3	8	170
52	456	264	457	859	1698	528	42	74	31	39	7	253
53	354	223	477	987	1695	490	37	3	35	19	16	186
54	365	223	476	987	1695	749	37	43	69	19	16	82
55	387	287	425	1072	1701	465	90	47	31	21	7	254
56	371	428	509	1039	1813	723	47	82	80	13	39	109
57	463	352	469	1016	1757	578	73	67	17	81	59	247
58	400	349	447	1044	1751	740	83	4	17	45	62	77
59	406	443	547	1045	1545	730	65	43	48	41	52	66
60	406	441	408	1045	1545	916	50	90	49	41	52	66
61	373	359	486	974	1109	886	63	36	32	33	7	85
62	389	400	447	1008	1327	901	0	48	64	26	28	75
63	390	427	545	993	1535	863	0	85	64	30	0	248

Grupa parametrów o numerach 13–24.

Lp.	13	14	15	16	17	18	19	20	21	22	23	24
1	127	34	28	98	49	20	0	93	7	18	0	85
2	9	117	29	26	89	8	4	46	2	16	15	72
3	9	117	29	26	89	8	4	46	2	16	15	72
4	9	117	29	26	89	8	4	46	2	16	15	72
5	70	49	84	11	84	24	7	95	10	17	5	23
6	1	120	22	6	2	80	2	27	2	15	5	84
7	1	120	22	6	2	80	2	27	2	15	5	84
8	62	136	18	58	2	80	2	27	2	15	5	84
9	1	120	25	6	13	35	2	27	9	15	5	84
10	31	128	20	6	2	80	2	27	2	0	5	80
11	31	128	90	6	2	80	2	27	2	2	5	80
12	31	128	20	6	2	80	2	120	2	0	5	80
13	31	128	20	6	2	80	2	120	2	0	5	80
14	31	4	55	6	2	80	0	27	2	1	7	80
15	31	4	55	6	2	80	0	27	2	1	7	80
16	31	4	8	6	2	87	0	36	2	14	7	80
17	31	4	22	5	2	77	4	63	2	0	6	80
18	31	4	22	5	2	77	4	63	2	0	6	80
19	31	4	22	5	2	77	4	63	2	0	6	80
20	31	11	20	7	3	75	9	100	5	12	5	80
21	37	63	20	13	5	44	6	78	0	6	5	8
22	53	50	21	31	11	73	7	58	2	13	5	59
23	142	66	41	42	14	69	1	61	7	7	6	69
24	128	72	53	67	50	32	4	10	4	8	6	41
25	31	57	43	69	62	63	9	61	7	7	7	33
26	52	52	21	96	11	78	3	67	2	2	13	58
27	34	35	42	82	33	65	3	94	4	9	11	18
28	34	35	43	82	25	91	9	76	4	1	1	70
29	34	35	42	82	33	65	3	94	4	9	11	18
30	140	55	42	82	33	96	2	89	2	9	8	18
31	91	36	56	82	33	65	3	93	10	9	16	19

Lp.	13	14	15	16	17	18	19	20	21	22	23	24
32	89	116	18	77	39	50	1	75	3	9	10	34
33	87	21	38	52	51	73	5	70	6	9	8	74
34	88	68	18	64	6	55	4	72	4	9	9	100
35	23	26	38	52	8	72	4	26	1	9	8	78
36	74	95	24	59	8	66	4	71	5	5	18	12
37	38	55	51	7	17	68	4	6	4	9	12	21
38	56	101	50	83	15	68	2	6	0	14	13	26
39	52	6	44	19	30	68	4	44	4	4	7	52
40	52	6	44	19	30	68	4	44	4	4	7	52
41	112	28	18	35	29	2	4	44	4	4	7	89
42	146	84	69	6	17	35	10	69	4	7	1	21
43	105	94	35	80	98	68	8	88	9	7	20	78
44	105	21	35	83	10	68	2	88	8	4	5	45
45	105	21	35	83	10	68	2	88	8	4	5	45
46	105	21	35	83	10	68	2	88	8	4	5	45
47	150	21	8	90	10	91	9	100	10	12	13	20
48	85	31	35	46	15	34	4	88	8	5	8	45
49	103	33	32	82	16	50	10	1	8	14	5	8
50	94	33	32	93	16	16	10	30	8	14	7	63
51	45	72	48	86	26	25	6	82	6	4	15	71
52	33	95	60	7	36	71	6	50	1	8	8	40
53	89	87	69	20	25	82	9	30	2	0	15	86
54	101	115	15	41	30	42	5	41	8	0	12	65
55	113	37	60	62	35	10	7	86	2	1	9	45
56	85	87	89	28	40	16	10	71	2	17	15	84
57	115	41	68	12	28	29	4	58	9	9	9	18
58	87	78	33	25	25	21	3	18	10	0	9	44
59	4	98	61	26	31	18	9	41	2	17	11	84
60	27	66	61	26	31	18	9	1	2	17	11	36
61	10	122	14	38	69	28	9	53	3	4	8	67
62	29	81	37	32	50	23	9	27	9	10	18	31
63	88	88	24	69	1	24	10	68	3	13	7	8

Dla uzupełnienia powyższych danych podaję w drugim wierszu poniższych dwóch tabel parametry zdefiniowane przeze mnie jeszcze przed rozpoczęciem procesu uczenia (p_0). W trzecim wierszu znajdują się parametry zwycięzcy ostatniego turnieju (p_{63}), w czwartym średnia 10 ostatnich zestawów parametrów ($p_{śr}$) oraz — w piątym — odchylenie standardowe od tej średniej (σ_p) zaokrąglone do pełnych jedności.

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i, \quad \sigma^2 = \frac{1}{n(n-1)}\sum_{i=1}^{n}(\bar{x} - x_i)^2, \quad \sigma = \sqrt{\sigma^2}$$

Lp.	1	2	3	4	5	6	7	8	9	10	11	12
p_0	350	350	500	900	1500	450	20	20	10	20	10	100
p_{63}	390	427	545	993	1535	863	0	85	64	30	0	248
$p_{śr}$	395	371	475	1022	1578	755	51	55	47	35	32	131
σ_p	9	23	15	10	69	46	10	8	7	6	8	26

Lp.	13	14	15	16	17	18	19	20	21	22	23	24
p_0	20	20	40	20	10	10	4	40	1	5	10	20
p_{63}	88	88	24	69	1	24	10	68	3	13	7	8
$p_{śr}$	66	81	46	36	34	23	8	46	5	9	11	48
σ_p	14	9	8	6	6	3	1	8	1	2	1	8

Analogicznie, jak w rachunku błędu pomiarowego wprowadzimy pojęcie błędu względnego jako iloraz $\frac{\sigma_p}{p_{śr}}$. Wynik wyrażamy w procentach.

1	2	3	4	5	6	7	8	9	10	11	12
2,3	6,2	3,2	1,0	4,4	6,1	20	15	15	17	25	20

13	14	15	16	17	18	19	20	21	22	23	24
21	11	17	17	18	13	13	17	20	22	9,1	17

Na podstawie odchylenia od średniej można stwierdzić, które parametry są istotne i jakie powinny mieć wartości. Przyjmując próg 15% dla wartości niestabilnych dla pojedynczego parametru, można stwierdzić, że parametry o numerach 7, 8, 9, 10, 11, 12, 13, 15, 16, 17, 20, 21, 22, 24:

- *KrólWCentrum*,
- *PremiaZaBezpiecznegoKróla*,

- *KaraZaBrakRozwoju* ,
- *KaraZaIzolowanePionki* ,
- *KaraZaZdwojonePionki* ,
- *KaraZaD2E2* ,
- *KaraZaD2iD3* ,
- *PremiaZaZaawansowaniePionka* ,
- *KaraZaBrakRozwojuSkoczka* ,
- *KaraZaOdległośćSkoczkaOdCentrum* ,
- *PremiaZaWieżęNa7* ,
- *PremiaZaIlośćRuchówWieży* ,
- *KaraZaOdległośćHetmanaOdCentrum* ,
- *PremiaDlaHetmanaZaOsłabienieKrólaPrzeciwnika*

powinny być dokładniej zdefiniowane, gdyż są zbyt ogólne lub też proces uczenia powinien być kontynuowany, aż do uzyskania lepszej zbieżności (mniejszych współczynników błędu względnego). Natomiast np. dla parametrów: *Skoczek*, *Goniec*, *Wieża* i *Hetman* — można mówić o zbieżności. Przyglądając się bliżej zestawom parametrów zwycięzców poszczególnych turniejów można dostrzec, że wartości (395 ± 9) dla parametru *Skoczek* oraz (371 ± 23) dla parametru *Goniec* zostały znalezione stosunkowo „niedawno" (w sensie ewolucji procesu uczenia). Wcześniej, w okolicy 10 turnieju dominowały wartości 201 dla parametru *Skoczek* oraz 231 dla parametru *Goniec*. Jednocześnie należy dodać, że w wiekszości programów szachowych wartości lekkich figur wynoszą więcej niż 300. Zatem wartości parametrów: *Skoczek*, Goniec, *Wieża* i *Hetman* znalezione przez algorytm genetyczny są poprawne.

4.2 Mecz sprawdzający

W celu sprawdzenia skuteczności uczenia zostało rozegranych kilka partii między zwycięzcą ostatniego turnieju, a programem zdefiniowanym przeze mnie. Zestaw parametrów, który dobrałem, był ustalony kilka miesięcy temu, aby przetestować metodę przeszukiwania drzewa gry. Po zaprogramowaniu

algorytmu genetycznego, naturalne było wykorzystanie tego zestawu do ukierunkowania poszukiwań. Dziś istotne jest, czy „uczeń przerósł mistrza" (a dokładniej „przodka").

Zostało rozegranych 8 partii: po dwie różnymi kolorami przy różnych głębokościach analizy $(2+1)$, $(2+2)$, $(3+1)$ oraz $(3+2)$. Limit posunięć na partię został ustalony na 200, aby oba programy miały szansę zrealizować przewagę, (jeśli taką posiadały). Dla ustalenia uwagi nazwijmy programy: Przodek i Potomek (zwycięzca ostatniego turnieju).

Przy głębokości $(2+1)$ (programy uczono przy głębokości $(2+2)$ z wariantem forsownym — przez co faktyczna głębokość często była większa) Przodek zwyciężył dwukrotnie. Przy głębokości $(2+2)$ zanotowano remis — po jednym zwycięstwie dla każdego (przy czym wygrywały czarne). Przy głębokości $(3+1)$ Potomek z kolei odniósł dwa zwycięstwa. Wreszcie przy ustawionej głębokości $(3+2)$ partia Potomek–Przodek zakończyła się remisem, natomiast partia Przodek–Potomek zakończyła się zwycięstwem Przodka.

Niezależnie od powyższych wyników można stwierdzić, że automatyczne wyprodukowanie Potomka, który może grać z Przodkiem, jak równy z równym jest już sukcesem. Oprócz tego, z uwagi na brak przechodniości, bardziej miarodajne może być rozegranie turnieju ze zwycięzcami kilku ostatnich turniejów (rozdział 4.4) lub też wybranie za Potomka średniej z ostatnich 10 zwycięzców (rozdział 4.3). Jeśli okaże się, że gra jest na zbyt niskim poziomie, to zawsze można uruchomić algorytm genetyczny i pouczyć programy. O ile tylko istnieje lepsze rozwiązanie, to na pewno ma szansę być znalezione. Jedynym ograniczeniem na jakość gry może być ograniczoność samego modułu przeszukującego drzewo gry i uproszczona postać funkcji oceniającej. Po zaimplementowaniu tablic transpozycyjnych, iteracyjnego pogłębiania, tablicy ruchów morderców i myślenia na czasie przeciwnika można w ten sposób stworzyć naprawdę silny program. Wtedy wykonywane obliczenia podczas wybierania jednego posunięcia mogłyby być wykorzystane podczas analizy kolejnego. Nie jest to jednak celem przewodnim niniejszej pracy.

4.3 Mecz ze „średnią"

Aby Potomek był bardziej reprezentatywny, zamiast zwycięzcy ostatniego turnieju, wybrałem zestaw parametrów będący średnią ostatnich 10 zwycięzców. Zostały rozegrane 2 partie przy głębokości analizy $(3+1)$ z tak utworzonym programem nazywanym Średnia. Wyniki: Średnia–Przodek: 1–0, Przodek–Średnia: 1/2–1/2.

4.4 Superturniej

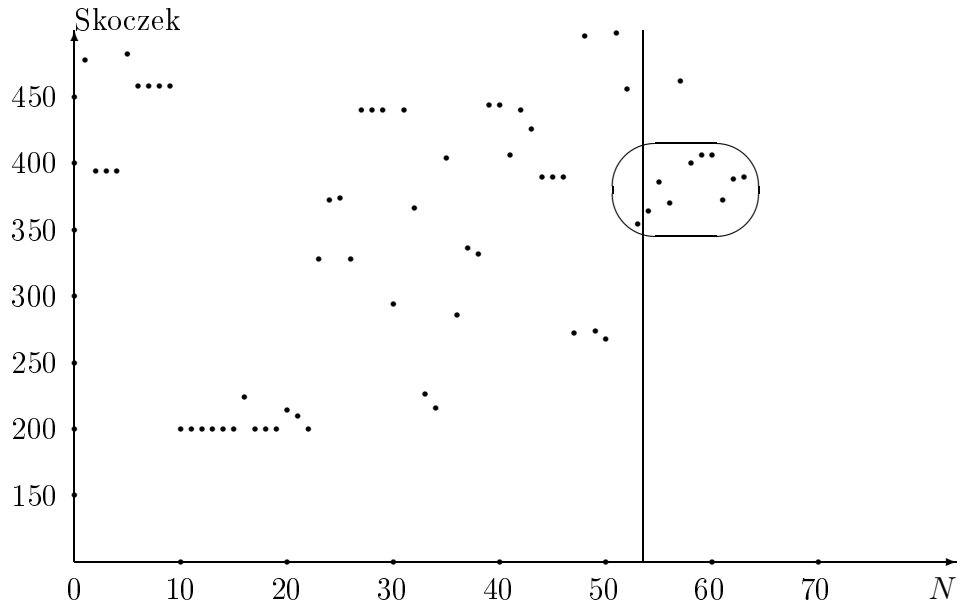
Jako ostatni został rozegrany turniej, w którym startowały programy Przodek i Średnia oraz sześciu ostatnich zwycięzców (wraz z ostatnim zwycięzcą turnieju — Potomkiem z rozdziału 4.2). Turniej był rozgrywany przy głębokości $(2+1)$. Wyniki turnieju są następujące:

Przodek	19947	(3 miejsce)
Średnia	39878	(2 miejsce)
1	330	
2	1065	
3	−19994	
4	−61308	
5	40062	(1 miejsce)
6	−19980	(Potomek z rozdziału 4.2)

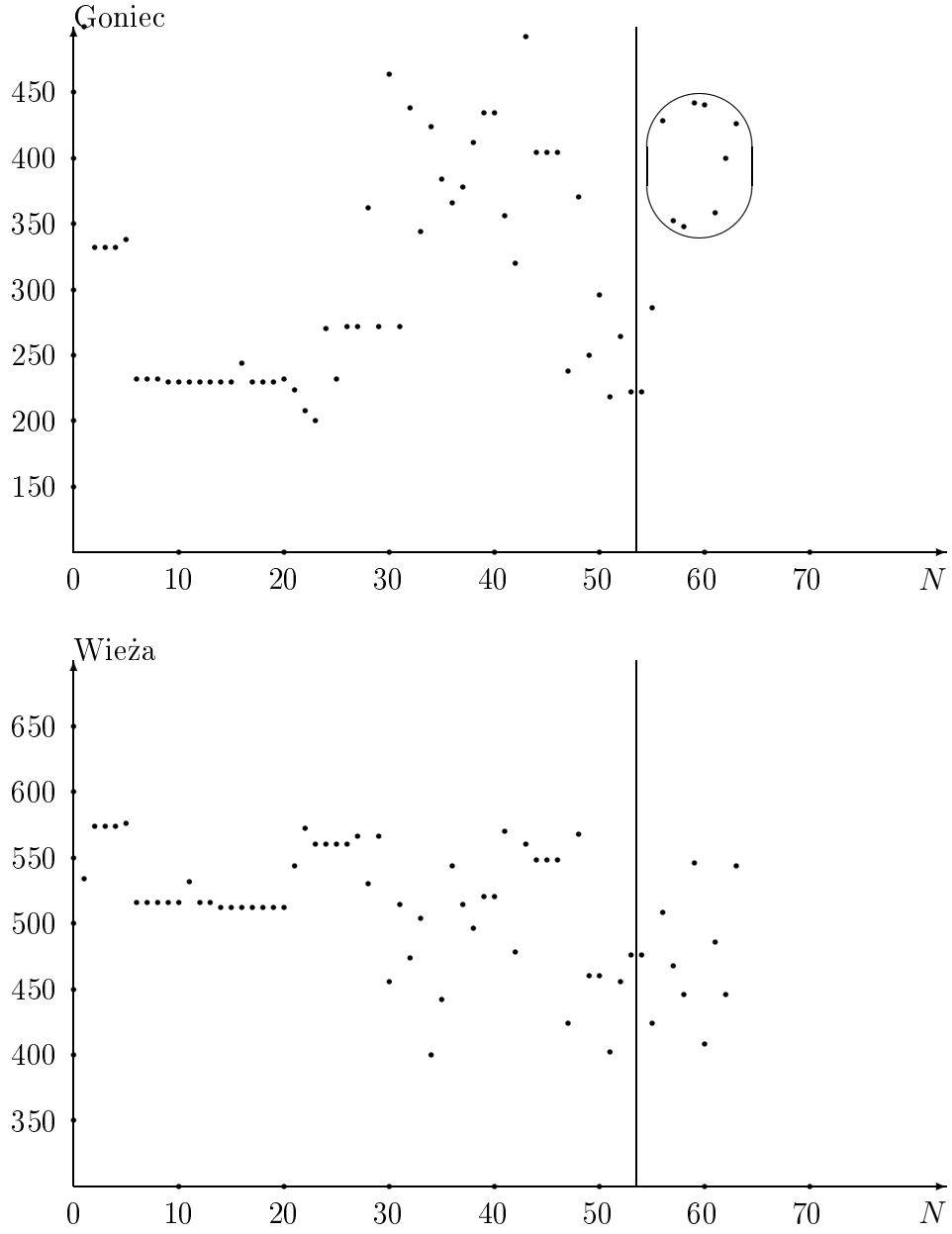
Zatem widać, że choć Przodek jest w czołówce, to jednak są od niego lepsze programy. To częściowo potwierdza skuteczność metody uczenia opartej na algorytmie genetycznym. Utrudnieniem w uczeniu jest to, że Przodek jest programem bliskim optymalnemu (przy takiej funkcji oceniającej i sposobie przeszukiwania drzewa gry). Gdyby porównać program o numerze 5 z losowym programem z początkowej fazy uczenia, to przyrost siły gry byłby widoczny gołym okiem. Takie porównanie ma sens — jeśli się przyjrzeć bliżej danym o zwycięzcach, to nie znajdziemy wśród nich Przodka (nie wygrał żadnego turnieju! — był kiedyś tylko drugi).

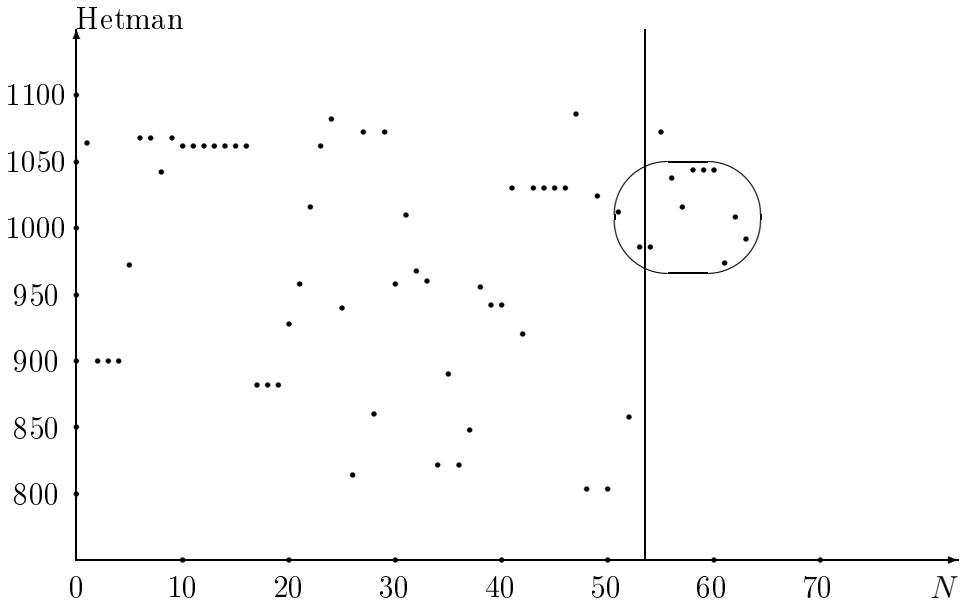
4.5 Wybrane parametry

Na poniższym wykresie jest przedstawiona ewolucja czasowa parametru *S*koczek. Na poziomej osi jest zaznaczony numer kroku N, na pionowej są odkładane wartości parametru dla zwycięzcy N-tego turnieju.



Pionowa linia na powyższym wykresie odgranicza ostatnie 10 turniejów. W owalu znajdują się punkty, w których widać pozytywną tendencję poszukiwań. Nie można jednak użyć tu słowa zbieżność — liczba iteracji była zbyt mała, aby można było wykryć jakąś prawidłowość. Dla uzupełnienia powyższego wykresu podaję podobne wykresy jeszcze dla trzech parametrów: *Goniec*, *Wieża* i *Hetman*.


Goniec
450
400
350
300
250
200
150
0
10
20
30
40
50
60
70
N
Wieża
650
600
550
500
450
400
350
0
10
20
30
40
50
60
70
N



Zatem widać, że dla parametrów *Skoczek* i *Hetman* można zaobserwować „pozytywną tendencję". Trochę gorzej jest dla *Gońca*, natomiast beznadziejnie dla *Wieży*. Wyjaśnić to można tylko przy użyciu wiedzy szachowej. Otóż, wieża jest figurą, którą używa się praktycznie w końcówce. Zatem nieprawidłowe jej ocenianie nie prowadzi do natychmiastowych skutków. Często też program, który lepiej od innych ocenia wieżę, a gorzej inne bierki, nie wykorzysta tej przewagi, ponieważ przegra partię już wcześniej.

Trochę podobna sytuacja jest z gońcem. Natomiast skoczek i hetman są figurami, które są użyteczne od samego początku. Programy, które nauczą się nimi właściwie posługiwać, nie będą musiały w ogóle używać wież. To tłumaczy kształty owali zamieszczonych na powyższych wykresach. Wynika stąd jednocześnie, że próg 15% dla błędu względnego jest zbyt wysoki — bardziej sensowne jest wybranie go np. na poziomie 5%.

5 Heurystyki

5.1 Siła heurystyk

> „Heurystyka (heurystyczna reguła, heurystyczna metoda) jest to ogólna zasada oparta na doświadczeniu, strategia, uproszczenie lub jakiś inny sposób, który drastycznie ogranicza poszukiwanie rozwiązań w dużych obszarach problemowych. Heurystyka nie gwarantuje optymalnych rozwiązań, co więcej, nie gwarantuje ona w ogóle żadnego rozwiązania; wszystko, co można powiedzieć na temat użyteczności heurystyki jest to, że daje ona rozwiązania prawie zawsze wystarczająco dobre." [6]

Wydaje się, że niezależnie od szybkości komputerów i skuteczności metod uczenia ostateczny sukces będzie można osiągnąć dopiero wtedy, gdy wymyślimy (my, jako ludzie) metody wykorzystania silnych heurystyk, którymi posługuje się szachista-człowiek. Okazuje się bowiem, że metody oparte na tzw. „brutalnej sile", jak na razie nie pozwoliły pokonać mistrza świata w szachach. A przecież jest wiele innych problemów (o ile grę w szachy można nazwać problemem), w których jesteśmy jeszcze dalej od dostatecznie dobrego rozwiązania. Przy grze w szachy użycie heurystyk jest najbardziej naturalnym podejściem. Mamy przecież do dyspozycji setki, a może tysiące reguł heurystycznych zawartych w publikacjach szachowych. Skoro funkcja oceniająca jest oparta na wiedzy heurystycznej, to dlaczego nie można wykorzystać wiedzy heurystycznej do selektywnego przeszukiwania drzewa gry zbliżając tym samym sposób analizy do tego stosowanego przez człowieka.

Przypuśćmy, że dane jest drzewo gry, które program szachowy bada do głębokości n, i niech z każdego węzła tego drzewa wychodzą cztery gałęzie. Gdyby można było zwiększyć dwa razy szybkość działania programu, tzn. dwa razy więcej pozycji byłoby rozpatrywanych przez program w tym samym czasie, to głębokość analizy zwiększyłaby się tylko do $(n + \frac{1}{2})$. Jeśli, z drugiej strony, można byłoby zwiększyć dwa razy selektywność wyboru posunięć, tzn. rozpatrywać w każdym węźle tylko dwie z czterech gałęzi, to głębokość analizy zwiększyłaby się dwa razy (do $2n$). Moglibyśmy oczywiście pozwolić sobie na poświęcenie dość dużej, dodatkowej ilości czasu dla każdej rozpatrywanej pozycji, aby można było osiągnąć taki wzrost selektywności.

Zalety selektywnego przeszukiwania drzewa gry są jasne. Problem pojawia się jedynie przy pytaniu: które ruchy analizować?

W 1958 roku, po dwóch latach pracy, Alex Bernstein, szachista i programista firmy IBM napisał program oparty na powyższej idei. Program badał kontynuacje na dwa posunięcia w przód (ówczesne komputery nie pozwalały na więcej). Na każdym etapie były stosowane generatory sensownych

posunięć rozpatrując nie więcej niż 7 bezpośrednich kontynuacji. Program wykorzystywał procedurę minimaksową i wybierał posunięcie o największej efektywnej ocenie [3].

Bernstein stworzył program, w którym generowanie posunięć oparte było na regułach. Generator posunięć generował posunięcia prowadzące do osiągnięcia konkretnych celów. Na generatorze posunięć spoczywał obowiązek znalezienia uzasadnienia potrzeby danego posunięcia. Np. posunięcie d2–d4 w debiucie mogło być zaproponowane, bo sprzyjało osiągnięciu celu pod nazwą opanowanie centrum; osiągnięcie celu równowaga materialna powodowało zaproponowanie wycofania figury znajdującej się pod biciem.

Myślę, że Bernstein pokazał właściwą drogę, którą powinny pójść szachy komputerowe. Ostatnie prace w tej dziedzinie oparte były na pełnym przeszukiwaniu drzewa gry (metoda „brutalnej siły"). Główny nacisk położony był na przyśpieszenie programów przez pisanie w assemblerze i budowanie specjalistycznych układów scalonych. Praca Bernsteina była niedoceniona, gdyż jego program nie odnosił takich zwycięstw, jak np. program Samuela.

Przyjrzyjmy się bliżej selektywnej analizie. Praca generatora posunięć programu dokonującego selektywnej analizy sprowadza się do rozpoznawania pewnych cech pozycji (przesłanek, np. debiut, figura pod biciem, zderoszowanie króla, opanowanie centrum, zajęcie linii) oraz odpowiedniej reakcji na powyższe przesłanki. Generator posunięć może być napisany na zasadzie reguł postaci: **if występuje_X then reakcja_X**. Zbiór takich reguł to właśnie wiedza heurystyczna. Im bogatszy, tym lepsza gra programu i większy pożytek z tysięcy rozegranych już partii szachowych. Występowanie warunku w regule (przesłanki) nasuwa analogię do problemu rozpoznawania. W następnym rozdziale zobaczymy, jak można zdefiniować warunek: **występuje_DEBIUT** przy użyciu sieci neuronowej.

Wydaje mi się, że uczenie programów szachowych należy rozważać w połączeniu z biblioteką heurystyk zapisanych w postaci reguł „ **if ... then ...**". Wtedy metoda uczenia opisana w poprzednich rozdziałach mogłaby być zastosowana do łączenia w pary funkcji typu **występuje_X** z procedurami **reakcja_X**.

5.2 System klasyfikujący

Def.:
System klasyfikujący jest to funkcja

$$\varphi : D^n \mapsto \{1, \ldots, k\} \ (D = R \text{ lub } D = \{0, 2\},\ k \geqslant 2),$$

która każdemu wektorowi wejściowemu przyporządkowuje numer klasy abstrakcji, do której ten wektor należy.

W ten sposób niejawnie tworzymy relację równoważności, w której w relacji są pozycje podobne.

Można rozważać uogólnienie tak pojmowanego systemu klasyfikującego w postaci $\Phi : D^n \mapsto R^k$, w której wyjście interpretujemy jako wektor miar przynależności do poszczególnych klas. W szczególności j takie, że $y_j = \max_{1 \leqslant j \leqslant k}\{y_j\}$ ($y_j = j$-ta współrzędna wyjścia) jest numerem klasy, do której należy zaklasyfikować prezentowany wektor.

W przypadku klasyfikowania pozycji szachowych przestrzeń D^n ($n = 64$) wektorów podlegających klasyfikowaniu jest bardzo duża. Jest to ponad 10^{40} różnych pozycji, jakie mogą powstać na szachownicy (przynajmniej teoretycznie — nigdy nie zdarzyło się jeszcze, żeby ktoś miał 9 hetmanów, choć jest to możliwe).

Na klasyfikowanie można też patrzeć, jak na problem wykrywania skupisk. Wyobraźmy sobie przestrzeń, w której zaznaczone (wyróżnione) jest n różnych punktów (n duże, rzędu kilku tysięcy). Oprócz tego mamy do dyspozycji k ($k \ll n$) żetonów, które możemy dowolnie umieścić w naszej przestrzeni. Dla danego, konkretnego rozmieszczenia żetonów mówimy, że punkt przestrzeni x_i ($i = 1, 2, \ldots, n$) należy do klasy j ($j = 1, 2, \ldots, k$) wtedy i tylko wtedy, gdy żeton o numerze j leży najbliżej punktu x_i (jeśli jest kilka takich żetonów, to wybieramy żeton o najmniejszym indeksie). Zatem każdy punkt x_i ma jednoznacznie przyporządkowany żeton. Rozwiązanie problemu klasyfikowania polega na takim rozmieszczeniu żetonów, aby suma odległości punktów x_i od przyporządkowanych im żetonów była stosunkowo mała (niekoniecznie minimalna). Więcej informacji na temat klasyfikowania rozumianego jako wykrywanie skupisk można znaleźć w [12].

5.2.1 Sieć Kohonena jako system klasyfikujący

Sieć Kohonena uczy się odwzorowania $\Phi : D^n \mapsto R^k$ ($k \geqslant 2$) (zatem pasuje do uogólnionej definicji systemu klasyfikującego). Jest to sieć jednowarstwowa o połączeniach jednokierunkowych uczona bez nauczyciela. Dla danego prezentowanego wzorca x wybierany jest neuron zwycięzca, czyli ten, którego wyjście $y_j = x \bullet w_j$ ($x, w_j \in R^n, j = 1, 2, \ldots, k$) jest największe. Najczęściej wejście x i wszystkie wektory w_j pamiętane przez neurony są unormowane. Warunek ten jest równoważny poszukiwaniu neuronu o najmniejszej długości różnicy $\|x - w_j\|$ (lub też, innymi słowy, szukamy takiego w_j, aby kąt pomiędzy x i w_j był minimalny). Tylko ten neuron podlega uczeniu:

$$w_j(i+1) = w_j(i) + \eta_j(i) \cdot (x - w_j(i)),$$

(przy współczynniku uczenia występuje indeks j, ponieważ współczynnik ten może być różny dla różnych neuronów). Po takiej modyfikacji okazuje się,

że w_j jest jeszcze bardziej zbliżony do x. Aby zagwarantować unormowanie wszystkich w_j można dokonywać normalizacji po każdej modyfikacji w_j: $w_j = \frac{w_j}{\|w_j\|}$ [13, 22].

Gdybyśmy mieli tylko jeden neuron, to można pokazać, że zbiega on do średniej arytmetycznej prezentowanych wektorów wejściowych (szkic dowodu tej własności znajduje się w Dodatku A). Jeśli mamy więcej wektorów, to każdy neuron (o ile uczestniczył w procesie uczenia — a więc wykazał największe pobudzenie dostatecznie wiele razy) zbiega do średniej arytmetycznej tych wektorów zbioru uczącego, które zostały zaklasyfikowane do klasy przez niego reprezentowanej, a dokładniej do średniej arytmetycznej tych wektorów, dla których podlegał uczeniu, (chodzi o to, że na początku procesu uczenia neuron może wykazać największe pobudzenie dla wektora, który później zostanie rozpoznawany przez inny neuron).

Mamy więc sieć, która zwraca wektor k liczb rzeczywistych (wyjście wszystkich neuronów) oraz potrafimy obliczyć numer klasy, do której należy zaklasyfikować dany wektor. Poza tym, każdy neuron, (który podlegał uczeniu dostatecznie długo) reprezentuje uogólniony wzorzec danej klasy.

Często uczeniu podlega nie jeden neuron, lecz pewne jego otoczenie. Dzięki temu można pobudzić mało aktywne neurony i zapobiec powstawaniu obumarłych neuronów (tzn. takich, które nic nie rozpoznają).

Jeśli chodzi o $\eta_j(i)$ $(j = 1, 2, \ldots, k)$, to powinna ona spełniać trzy warunki:

$$\forall i \in N : \eta_j(i) > 0,\ \lim_{i\to\infty} \eta_j(i) = 0,\ \sum_{i=1}^{\infty} \eta_j(i) = \infty.$$

Np. $\eta_j(i) = \frac{1}{i+2}$

5.2.2 Opis implementacji sieci Kohonena

Używałem sieci o liczbie neuronów $k = 180$. Nie wiem dokładnie, jakie powinno być k. Pochopne jest stwierdzenie, że jak największe. Nawet, jeśli dysponujemy szybkim komputerem, dobranie zbyt dużego k może prowadzić do zwykłego „wkuwania" na pamięć, a nie do „inteligentnego" uogólniania.

Wektor wejściowy składał się z 64 współrzędnych. Pola szachownicy numerowałem od dołu do góry, a w wierszach od lewej do prawej (a1 — pierwsze pole, h8 — ostatnie). Współrzędne wektora wejściowego przed unormowaniem, to jedna z 13 wartości: biały pion = 1, biały skoczek = 2, biały goniec = 4, biała wieża = 8, biały hetman = 16, biały król = 32; czarne bierki tak samo tylko z minusami, natomiast 0 oznacza puste pole. Dlaczego takie, a nie inne wagi trudno mi powiedzieć. Być może okaże się, że lepsze wartości dają wartości $1, 2, 3, 4, 5, 6$ lub inne. Na przykład przy wagach będących potęgami

dwójki, okazuje się, że przestawienie króla o jedno pole powoduje najczęściej zaklasyfikowanie pozycji do innej klasy, czego nie można powiedzieć np. o pionku.

Z każdym neuronem pamiętam liczbę całkowitą IQ[j], na której zapisuję liczbę zwycięstw danego neuronu — stąd nazwa (współczynnik inteligencji). Wartość tą zwiększam o 1 w każdej iteracji procesu uczenia dokładnie dla jednego neuronu (dla zwycięzcy).

Podczas uczenia modyfikowane są wagi neuronu zwycięzcy oraz dwóch jego sąsiadów. Przy każdej modyfikacji wag j-tego neuronu współczynnik uczenia wynosi $\eta_j(i) = \frac{1}{\mathrm{IQ}[j]+2}$, przy czym, jeżeli j-ty neuron jest zwycięzcą a nie sąsiadem, to jego współczynnik IQ[j] jest zwiększany o 1. Taki dobór współczynnika uczenia wymaga szerszego skomentowania. Po pierwsze widać, że wszystkie $\eta_j(i)$ są nierosnące, po drugie, każdy neuron ma swój własny współczynnik, a nie jak w oryginalnej sieci, gdzie jest jeden współczynnik stale malejący. Dlaczego neuron, który zareaguje na wzorzec, który jest do niego podobny, lecz był później zaprezentowany, ma mieć inny współczynnik, niż jego kolega, który miał więcej szczęścia i wcześniej został zwycięzcą.

Wraz ze zwycięzcą uczeni są niekiedy dwaj jego sąsiedzi, ale IQ[j] zmienia się tylko dla zwycięzcy. To też wydaje się logiczne. Dla zwycięzcy rośnie IQ[j], a tym samym maleje η_j. Chcemy przecież coraz lepszego dopasowania wag. Natomiast sąsiedzi (zakładam, że neurony są ułożone w pierścień tzn. neuron 1-szy i k-ty są też swoimi sąsiadami i każdy neuron ma dwóch sąsiadów) podlegają uczeniu tylko, jeśli ich IQ jest mniejsze niż IQ zwycięzcy (czyli są mniej „zdolne") oraz ich IQ jest mniejsze od pewnego parametru MŁODY. Parametr MŁODY wynosi ok. 100. Jeśli sąsiad nie był jeszcze 100 razy zwycięzcą, to uznajemy go za podatnego na wskazówki zdolniejszych i uczymy go razem ze zwycięzcą, nie zmieniając jednakże jego IQ, w przeciwnym przypadku uznajemy go za osobnika z nawykami, których mu się już nie wykorzeni i pozwalamy mu nie reagować na wskazówki zdolniejszego sąsiada zwycięzcy (nie uczymy go).

Porównywanie IQ zwycięzcy i sąsiadów ma następujące wytłumaczenie: jeśli sąsiad zwycięzcy ma mniejsze IQ, to jest uczony, aby odciążyć zwycięzcę i przejąć wiedzę od zdolniejszego (w ten sposób można pobudzać tzw. neurony obumarłe — takie, które nigdy nie zostają zwycięzcami). Natomiast, jeśli sąsiad zwycięzcy ma większe IQ, to nie podlega on uczeniu, aby nie zapomniał tego, czego się wcześniej nauczył i aby neuron mniej zdolny (a mimo to zwycięzca) zachował wiedzę dla siebie — wygrał, więc są pewne nadzieje. Zatem mamy do czynienia z przekazywaniem wiedzy od zdolniejszych do mniej zdolnych — i tak powinno być — chodzi o równomierne pokrycie przestrzeni wektorami wag neuronów. Jasne się teraz wydaje, dlaczego są-

siedzi zwycięzcy — jeśli są uczeni — nie mają zmienianego IQ i tym samym η_j. Ponieważ nie byli zwycięzcami, więc ich uczenie bardziej przypomina ściąganie, niż zapamiętywanie ze zrozumieniem (w myśl zasady zwycięzca bierze wszystko), dlatego nie zwiększamy ich IQ, czyli nie zmniejszamy η_j, ponieważ nie nadszedł jeszcze czas na łagodne dostrajanie.

Dzięki takiemu doborowi współczynnika η można uczyć sieć wieloetapowo (nie trzeba pamiętać numeru kroku — jest on pamiętany we współczynnikach IQ[j] dla każdego neuronu niezależnie).

Ze względu na to, że podczas uczenia może wystąpić błąd niedomiaru, przyjmowałem, że wagi mniejsze niż parametr MAŁA_LICZBA ($= 10^{-10}$) są równe 0 (wykorzystując fakt, że wartości niezerowe w pozycji są co do modułu $\geqslant 1$).

5.2.3 Wnioski z przeprowadzonych testów

Testy były przeprowadzane na komputerze 386 DX 40 MHz z koprocesorem.

- Uczenie jest dość szybkie — około kilkadziesiąt pozycji na sekundę, co pozwala przeprowadzić cały proces uczenia sieci w ciągu 2–3 godzin.
- Gdy liczba wektorów wejściowych nie przekracza 80% liczby neuronów, sieć potrafi nauczyć się wzorców na pamięć.
- W zdegenerowanym przypadku: 4 neurony i 2 wzorce — uczą się dwa przeciwległe neurony (np. 1 i 3 lub 2 i 4), przy czym jeden z pozostałych średnio raz był uczony. Okazuje się, że sąsiedztwo „zdolniejszego" neuronu powoduje zamazanie jego pamięci.
- W końcówkach, gdy liczba bierek jest mała, przestawienie dowolnej bierki często prowadzi do zmiany klasy, ponieważ wartości bierek w pozycjach z małą ilością materiału są większe (bo wektory są unormowane). Intuicyjnie mogłoby się wydawać, że mniej bierek na szachownicy, to mniej klas. Tymczasem niekoniecznie tak być musi.

Na podstawie własności o „zbieżności do średniej arytmetycznej" można wymyślić inną metodę uczenia. Krótko uczymy sieć, aby uzyskać odpowiedź na pytanie: do której klasy należy dany wektor wejściowy? Następnie obliczamy średnią arytmetyczną w poszczególnych klasach, inicjujemy nimi wagi neuronów odpowiednich klas (dla pustych klas — neurony z IQ $= 0$ — można przyjąć wartości losowe). Uzyskujemy w ten sposób inny podział wektorów wejściowych na klasy. Znowu liczymy średnią arytmetyczną w klasach, itd., aż nie będzie modyfikacji klas (i tym samym wag neuronów). Przy czym

na początku można wstępnie nie stosować starych metod iteracyjnych, a od razu zastosować powyższy proces z losowymi wagami. Jest to nowa metoda uczenia sieci Kohonena.

Powstaje pytanie, czy zbieżna do tego samego, co oryginalna metoda (z dokładnością do permutacji neuronów) i czy w ogóle zbieżna — chodzi tu o stworzenie klas, a nie obliczenie średniej arytmetycznej w klasach.

Rozpoznawanie debiutu może być sprowadzone do:

występuje_DEBIUT(pozycja):=
(NumerKlasy(pozycja) = NumerKlasy(pozycja_początkowa))

Sprowadziliśmy zatem rozpoznawanie jednej, konkretnej własności pozycji do problemu klasyfikowania pozycji. W przypadku rozpoznawania, czy daną pozycję można zaklasyfikować jako debiut istnieją prostsze metody. Wybór debiutu miał być tylko przykładem, jak użyć system klasyfikujący. W przypadku bardziej skomplikowanych cech należy zastąpić wektor wejściowy (reprezentujący pozycję na szachownicy) innym wektorem zawierającym tylko istotne informacje (i pełne — aby zawsze można było prawidłowo podjąć decyzję).

Przykładowo wektor opisujący istotne informacje potrzebne do rozpoznawania cechy występuje_BEZPIECZNY_KRÓL może wyglądać następująco (LiczbaPrzyległychPionów, LiczbaMożliwychSzachów, IlośćFigurNaSzachownicy). Tutaj podobnie, jak przy definiowaniu parametrów funkcji oceniającej ekspert nie musi definiować, kiedy król jest bezpieczny. Wystarczy, że powie, na co należy zwracać uwagę przy sprawdzaniu zabezpieczenia króla. Resztę zrobi system klasyfikujący i algorytm genetyczny.

Nawet, jeśli okaże się, że pozycje z zabezpieczonym królem należą do dwóch różnych klas k_1 i k_2 (co ekspert łatwo stwierdzi), to zawsze funkcję występuje_BEZPIECZNY_KRÓL można zdefiniować następująco:

występuje_BEZPIECZNY_KRÓL(pozycja):=
((NumerKlasy(pozycja) = k_1) OR (NumerKlasy(pozycja) = k_2))

5.3 Podsumowanie

Wydaje się, że dzisiaj tylko programy uczące się mają szansę dalszego pomyślnego rozwoju. Inne drogi zostały już wystarczająco dobrze przebadane. Na uczenie programów rozwiązujących złożone problemy należy patrzeć w perspektywie wykorzystania wiedzy heurystycznej. Heurystyki pozwalają wykorzystać wiedzę „z zewnątrz" w takiej postaci, w jakiej stosuje ją człowiek.

Większość idei zaczerpniętych z rozwoju programów szachowych można wykorzystać do rozwiązywania innych zadań. Jednym z takich przykładów

może być np. automatyczne dowodzenie twierdzeń matematycznych. Problemy, które się wtedy pojawiają są bardzo zbliżone do tych poznanych przy programowaniu szachów: wybór właściwego lematu, głębokość zagłębiania się w dowody lematów, rodzaj przeszukiwania, wykorzystanie wiedzy eksperta, itp.

A Zbieżność wag neuronu

Dany jest ciąg identycznych zmiennych losowych S_k o wartościach w R^n i rozkładzie dyskretnym na zbiorze $\{a_1, \ldots, a_m\}$, $a_i \in R^n$, $\Pr(S_k = a_i) = \frac{1}{m}$ $(i = 1, 2, \ldots, m;\ k = 1, 2, \ldots)$.

Dany jest także ciąg zmiennych losowych X_k o wartościach w R^n i następujących własnościach:

1. X_0 takie, że każda współrzędna ma rozkład jednostajny na odcinku $[-1, 1]$
2. $X_{k+1} = X_k + \frac{1}{k+2}(S_{k+1} - X_k)$ dla $k \geqslant 0$

Pytanie jest następujące: czy ciąg X_k jest zbieżny stochastycznie i jeżeli tak, to do czego? (Ciąg zmiennych losowych X_k jest zbieżny stochastycznie do stałej c, jeżeli $\forall \varepsilon > 0 \lim_{k\to\infty} \Pr(|X_k - c| > \varepsilon) = 0$).

Policzmy kilka początkowych wyrazów:

$$\begin{aligned} X_1 &= X_0 + \frac{1}{2}(S_1 - X_0) = \frac{X_0 + S_1}{2} \\ X_2 &= X_1 + \frac{1}{3}(S_2 - X_1) = \frac{2}{3}X_1 + \frac{1}{3}S_2 = \frac{X_0 + S_1 + S_2}{3} \\ &\vdots \\ X_k &= \frac{X_0 + S_1 + \cdots + S_k}{k+1} = \frac{X_0}{k+1} + \frac{k}{k+1}\frac{S_1 + \cdots + S_k}{k} \end{aligned}$$

Można to udowodnić indukcyjnie.

Pierwszy składnik zbiega stochastycznie do zera, drugi natomiast do średniej arytmetycznej $\bar{a} = \frac{1}{m}\sum_{i=1}^{m} a_i$ (zgodnie z odpowiednim twierdzeniem rachunku prawdopodobieństwa). W granicy X_0 nie wnosi nic do wyniku, natomiast jest potrzebne, aby na początku procesu uczenia równomiernie pokryć przestrzeń wektorami wag neuronów.

Można się zastanawiać, po co w takim razie cały nieskończony proces iteracyjny, skoro a_i są znane i średnią arytmetyczna można łatwo wyliczyć. Jest w tym trochę racji. Dla jednego neuronu nie ma sensu postępować iteracyjnie, jeśli można postąpić prościej. Sytuacja komplikuje się jednak, gdy mamy więcej niż jeden neuron. Nie można już postąpić „prościej", ponieważ nie wiadomo, jak wektory wejściowe zostaną podzielone na klasy i przyporządkowane poszczególnym neuronom.

Zatem, gdybyśmy wiedzieli, jak podzielić wektory wejściowe na klasy, to wagi neuronów można by dobierać deterministycznie (prosto i szybko), a tak pozostaje tylko proces iteracyjny, po zakończeniu którego, wiemy, jak wektory zostały podzielone na klasy oraz, co reprezentuje wektor wag poszczególnych neuronów, (tych dla których IQ jest dostatecznie duże).

B Załączona dyskietka

Na załączonej dyskietce znajdują się następujące pliki:

- TURNIEJ.EXE — główny program uczący wykorzystujący algorytm genetyczny
- ZMIENNE.DAT i INFO.TXT — pliki danych programu TURNIEJ.EXE
- GRAMY.EXE — program, przy pomocy którego można rozegrać partię z najlepszym programem (korzysta z pliku ZMIENNE.DAT)
- UCZSIE.EXE — program uczący sieć neuronową
- SIEC.DAT i BAZA.EXE — pliki danych programu UCZSIE.EXE
- README.!!! — plik tekstowy zawierający istotne informacje dotyczące instalacji i uruchamiania programów zawartych na dyskietce (nie zawarte tutaj) oraz krótki opis ich przeznaczenia i ograniczeń działania
- FONTY.COM, MENU.EXE, SZACHY.BAT, MENU.DBF — programy i pliki pomocnicze
- W katalogu PROGRAMY znajdują się pliki źródłowe powyższych programów

C Słownik terminów

PUSTY RUCH — zrezygnowanie z ruchu — pozwala wykryć, czy przeciwnik stwarza jakieś groźby, które należy neutralizować; stosowany podczas przeglądania drzewa gry; w prawdziwych szachach zabroniony.

TABLICA RUCHÓW MORDERCÓW — tablica, w której są zapamiętywane ruchy, które doprowadziły do cięcia w alfa-becie; ruchy te, o ile legalne, są wypróbowywane w podobnych pozycjach (będących blisko w drzewie gry).

TABLICA TRANSPOZYCYJNA — tablica, w której przechowuje się wyniki obliczeń (fragment drzewa gry wraz z oceną węzłów), dzięki czemu, po napotkaniu danej pozycji po raz drugi można rozpocząć analizę w miejscu, w którym poprzednio została zakończona.

WIELOMIAN OCENIAJĄCY — wielomian o pewnych współczynnikach i wyrazach rozpoznających pewne własności pozycji zwracający liczbę będącą oceną danej pozycji, szczególna postać to wielomian liniowy.

NEURON — funkcja $f : R^n \mapsto R$ postaci $f(x) = x \bullet w$, gdzie $w \in R^n$ jest ustalonym wektorem, a $\bullet$ oznacza iloczyn skalarny.

Bibliografia

[1] Adelson-Wielskij G., Arłazarow W., Bitman A., Donskoj M., *Maszina igrajet w szachmaty*, Izdatielstwo „Nauka", 1983 r.

[2] Bentley J., *Perełki oprogramowania*, WNT, 1986 r., str. 112–113

[3] Bernstein A., Roberts M., *Computer vs. chess-player*, „Scientific American", June, 1958 r., str. 96–105

[4] Bolz L., Cytowski J., *Metody przeszukiwania heurystycznego*, PWN, 1991 r.

[5] Botwinnik M., *Computers in Chess*, Springer-Verlag, 1984 r.

[6] Feigenbaum E.A., Feldman J., *Maszyny matematyczne i myślenie*, PWN, 1972 r. (tłumaczenie z angielskiego, oryginał z 1963 r.)

[7] Frey P., *Chess Skill in Man and Machine*, Springer-Verlag, 1997 r.

[8] Goldberg D.E., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley Publ. Comp., Inc. 1989 r.

[9] Gray F., *Pulse Code Communication*, U.S.Patent 2 632 058, 17 marca 1953 r.

[10] Holland J., *Adaptation in Natural and Artificial Systems*, MIT Press, 1992 r.

[11] Knuth D.E., Moore R.W., „An Analisys of Alpha-Beta Prunning", Artificial Intelligence, vol. 6, str. 293–326

[12] Kodratoff Y., Michalski R., *Machine Learning*, vol. 3, 1990 r., str. 235–268

[13] Korbicz J., Obuchowicz A., Uciński D., *Sztuczne sieci neuronowe*, Akademicka Oficyna Wydawnicza PLJ, 1994 r.

[14] Kujawski A., *Programowanie gry w szachy*, UW, praca magisterska, 1994 r.

[15] Kwaśnicka H., „Próba naśladowania natury — algorytmy genetyczne w informatyce", Informatyka, nr. 12, 1994 r., str. 12–16

[16] Marsland T.A., Schaeffer J., *Computers, Chess and Cognition*, Springer-Verlag, 1990 r.

[17] Michalewicz Z., *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, 1992 r.

[18] Owen G., *Teoria gier*, PWN, 1975 r.

[19] Samuel A.L., „Some studies in machine learning using the game of checkers", IBM Journal of Research and Development, July, 1959, vol. 3, str. 211–229

[20] Shannon C.E., „Programming a digital computer for playing chess", Philosofical Magazine, vol. 41, str. 356–375

[21] Shapiro S.C., *Encyclopedia of Artificial Intelligence* , Wiley–Interscience, 1987 r., str. 159–171

[22] Tadeusiewicz R., *Sieci Neuronowe*, Akademicka Oficyna Wydawnicza RM, 1993 r.

[23] Turing A.M., „Digital computers applied to games" z pracy *Faster then thought*, B.v.Bowden, 1953 r.