"Programowanie gry w szachy"

Praca magisterska

wykonana pod kierunkiem
prof. Stanisława Waligórskiego

Uniwersytet Warszawski 1994 r.

Adam Kujawski
ul. Broniewskiego 6 m. 154
01-785 Warszawa
tel. 663-35-17

SPIS TREŚCI

1. Wstęp.

W październiku roku 1991 powstała idea stworzenia programu szachowego jako pracy dyplomowej na wydziale MIM Uniwersytetu Warszawskiego. Inicjatywa wyszła od autora tych słów, opieki nad pracami podjął się prof. S. Waligórski.

Dlaczego program szachowy? Autor tych słów jest szachistą należącym do szerokiej czołówki krajowej, mistrzem Warszawy z roku 1992-go. Jako student informatyki dość szybko zainteresowałem się szachami komputerowymi i innymi pokrewnymi tematami. Programowanie gier jest jednym z klasycznych tematów badań nad sztuczną inteligencją a szachy zajmują wśród programowanych gier ważne miejsce. Owo szczególne miejsce zawdzięczają szachy swojej popularności a także względom historycznym (często uznaje się pracę C.Shannona "Programming computer for playing chess" z roku 1950-go za jedną z fundamentalnych prac sztucznej inteligencji). Przez czterdzieści lat programowanie szachów dorobiło się swojej pokaźnej teorii i praktyki. Pierwsza część tej pracy poświęcona jest krótkiemu przedstawieniu tych dokonań .

W wyniku prac prowadzonych między październikiem 1992 a styczniem 1994 powstał program nazwany przeze mnie "Joanna". "Joanna" jest programem napisanym według klasycznych reguł gatunku. Program ten jest jednak niezbyt silny, co jest dosyć zrozumiałe, skoro pisałem go w pojedynkę, w dodatku nie posiadając należytego doświadczenia. Nad najsilniejszymi programami szachowymi pracują dziś duże zespoły programistyczne przez wiele lat. Program "Joanna" opisany jest w drugiej części pracy.

Samodzielnym opracowaniem autora jest zaprezentowana w ostatniej, trzeciej części idea wykorzystania tablic transpozycyjnych do prostego mechanizmu "uczenia się". Stanowi ona rozwinięcie prac autorów znanego programu "BeBe". Idea ta może zostać zastosowana w dowolnych zadaniach związanych z przeszukiwaniem dużej przestrzeni stanów, w których ma sens stosowanie technik tablicy transpozycji i iteracyjnie pogłębianego przeszukiwania.

1.1. O programowaniu gier.

W pracy [17] pogrupowano hasła dotyczące sztucznej inteligencji w dziewiętnaście grup problemowych. Jedną z nich stanowi programowanie gier.

Programowanie gier jako gałąź informatyki pojawiło się w latach pięćdziesiątych dzięki pracom C. Shannona i A. Turinga. Gry takie jak szachy czy warcaby stanowią przykład zadania, które człowiek potrafi wykonywać w sposób prawie bezbłędny, a którego nie da się zaprogramować tak jak np. znajdowania pierwiastków równania kwadratowego czy sortowania tablicy. Miano więc nadzieję, że stworzenie silnego programu szachowego przyczyni się do zrozumienia zasad ludzkiego myślenia (stąd nazwa: sztuczna inteligencja) i pomoże w rozwiązaniu innych podobnych zadań, np. rozumienia języków naturalnych, rozpoznawaniu obrazów itp.

Dzisiaj po ponad czterdziestu latach historii programowania gry w szachy (patrz rozdział 1.2) te pierwotne cele i poglądy można zrewidować. Przez czterdzieści lat nie udało się bowiem nikomu skonstruować programu,

którego zasady działania opierałyby się na zasadach funkcjonowania intelektu gracza-człowieka. Natomiast udało się skonstruować silnie grające programy oparte na matematycznych modelach gry w szachy. Najlepsze na dzisiaj programy szachowe opierają się na zasadzie brutalnej siły (ang. "brute force") tzn. na wykorzystaniu szybkości komputera przy zastosowaniu pewnego prymitywnego algorytmu (patrz rozdz. 3.4). Przy tym wszystkim nastąpił ogromny rozwój wielu dziedzin sztucznej inteligencji, tak że programowanie gier stało się samoistną gałęzią badań .

Na ile jednak programowanie gier jest interesujące z punktu widzenia całości AI ? Odpowiedzi na to pytanie autor nie potrafi udzielić. Poszczególne dziedziny AI zazębiają się ze sobą. Dla zaprogramowania gry celowe lub wręcz niezbędne jest stworzenie modułu przeszukującego, bazy wiedzy szachowej, modułu uczącego się itp. Wszystkie te dziedziny znajdują więc w programowaniu gier pole do ekperymentów.

Rolę szachów wśród programowanych gier należy uznać za wiodącą. Wynika to z ich popularności, prestiżu i względów historycznych. Wypada jednak wspomnieć o miejscu jakie szachy zajmują wśród innych gier, które próbowano zaprogramować. Grami zbliżonymi do szachów są warcaby, GO, kółko i krzyżyk, Reversi. Wszystkie te gry podpadają pod jedną kategorię według teorii gier (patrz rozdział 2.1). Wszystkie te gry programowane są najczęściej z pomocą algorytmów minimaksowych (patrz rozdział 3.2). Jednak każda z nich jest w końcu inną grą, stąd odmiennie dla każdej gry konstruuje się jej funkcję heurystyczną (patrz rozdział 3.1). Nadto zasadniczym wyróżnikiem każdej z gier jest liczba możliwości, które można podjąć przy każdym posunięciu. Dla gier "małych", takich jak warcaby, udało się metodą "brutalnej siły" pokonać człowieka - mistrza świata. Dla gry bogatszej możliwościami - szachów problem ten nadal pozostaje aktualny. Natomiast dla GO, gdzie na każdym posunięciu trzeba wybierać pomiędzy ok. 300-ma możliwościami nie udało się nawet zbliżyć do poziomu gracza-pasjonata.

Gry dwuosobowe, skończone, z pełną informacją, o sumie zerowej stanowią jeden z najprostszych przypadków gry w ogóle. Dla porównania brydż będąc grą bez pełnej informacji jest już znacznie trudniejszy do zaprogramowania .

Wśród dzisiejszych motywacji do programowania szachów można wyróżnić cztery:

(a) Chęć pokonania Mistrza Świata w szachach.
(b) Poszukiwanie nowych rozwiązań w programowaniu szachów i spokrewnionych z tym dziedzinach.
(c) motywacje komercyjne.
(d) motywacje hobbystyczne.

Jak widać zaniechano prób skonstruowania programów przypominających swym działaniem rozumowanie człowieka (w każdym razie autorowi nic nie wiadomo o takich pracach).

Autor niniejszego opracowania przyznaje się do motywacji pośredniej między punktem (b) i (d).

1.2. Historia programów szachowych.

Idea stworzenia sztucznego szachisty sięga XVIII wieku. Niejaki Baron Von Kempelan objeżdżał dwory ówczesnej Europy dając pokazy gry gumowego turka wtopionego w drewnianą szafkę. Automat został po tajemniczym zniknięciu Kempelana oddany do muzeum i okazało się, że turek Kempelana był wewnątrz wydrążony i przystosowany do rozgrywania partii przez człowieka umieszczonego w jego wnętrzu.

Pierwsze próby nie stosujące podobnych "chwytów" pochodzą z końca XIX wieku. W 1890 roku hiszpański elektronik Torres y Quevedo skonstruował elektroniczny mechanizm wielkości szafy rozgrywający końcówkę szachową "Król+Wieża : Król". Późniejsza wersja tego automatu pochodząca z 1914-go roku znajduje się do dzisiaj w muzeum Politechniki Madryckiej i podobno nadal działa.

W 1950 roku opublikowana została przełomowa a zarazem fundamentalna praca: "Programming the computer for playing chess" Clauda Shannona. Zawarte w niej idee wyznaczyły kierunek rozwoju programowania gier dwuosobowych z pełną informacją obowiązujący do dzisiaj. Shannon przedstawił w swej pracy dwa schematy programowania gry w szachy znane dzisiaj jako schemat A oraz schemat B Shannona (patrz rozdział 3.2). Pracę o podobnej treści opublikował w 1953 roku Alan Turing. Znakomita większość wszystkich tworzonych dzisiaj programów szachowych to Schemat A lub B plus pewne dodatkowe techniki (patrz rozdział 3.4 , 3.5).

Przez następnych kilka lat powstało kilka programów dla różnych podproblemów związanych z szachami (dwuchodówki, rozgrywanie różnych końcówek etc.). Pierwszy program szachowy został napisany w 1956 roku przez Bernsteina.

W 1966 odbył się pierwszy oficjalny mecz komputerów szachowych. Uczestniczyły w nim dwa programy: radziecka "Kaissa" zaprogramowana przez grupę programistów z Wydziału ITEP Uniwersytetu Moskiewskiego (Adelson-Velsky, Arlazorov, Donskoy przy współudziale M.M.Botvinnika) oraz program amerykański zaprogramowany pod kierownictwem J.McCarthy'ego przez studentów wydziału MIT z Uniwersytetu w Standford. Mecz zakończył się zwycięstwem ZSRR 3:1 . Obie grupy kontynuowały pracę nad programami szachowymi. W rok później w MIT powstał "Mat Hack Six" (Greenblat, Eastlake, Crocker). Program ten wyposażono w kilka nowych technik, przy tym poziom jego gry pozwalał na pokonanie człowieka-amatora.

W 1968-ym roku amerykański mistrz międzynarodowy David Levy założył się z grupą naukowców, że przez następne dziesięć lat nie przegra meczu z programem szachowym. Zdarzenie te mogłoby być marginalnym, ale właśnie pokonanie m.m. Davida Levy'ego stało się celem społeczności programistów gry w szachy przez następne dwadzieścia lat.

W 1974 roku odbyły się w Sztokholmie pierwsze oficjalne Mistrzostwa Świata Komputerów Szachowych. Wzięło w nich udział trzynaście programów. Zabrakło "Mack Hack Six", była natomiast wciąż rozwijana radziecka "Kaissa". Faworytem imprezy był jednak program "Chess" stworzony przez grupę programistów pod kierownictwem D.J.Slate'a i L.R.Atkina z Uniwersytetu Północnozachodniego . "Chess" regularnie wygrywał przeprowadzane od 1970-go roku Mistrzostwa Ameryki Północnej Komputerów Szachowych. Pierwszy tytuł

Mistrza świata przypadł jednak "Kaissie", gdyż "Chess" przegrał jedną ze swoich partii z mniej znanym "przeciwnikiem".

Rozwijana przez piętnaście lat "Kaissa" była rzeczywiście silnym jak na ówczesne czasy programem osiągając wg. międzynarodowego rankingu szachowego 1700 p. ELO. Stało się jednak oczywiste, że program osiągnął granice swych możliwości i że dla dalszego postępu potrzeba zupełnie nowych idei. M.M.Botvinnik, który początkowo współpracował z autorami "Kaissy" już pod koniec lat sześćdziesiątych poszukiwał takich rozwiązań i rozpoczął prace nad własnym programem "Pionier". Autorzy "Kaissy" opisali swe dokonania w pracy [14].

W trzy lata później "Chess" został drugim Mistrzem Świata pokonując "Kaissę" w bezpośrednim pojedynku. "Chess" był interesującym a przede wszystkim silnym (ok. 2000 ELO), programem. Zastosowano w nim szereg ważnych z punktu rozwoju dziedziny rozwiązań takich jak tablice transpozycyjne (patrz rozdz. 3.5) czy podział drzewa gry na regiony (patrz rozdział 3.4). Praca Slate'a i Atkina [2] wywarła wpływ na wszystkich, którzy w następnych latach zajmowali się pisaniem programów szachowych. Wyznaczyła tym samym pewien obowiązujący kierunek w dziedzinie.

W 1978 roku upływał termin wspomnianego wcześniej zakładu między D. Levym a amerykańskimi naukowcami. "Chess" była w tym momencie najsilniejszą maszyną na świecie, ale ciągle jeszcze zdecydowanie za słabą by walczyć z mistrzem międzynarodowym o rankingu 2350 ELO. Mecz pomiędzy D.Levym a "Chess" zakończył się wynikiem 3.5 - 1.5 na korzyść człowieka. Przewaga Levy'go była ogromna i jedynie z powodu pewnej nonszalancji w grze oddał maszynie 1.5 punktu. OMNI Magazine ufundował wtedy specjalną nagrodę 5.000 dolarów dla pierwszego programu który zdoła pokonać D.Levy'go.

W 1980 roku trzecim Mistrzem Świata Komputerów w Szachach został "Belle" autorstwa Kena Thompsona i Joe Condona. "Belle" wyposażono w specjalne, dedykowane układy elektroniczne pozwalające na przeglądanie drzewa gry z prędkością ok.15 tys. pozycji na sekundę, co było wartością znacznie wyższą od dotychczas spotykanych. Użyty w programie algorytm był nieskomplikowany, a wiedza szachowa zawarta w funkcji oceniającej nieduża. Okazało się jednak, że głębokość obliczeń dochodząca do 9-u półruchów dała "Belle" decydującą przewagę nad wszystkimi bardziej "doświadczonymi" programami. Był to początek tendencji nazwanych potem "brute force" - brutalna siła.

Kolejny Mistrz Świata Komputerów - "Cray Blitz" (Hyatt, Gower, Nelson) zawdzięczał swój dwukrotny (1983, 1986) sukces najszybszej wówczas maszynie na świecie: Cray X-MP . W warstwie programistycznej "Cray Blitz" był pochodną "Chess". W 1984-ym roku autorzy programu ponownie wyzwali na pojedynek Davida Levy'go. Jednak "Cray Blitz" gładko przegrał mecz z wynikiem 0-4.

Tendencje "brutalizacji" przy programowaniu gry w szachy zdominowały w znacznej części lata osiemdziesiąte. Praktyka potwierdzała tezę , że głębokość obliczeń wariantów z naddatkiem rekompensuje brak wiedzy szachowej w programie. Zaczęto więc powszechnie stosować dedykowane pod szachy układy elektroniczne. Na rynku pojawiły się pierwsze mikrokomputery szachowe sprzedawane na komercyjną skalę. W dziedzinie produkcji takich mikrokomputerów prym wiodły firmy: "Mephisto", "Fidelity" oraz "Chaitek". Ich siła stopniowo wzrastała wraz z postępem technologii elektronicznej od 1800 p. ELO w roku 1980, do 2200 p.

ELO w 1987 r. Tym samym zaczęły one zagrażać dużym programowanym "szafom" w stylu "Cray X-MP". Na rynku zaczęły się pojawiać programy szachowe na różne komputery sprzedawane jako produkty zabawowe.

Komercja i chęć osiągnięcia celu jak najkrótszą drogą chyba trochę zaszkodziła programowaniu szachów. Przeprowadzane regularnie zawody oraz wysokie nagrody jak np. 100.000 dolarów dla pierwszego programu, który pokona mistrza świata, przyczyniły się do dobrze znanych zjawisk strzeżenia informacji o swych programach. Schemat A Shanona plus elektronika dająca szybkość obliczeń stały się standardową, powszechnie stosowaną metodą.

W 1984-ym roku M.M.Botvinnik opublikował swoje prace nad programem "Pionier". Przez dwadzieścia pięć lat wieloletni Mistrz Świata w szachach zdołał doprowadzić swój program jedynie do poziomu mistrza i jego książka nie spotkała się z należytym zainteresowaniem. Botvinnik od początku był przeciwnikiem brutalnych metod przeszukiwania pełnego drzewa gry. Usiłował więc stworzyć nowy model szachów zbliżony do schematu B Shannona, ale jednak istotnie od niego różny. Była to jak do tej pory jedyna ważna praca nastawiona na zbliżenie algorytmu maszyny szachowej do algorytmu jakim posługują się ludzie.

Innym ważnym wydarzeniem lat osiemdziesiątych było stworzenie przez K. Thompsona programów rozgrywających bezbłędnie końcówki cztero- i pięciofigurowe typu "Król+Hetman: Król+wieża", czy "Król+Goniec+Goniec : Król+Skoczek". Komputery rozgrywały te pozycje rzeczywiście bezbłędnie, tym samym pokazano, że komputer jest w stanie robić coś w dziedzinie szachów lepiej od człowieka. Co więcej zaprezentowane przez komputer metody wygrywania końcówek uważanych dotąd za remisowe (np. "Król+Wieża+Goniec : Król+Wieża") okazały się zupełnie niezrozumiałe i nieprzyswajalne dla ludzi.

Mistrzostwa Świata z roku 1989 zakończyły się następującymi wynikami: Mistrz Świata na 1989 r. "Deep Thought", "Bebe", "Cray Blitz", "Hitech" i na piątym miejscu "Mephisto".

"Mephisto" wysunął się na czoło mikrokomputerów szachowych zarówno pod względem komercyjnym jak i czysto szachowym. Wprawdzie w 1989 na Mistrzostwach Świata był dopiero piąty, ale zaraz potem pokonał w bezpośredniej partii samego mistrza - kilkadziesiąt razy od niego cięższego "Deep Thougt".

"Hitech" był programem autorstwa grupy programistów pod kierownictwem zdobywającego coraz większą renomę w świecie specjalistę od programowania gier Hansa Berlinera, niegdyś Mistrza Świata w szachach korespondencyjnych.

Sukces niedużego komputera "Bebe" był pewnym zaskoczeniem. Tym niemniej "Bebe" należy uznać za ważny program, był to pierwszy samouczący się program/komputer szachowy.

W Mistrzostwach 1989 roku zwyciężył "Deep Thought" autorstwa F-h Hsu, T.S.Anantharamana, M.S.Campbela i A.Nowatrzyka. "Deep Thought" łączył w sobie wszystkie pozytywne doświadczenia swych poprzedników. Silny algorytm wyrosły z linii "Chess" - "Cray Blitz", nowe idee w przeszukiwaniu drzewa gry, funkcja oceniająca wyposażona w wiedzę szachową oraz układ elektroniczny dający programowi prędkość przeszukiwania rzędu 10^6 pozycji na sekundę, wszystko to dało programowi siłę ok. 2500 ELO tj. gracza szerokiej czołówki światowej.

W 1988 po raz pierwszy "Deep Thought" pokonał w normalnej partii arcymistrza (arcymistrz to najwyższy tytuł przyznawany szachistom przez Międzynarodową Federację Szachową FIDE). Był nim Bent Larsen, niegdyś jeden z najsilniejszych szachistów świata. W 1989 roku odbył się pierwszy mecz pomiędzy dwoma

oficjalnymi Mistrzami Świata w szachach: komputerów oraz ludzi. Mecz zakończył się zwycięstwem człowieka - Garry Kasparowa, genialnego ormiańskiego szachisty. Kasparow jest Mistrzem Świata od roku 1984-go i zapowiedział, że komputer nie będzie w stanie go pokonać przynajmniej do końca wieku . Jednak pod koniec roku 1989-go "Deep Thought" zdołał odnieść bardzo ważny sukces w walce z człowiekiem: wyzwał na mecz D.Levy'go i pokonał go z wynikiem 4-0 na swoją korzyść. Właśnie to wydarzenie okrzyknięto końcem pewnej epoki w programowaniu gry w szachy.

Tak więc siła gry komputerów szachowych osiągnęła poziom arcymistrza. Gdyby porównać siłę "Deep Thought" z szachistami-ludźmi to uplasowałby się mniej więcej na 150-tym miejscu w świecie. Nie ma już mowy o pokonaniu komputera przez człowieka-amatora, czy nawet średniej siły gracza turniejowego. Jest jednak na świecie ok. 150-u ludzi , którzy grają w szachy co najmniej tak samo lub lepiej niż "Deep Thought". Nie ma na razie mowy o pokonaniu przez komputer Garry Kasparova. Tak więc dzisiejsza epoka w programowaniu szachów ma jasny cel: pokonać mistrza świata !

W 1989 Newborn analizując wzrost siły programów szachowych doszedł do wniosku, że w 1992-im roku komputer będzie mistrzem świata w szachach. Newborn zauważył bowiem, że do 1989-go roku wzrost siły komputerów w punktach ELO był mniej więcej liniowy wraz z upływem lat. Koncern IBM wierząc, że detronizacji człowieka dokona właśnie "Deep Thought" podjął się sponsorowania tego projektu. Jednak wbrew przwidywaniom siła komputera od 1989 roku nie podniosła się ani trochę! Niedawno (sierpień 93) odbył się mecz sparingowy między dwoma "starymi" przeciwnikami: "Deep Thought" i arcymistrzem Bentem Larsenem. Tym razem Larsen zwyciężył z wynikiem 2.5 - 1.5 nie przegrywając żadnej partii .

Kiedy więc komputer będzie mistrzem świata w szachach? Na to pytanie nie można dzisiaj udzielić jakiejkolwiek odpowiedzi.

2. Matematyczny model gry w szachy.

Zaprogramowanie gry w szachy wymaga określenia jakiegoś jej modelu matematycznego. W fundamentalnej pracy Shannona przyjęto model zaczerpnięty z teorii gier.

Niniejszy rozdział traktuje skrótowo o teorii gier typu szachów. Omawiam tutaj jedynie podstawowe definicje, niezbędne do zrozumienia zasady MINIMAKSU. Postanowiłem przedstawić te definicje w sposób nieformalny, zainteresowanych matematycznymi sformułowaniami przedstawianych pojęć odsyłam do [1].

2.1. Gry dwuosobowe, deterministyczne, skończone, z pełną informacją, o sumie zerowej.

Z punktu widzenia teorii gier szachy można zakwalifikować jako grę dwuosobową, skończoną, z pełną informacją, o sumie zerowej. Do tej samej kategorii przynależy wiele popularnych gier: GO, warcaby, kółko i krzyżyk, reversi itp. Do wszystkich tych gier można stosować algorytmy oparte na omawianych dalej schematach Shannona.

Definicja 1. Grą dwuosobową, skończoną, z pełną informacją, o sumie zerowej nazywamy grę, o następujących własnościach:
(1) W grze bierze udział dwóch graczy (gra dwuosobowa),
(2) Każda rozgrywka musi się kiedyś skończyć (gra skończona),
(3) Obaj gracze mają dokładną informację o sytuacji w grze (gra z pełną informacją),
(4) Cele przeciwników są dokładnie przeciwstawne, zwycięstwo jednego oznacza porażkę drugiego (gra o sumie zerowej).

Uwaga 1. Skończoność szachów wynika z prawa trzykrotnego powtórzenia pozycji (trzecie powtórzenie się tej samej pozycji oznacza remis). Liczba możliwych konfiguracji na szachownicy jest skończona, powiedzmy ograniczona przez N. Tak więc po 2N +1 posunięciach jakaś pozycja musi powtórzyć się trzykrotnie.

Uwaga 2. Ogólnie w teorii gier przyjmuje się, że końcowym pozycjom w grach dwuosobowych o sumie zerowej przyporządkowana jest wypłata dla obu graczy dająca w sumie zero. Oznacza to, że jeśli gracz otrzyma w pozycji końcowej wypłatę 5, to przeciwnik otrzyma wypłatę -5. Zwyczajowo w szachach wartościuje się wyniki gry rozdzielając 1 punkt: zwycięstwo jednej ze stron 1:0, remis 1/2:1/2. Tak więc suma punktów zdobytych przez przeciwników wynosi 1. Szachy są jednak grą o sumie zerowej, wystarczy zmienić punktację na 1/2:(-1/2) za zwycięstwo i 0:0 za remis.

Uwaga 3. Specyficzną cechą szachów jest to, że gracze wykonują swoje posunięcia dokładnie na przemian.

Uwaga 4. Szachy można traktować jako drzewo. Pozycja początkowa stanowi korzeń tego drzewa, pozycje które można otrzymać bezpośrednio z pozycji początkowej są bezpośrednimi potomkami wierzchołka itd. Ponieważ jednak te same pozycje w szachach mogą powstać przy różnej kolejności posunięć wygodniej jest czasem patrzeć na szachy jako na graf a nie drzewo. Dla wygody będę dalej nazywał gracza, który w danej pozycji jest na posunięciu graczem, a drugiego z graczy przeciwnikiem. Będę też czasem używał terminologii grafowej nazywając pozycję "węzłem" a posunięcie "krawędzią".

Definicja 2. Strategią gracza nazywamy wybór "a priori", tzn. jeszcze przed rozpoczęciem gry, swego posunięcia w każdej możliwej w rozgrywce pozycji.

Uwaga 5. Szachistom wyraz strategia kojarzy się z czymś zupełnie innym !

Uwaga 6. Przy ustalonych strategiach obu graczy gra kończy się określonym "a priori" wynikiem.

Definicja 3. Gracz A wybiera pewną strategię S_A. Gracz B zna strategię gracza A i wybiera do niej strategię S_B taką, żeby rezultat gry był dla niego jak najkorzystniejszy. Rzecz jasna istnieje pewna optymalna strategia S_A* przy której gracz A zabezpiecza sobie pewien najkorzystniejszy w tej sytuacji wynik gry. Ten wynik nazywamy górną przegraną gracza A.

Uwaga 7. Ze skończoności gry wynika skończoność zbioru strategii obu graczy, co za tym idzie także skończoność iloczynu kartezjańskiego zbioru strategii obu graczy. Stąd jasno wynika istnienie optymalnej strategii S_A*.

Definicja 4. Gracz B wybiera pewną strategię S_B. Gracz A zna strategię gracza B i wybiera do niej strategię S_A taką, żeby rezultat gry był dla niego jak najkorzystniejszy. Rzecz jasna istnieje pewna optymalna strategia S_B* przy której gracz A nie jest w stanie osiągnąć więcej ponad pewien wynik gry. Ten wynik nazywamy dolną wygraną gracza A .

Uwaga 8. Odnośnie istnienia optymalnej strategii S_B* patrz uwagę 7 .

Twierdzenie 1. (Twierdzenie o MINIMAKSIE). Dolna wygrana gracza jest równa jego górnej przegranej.

Dowód Tw.1. Patrz [1].

Uwaga 9. Dolną wygraną gracza w danej pozycji, czy też jego górną przegraną, co na jedno wychodzi, nazywamy wartością minimaksową gracza w tej pozycji. Wartość minimaksowa gracza w danej pozycji jest liczbą przeciwną do wartości minimaksowej jego przeciwnika w tejże pozycji. Będziemy mówić poprostu o

wartości minimaksowej pozycji mając na myśli wartość minimaksową tego z graczy, który jest w tej pozycji na posunięciu.

Twierdzenie 2. Jeśli gracz jest na posunięciu to ma przynajmniej jedno posunięcie nie pogarszające, z jego punktu widzenia, wartości minimaksowej i nie ma żadnego, które mogłoby tą wartość poprawić.

Uzasadnienie Tw.2. Żeby nie pogorszyć swej wartości minimaksowej wystarczy wykonać posunięcie wg. optymalnej strategii S_A*. Natomiast każde inne posunięcie nie może poprawić wartości minimaksowej gdyż wtedy S_A* nie byłoby strategią optymalną.

Uwaga 10. Można to twierdzenie rozumieć po prostu tak, że jeżeli żaden z graczy nie popełni żadnego błędu, to gra skończy się pewnym z góry ustalonym rezultatem. Rezultat ten jest minimaksową wartością wierzchołka w drzewie gry.

Algorytm wyznaczania wartości minimaksowej.

Niżej podany jest rekurencyjny algorytm wyznaczania wartości minimaksowej danej pozycji P_0. Podobnie jak we wszystkich algorytmach grafowych, algorytmy rozgrywania gier są w naturalny sposób rekurencyjne. W algorytmie Minimaks użyto kilku niżej opisanych procedur i predykatów.

- Końcowa(P) : predykat, Końcowa (P) jest równe PRAWDA, jeżeli pozycja P jest końcową pozycją danej gry (np. mat w szachach).
- Wynik(P) : funkcja zwracająca wynik gry z punktu widzenia gracza.
- Dołącz_do_listy(W , m) : dołącza wartość m do listy liczb całkowitych W.

Max(W) : Podaje maksimum spośród liczb na liście W.

Min(W) : Podaje minimum spośród liczb na liście W.

```
Function Minimaks ( P : Pozycja ) : integer ;
var
    W : list of integer ;
begin
    if Końcowa ( P ) then
        return ( Wynik ( P ) ) ;
    else
        begin
            W := NIL ;
```

```
            for all  { P' : P' jest następnikiem P }
                Dołącz_do_listy( W , Minimaks (P') ) ;
            if  Gracz_A_na_posunięciu ( P ) then
                return ( Max ( W ) ) ;
            else
                return ( Min ( W ) )
        end
end ;
```

W celu znalezienia wartości minimaksowej pozycji P_0 wystarczy wywołać funkcję MiniMaks z parametrem P_0.

Zasada działania powyższego algorytmu opiera się wprost na twierdzeniu 2 .

Uwaga 11. Powyższy algorytm łatwo jest przerobić na grający program. Rzecz jasna program szachowy powinien wskazywać jakieś posunięcie w zadanej pozycji, a nie obliczać wartość minimaksową. Należy więc zapamiętać jedno z posunięć, które prowadziło do potomka o tej samej wartości minimaksowej.

Na zasadzie minimaksu oparte jest działanie wszystkich programów szachowych. Dokładnie tą samą zasadą posługują się w swym procesie myślenia ludzie, chociaż zdarza się, że w niektórych przypadkach świadomie od niej odstępują.

Trzeba jednak trzeba zwrócić uwagę, że nie da się zastosować zasady minimaksu wprost, gdyż obliczenie funkcji Minimaks dla szachów wymagałoby przejrzenia ok. 10^{43} pozycji a to jest niemożliwe. Trzeba zatem, podobnie jak czynią to ludzie, przejrzeć tylko okrojone drzewo gry wartościując w sposób heurystyczny liście tak okrojonego drzewa. Taki schemat zaproponował Shannon w 1950 roku.

3. Metody programowania gry w szachy.

Jak wspomniano w rozdziale 2.1. algorytm MiniMaks można łatwo przerobić na program rozgrywania gry w szachy. Szachy są jednak zbyt dużą grą do bezpośredniego zastosowania Minimaksu (można by to zrobić na przykład w przypadku dziewięciopolowego kółka i krzyżyka), gdyż drzewo gry w szachy ma ok. 10^{43} węzłów. Można zatem dokonać przeglądu jedynie fragmentu drzewa gry.

Dlatego potrzebna jest funkcja oceniająca (zwana też funkcją heurystyczną). Człowiek-szachista zawsze w swoim procesie myślenia dokonuje pewnych ocen końcowych pozycji wariantów które obliczył: "łatwa wygrana", "co najmniej remis", "przegrana pozycja z szansami na remis ok. 30%" itp. Funkcja oceniająca przyporządkowuje danej pozycji wartość z pewnego zbioru, w przypadku algorytmów minimaksowych wartość liczbową. W założeniu implementacja programowa funkcji oceniającej nie dokonuje żadnych obliczeń drzewa gry (Slate i Atkin w pracy [2] uważają takie podejście za nienaturalne, i w swoim programie "Chess" połączyli ze sobą moduł przeszukujący z oceniającym). Idealną funkcją oceniającą jest taka funkcja, która zawsze zwraca jako wynik dokładną wartość minimaksową pozycji. Oczywiście skonstruowanie takiej funkcji na ogół nie jest możliwe, gdyby było inaczej niniejsza praca byłaby w zasadzie bezprzedmiotowa. Konieczne jest zatem konstruowanie funkcji oceniających opartych na przesłankach heurystycznych.

Dysponując pojęciem funkcji oceniającej możemy skonstruować program rozgrywania gry w szachy wg. idei Shannona (patrz. [7]). Program taki składa się z trzech modułów:

(A) - Moduł implementacji zasad gry: zakodowanie "planszy" gry, generator posunięć, Wejście/Wyjście itp.
(B) - Moduł przeszukujący drzewo gry.
(C) - Funkcja oceniająca.

Pierwszy z nich to zwykłe programistyczne rzemiosło. Tym niemniej jakość tego rzemiosła nie jest bez znaczenia. Szybszy generator posunięć pozwala na głębsze przeszukanie drzewa gry, a co za tym idzie pozwala zwiększyć siłę gry programu.

W kolejnych podrozdziałach omawiam algorytmy przeszukiwania drzewa gry oraz metody konstruowania funkcji oceniających.

3.1. Funkcje oceniające.

W tym rozdziale omawiam trzeci z modułów klasycznie skonstruowanego programu szachowego - funkcję oceniającą. Szachy dysponują swoją narosłą przez lata kulturą: tysiącami książek poświęconych debiutom, końcówkom i strategii rozgrywania gry środkowej. Jest rzeczą oczywistą, że od ilości tej wiedzy zawartej w programie zależeć będzie jego siła. Funkcja oceniająca jest zasadniczą częścią programu szachowego, w której wiedza ta może zostać umieszczona. Wiedza szachowa może być także wykorzystana do sortowania rozpatrywanych posunięć wg. ich potencjalnej siły, (prezentowany w następnym rozdziale algorytm Alfabeta

działa optymalnie przy idealnym posortowaniu rozpatrywanych posunięć), heurystycznego odrzucania niektórych możliwości itp. Konstruowanie funkcji heurystycznej wymaga współpracy programistów ze specjalistami od rozgrywania gry i dlatego proces ten stanowi "wąskie gardło" przy pracy nad programami szachowymi.

Dla najczęściej stosowanych w praktyce algorytmów minimaksowych wartością funkcji oceniającej jest liczba. Wybór pomiędzy użyciem wartości całkowitej a rzeczywistej jest kwestią gustu. Istnieją jednak algorytmy wymagające innych wartości funkcji oceniającej. Dla przykładu: algorytm B* (patrz rozdział 3.3) wymaga pary liczb - pesymistycznej i optymistycznej oceny pozycji. W dalszym ciągu omawiam funkcje dające w wyniku wartość liczbową.

Od czasów Shannona jako f.o. używa się sumy:

$$E(P) = \Sigma_{i=0..N} (f_i * A_i(P))$$

E - funkcja oceniająca.

P - oceniana pozycja.

A_i - funkcja o wartościach w zbiorze { 0,1 } określająca, czy dana pozycja ma własność A_i.

f_i - waga własności A_i.

Podstawowym składnikiem powyższej sumy jest występujący w każdej pozycji bilans materialny, stąd najczęściej określa się go jako A_0 . Oblicza się go przyjmując wartości bierek:

pionek = 1 , skoczek = 3 , goniec = 3 , wieża = 5 , hetman = 9 .

Czasami podaje się również wartość materialną króla przyjmując pewną bardzo wysoką wartość np. król = 2000. Oczywiście za bierki przeciwnika należy odpowiednie wartości brać z minusem.

Pozostałe własności, zwane przez szachistów własnościami pozycyjnymi, są znacznie bardziej subtelne i trudne do zdefiniowania. Przykładami takich własności mogą być: "kontrola centrum", "bezpieczeństwo króla", "zaawansowane pionki" itp. Podstawową trudnością jest fakt, że własności takie wzajemnie na siebie oddziaływują i przy poszczególnych sytuacjach w grze mogą nawet być raz korzystne, a innym razem szkodliwe. Na przykład pozycja króla w centrum szachownicy jest silnym atutem w końcówce a decydującą słabością w grze środkowej.

Zawsze należy określić z czyjego punktu widzenia oceniamy daną pozycję. Jeśli, dla ustalenia uwagi, przyjmiemy że funkcja oceniająca zawsze ocenia pozycję z punktu widzenia białych, to powinna ona za białego pionka na szachownicy liczyć 1a za czarnego -1.

Przy konstruowaniu funkcji oceniającej trzeba określić:

(a) zbiór własności pozycji A_i ,

(b) wagi poszczególnych własności f_i .

Jak wspomniano, waga własności nie musi być stała a wręcz może przyjmować raz wartość ujemną, raz dodatnią.

Rozwiązania problemu (a) przedstawiono w wielu pracach traktujących o programach szachowych np. [2] str. 93-101, [5] str.91-115 czy [6] str. 119-126 . W części II niniejszej pracy, w rozdziale 5.3 prezentowana jest funkcja oceniająca programu "Joanna".

Problem (b) sprowadza się do określenia stosunku danej własności, np. "wieża na siódmej linii" do wartości materialnej jednego pionka. Jeśli więc przyjmiemy, że "wieża na siódmej linii" = 1 , to każda pozycja z wieżą na siódmej linii i jednym pionkiem mniej będzie przez program uznana za równą. W pracy [5] na str. 91-115 D. Hartman przedstawił interesujący algorytm weryfikacji statystycznej wag typowych własności pozycyjnych korzystając z zbioru 849-u partii rozegranych w turniejach arcymistrzowskich w roku 1986 . Idea algorytmu sprowadza się do sprawdzenia, na ile dana własność pozycyjna występowała w końcowych fragmentach wygrywanych, a na ile w końcowych fragmentach przegrywanych partii arcymistrzów. Jeżeli dana własność występowała w dużej liczbie końcowych części partii wygranych i w małej liczbie partii przegranych, to należy dać odpowiednio wysoki czynnik f_i przy tej własności.

Jak wspomniano, w różnych sytuacjach poszczególne własności pozycji A_i mogą mieć zupełnie odmienny wpływ na jej ocenę i powinno się przypisywać im różne wagi. Slate i Atkin [2] proponowali utworzenie bazy różnych funkcji oceniających. Przy rozpoczynaniu przeszukiwania drzewa pozycja początkowa podlegałaby klasyfikacji i na jej podstawie dokonywanoby wyboru odpowiedniej funkcji.

3.2. Algorytmy minimaksowe.

Schematy Shannona.

W roku 1950 roku Claude Shannon zaproponował dwa prezentowane niżej schematy wyznaczania wartości minimaksowej wierzchołka, nazwał je "Schemat A" oraz "Schemat B".

```
Function Shannon_A ( P : pozycja ; M : integer ) : integer ;
var
    W : list of integer ;
begin
    if (M = 0) OR Końcowa( P ) then
        Shannon_A := Funkcja_oceniająca ( P )
    else
        begin
            W := NIL ;
            for all  { P' : P' jest następnikiem P }
                Dołącz_do_listy( W , Shannon_A (P',M-1) ) ;
```

```
            if Gracz_A_na_posunięciu ( P ) then
                return ( Max ( W ) ) ;
            else
                return ( Min ( W ) ) ;
        end
end ;
.....
N := ... ;
wynik := Shannon_A ( P0 , N ) ;
```

Funkcja Shannon_A przypomina funkcję Minimaks. Różnica polega na tym, że drzewo gry zostanie przeszukane maksymalnie do głębokości N , zaś liście tak "okrojonego" drzewa zostaną ocenione przez funkcję oceniającą.

```
Function Shannon_B ( P : pozycja ; M : integer ) : integer ;
var
    W : list of integer ;
begin
    if (M = 0) OR Końcowa( P ) then
        Shannon_A := Funkcja_oceniająca ( P )
    else
        begin
            W := NIL ;
            for all  { P' : ( P' jest następnikiem P ) AND ( Posunięcie P->P' jest "sensowne" ) }
                Dołącz_do_listy( W , Shannon_A (P',M-1) ) ;
            if Gracz_A_na_posunięciu ( P ) then
                return ( Max ( W ) ) ;
            else
                return ( Min ( W ) ) ;
        end
end ;
.....
N := ... ;
wynik := Shannon_B ( P0 , N ) ;
```

Jak widać zasadniczą cechą Schematu B Shannona jest branie pod uwagę jedynie "sensownych" możliwości . Zasadnicza trudność polega na rozstrzygnięciu, jakie posunięcia rzeczywiście należy uznać za sensowne.

Procedura cięć alfa-beta.

Autorstwo procedury Alfabeta przypisywano różnym ludziom , wymienię więc kilka nazwisk: Brudno, Newell, Shaw, Simon. Idea działania procedury Alfabeta opiera się na obserwacji, że dla wyznaczenia minimaksowej wartości pewnego drzewa nie potrzeba znać minimaksowych wartości wszystkich jego węzłów ! Wytłumaczyć można to bezpośrednio na przykładzie szachów.

Przypuśćmy, że myśląc nad pewną pozycją znaleźliśmy posunięcie pozwalające wygrać nam "na czysto" skoczka. Szukamy jednak innych posunięć, możliwe jest bowiem, że możemy uzyskać jeszcze więcej. Jednak na kolejne rozpatrywane przez nas posunięcie przeciwnik dysponuje odpowiedzią, po której wygrywamy tylko pionka. Możliwe jest i to, że przeciwnik dysponuje nawet taką odpowiedzią, po której w ogóle nic byśmy nie wygrali. Czy trzeba szukać takiej odpowiedzi ? Oczywiście nie ! Należy powrócić do pozycji wyjściowej i szukać naszych możliwości wygrania więcej niż skoczka.

Na tej właśnie zasadzie opiera się procedura cięć alfa-beta . Cięciem nazywa się odrzucenie rozpatrywania części drzewa jak w przykładzie powyżej. Procedura Alfabeta działa przy założeniu, że mamy do czynienia z sekwencyjnymi obliczeniami. Odpowiednik algorytmu Alfabeta dla obliczeń równoległych podali R.A.Finkel i J.Fishburn.

Niżej przedstawiam pseudokod Alfabety w jej odmianie Falfabeta (Fail-Soft Alfabeta, Fishburn 1981) . Falfabeta pozwala ujednolicić zapis algorytmu licząc zawsze maksimum z odpowiedniego zbioru wartości. Możliwość takiego ujednolicenia wynika ze wzoru :

Min { x1 , x2 , ... } = - Max { -x1 , -x2 , ... }

Tak sformułowana zasada minimaksu nazywana jest zasadą negamaksu.

Będziemy też musieli ustalić założenia dotyczące funkcji oceniającej. Ustalamy, że funkcja oceniająca zwraca wynik z punktu widzenia strony będącej na posunięciu, tzn. jeśli pozycja jest wygrana dla białych, a na posunięcie są czarne, to funkcja oceniająca zwraca wynik ujemny.

```
function FAlfabeta ( P : Pozycja ; M : integer ; Alfa, Beta : integer ) : integer ;
begin
    if M = 0 then
        return ( FunkcjaOceniająca( P ) ) ;
    else
        begin
            Best := -MAXINT ;
            while jest_kolejny_ruch_do_rozpatrzenia  do
                begin
                    P' := pozycja_po_tym_ruchu ;
                    Value := -FAlfabeta( P',M-1,-Beta,-max(Alfa,Best) ) ;
                    if Value > Best then
```

```
                        begin
                            Best := Value ;
                            if Best >= Beta then
                                return ( Best )
                        end
                end ;
            return Best
        end
end ;
```

```
N := ... ;
Wynik := FAlfabeta ( P0 , N , -MAXINT , +MAXINT ) ;
```

Parametry Alfa oraz Beta , reprezentują tzw. gwarantowane wartości gracza i przeciwnika w aktualnym stanie obliczeń. Oznacza to, że w danej pozycji gracz nie zgodzi się na posunięcia nie dające mu więcej niż Alfa, a przeciwnik, w pozycjach które pojawią się w przeszukiwanym drzewie, nie zgodzi się na posunięcia nie dające mu mniej niż Beta. Początkowo te gwarantowane wartości wynoszą odpowiednio -MAXINT dla gracza i +MAXINT dla przeciwnika, innymi słowy przed rozpoczęciem przeszukiwania nic nie jest gwarantowane żadnej ze stron. Przy rekurencyjnym wywołaniu parametry Alfa i Beta zamieniają się miejscami i zmieniają znaki na przeciwne, tak że algorytm można było zapisać używając w treści procedury tylko parametru Beta.

Drzewo gry w algorytmie FAlfabeta jest przeszukiwane do głębokości N. Falfabeta jest więc modyfikacją Schematu A Shannona. Knuth i Moore udowodnili ([12]), że algorytm FAlfabeta daje dokładnie ten sam wynik, co Minimaks.

Niżej podaję szacowany zysk ze stosowania algorytmu cięć alfa-beta w miejsce zwykłego minimaksu. Dla tego celu będziemy musieli poczynić pewne uproszczenia co do modelu gry w szachy. Przyjmiemy model wyważonego drzewa gry o wysokości N i stopniu rozgałęzienia każdego niekońcowego węzła M. Mówiąc dalej o wartości minimaksowej drzewa, będziemy mieli na myśli wartość minimaksową takiego modelu przy zastosowaniu do liści funkcji oceniającej.

Jest oczywiste, że funkcja Minimaks dla takiego drzewa przegląda wszystkie M*N węzłów końcowych. Liczbę odwiedzanych przez dany algorytm węzłów końcowych przyjmujemy jako jego koszt (liczba węzłów końcowych jest znacznie większa od liczby węzłów wewnętrznych).

Efektywność algorytmu Alfabeta pokazali Knuth i Moore w [12]. Otóż efektywność ta silnie zależy od uporządkowania możliwości branych pod uwagę podczas przeszukiwania drzewa gry.

W przypadku pesymistycznym: wierzchołki potomne każdego węzła brane są w kolejności najgorszej, tzn. od ich najmniejszej wartości minimaksowej (z punktu widzenia posiadacza rozpatrywanego wierzchołka) do największej, algorytm Alfabeta zachowuje się dokładnie tak jak Minimaks. Nie ma żadnych cięć i w efekcie koszt działania wynosi M*N. Natomiast w przypadku optymistycznym , tzn. potomkowie danego węzła rozpatrywani są w kolejności od najlepszego do najgorszego, koszt działania algorytmu maleje do :

$2M^{N/2}-1$ dla N parzystych

$M^{(N+1)/2}+M^{(N-1)/2}-1$ dla N nieparzystych

W przybliżeniu więc zależność między kosztem pesymistycznym K_{pes} a optymistycznym K_{opt} wynosi :

$K_{pes} = K_{opt} * K_{opt}$

Drzewo gry o idealnym uporządkowaniu przeglądanych węzłów nazywane jest drzewem minimalnym. To drzewo zawsze musi zostać przeszukane dla znalezienia dokładnej wartości minimaksowej korzenia.

PAB, SCOUT, PVS.

PAB (Principal Variation Alfabeta) została zaprezentowana przez Fishburna i Finkela w 1980-ym roku.

Ideę działania PAB można przedstawić następująco. Najpierw brana jest dowolna ścieżka w rozpatrywanym drzewie gry. Następnie przyjmuje się hipotezę, że wartość funkcji oceniającej na liściu tej ścieżki jest dokładnie równa wartości minimaksowej korzenia. Innymi słowy przyjmuje się na początku, że "z nikąd" udało się zgadnąć najlepszy dla obu stron ciąg posunięć. Następnie bada się wszystkie węzły na tej ścieżce poczynając od ojca liścia, czy rzeczywiście żadne z alternatywych posunięć nie jest lepsze od założonego. Badanie takie ma tą zaletę w stosunku do FAB, że nie wymaga obliczenia dokładnej wartości minimaksowej węzła a jedynie stwierdzenie, że jest ona nie większa (nie mniejsza dla węzłów przeciwnika) od wartości założonej. Jeżeli jednak badanie wykaże istnienie lepszej alternatywy, należy uruchomić dla danego węzła algorytm FAB i znaleźć ową najlepszą alternatywę.

Niżej podajemy schemat algorytmu PAB.

```
function PAB ( P : Pozycja , M : integer ) : integer ;
begin
    if M = 0 then
        return ( FunkcjaOceniająca( P ) );
    else
        begin
            Weź_pierwszy_ruch_do_rozpatrzenia ( P );
            P' := pozycja_po_tym_ruchu;
            Best :=  - PAB ( P' , M-1 );
            while jest_kolejny_ruch_do_rozpatrzenia  do
                begin
                    P' := pozycja_po_tym_ruchu;
                    Value := -FAlfabeta( P',M-1,-Best-1,-Best );
```

```
                    if  Value > Best then
                        Best := -FAB ( P',M-1,-ý,-Value );
                end
            return  Best  ;
        end
end;
```

Algorytm Scout jest adaptacją ogólnego algorytmu do gier na potrzeby drzew minimaksowych. Scout opiera się na podobnej idei co PAB. W przeciwieństwie do PAB, która korzysta z algorytmu FAB przy powtórnych przeszukiwaniach, Scout wywołuje rekurencyjnie siebie samego. Natomiast do testowania alternatyw korzysta on z funkcji logicznej TEST zwracającej wartość FAŁSZ, jeśli znaleziono alternatywę lepszą od założonej. Poniższy tekst opublikował Pearl w 1980 roku.

```
function TEST ( P : Pozycja ; M , Value : integer ) : boolean ;
begin
    if M = 0 then
        return ( FunkcjaOceniająca( P ) > Value ) ;
    else
        begin
            while jest_kolejny_ruch_do_rozpatrzenia do
                begin
                    P' := pozycja_po_tym_ruchu ;
                    Search := NOT TEST ( P' , M-1 , -Value-1 ) ;
                    Value := -FAlfabeta( P',M-1,-Best-1,-Best ) ;
                    if  Search then
                        return ( TRUE ) ;
                end;
            return ( Best )
        end
end ;

function SCOUT ( P : Pozycja , M : integer ) : integer ;
begin
    if M = 0 then
        return ( FunkcjaOceniająca( P ) ) ;
    else
        begin
            Weź_pierwszy_ruch_do_rozpatrzenia ( P ) ;
```

```
            P' := pozycja_po_tym_ruchu ;
            Best := -SCOUT ( P' , M-1 ) ;
            while jest_kolejny_ruch_do_rozpatrzenia  do
                begin
                    P' := pozycja_po_tym_ruchu ;
                    Search := NOT TEST ( P' , M-1 , -Best-1 ) ;
                    if  Search then
                        Best := -SCOUT ( P', M-1 ) ;
                end;
            return ( Best )
        end
end ;
```

W 1983 roku Marsland przerobił algorytm SCOUT tak , by można było stosować cięcia alfa-beta. Algorytm ten opublikował pod nazwą Principal Variation Search (PVS).

```
function PVS ( P : Pozycja ; Alpha,Beta : integer ; M : integer ) : integer ;
begin
    if  ( M = 0 ) OR ( Końcowa ( P ) ) then
        return ( FunkcjaOceniająca( P ) ) ;
    else
        begin
            Weź_pierwszy_ruch_do_rozpatrzenia ( P ) ;
            P' := pozycja_po_tym_ruchu ;
            Best := -PVS ( P' , -Beta , -Alfa , M-1 ) ;
            while jest_kolejny_ruch_do_rozpatrzenia  do
                begin
                    P' := pozycja_po_tym_ruchu ;
                    if Best >= Beta then
                        return ( Best ) ;
                    Alfa := MAX( Best , Alfa ) ;
                    Value := -PVS ( P' , -Alfa-1 , -Alfa , M-1 ) ;
                    if  Value > Alfa AND Value < Beta then
                        Best := --PVS ( P' , -Alfa-1 , -Alfa , M-1 ) ;
                    else if Value > Best then
                        Best := Value ;
                end
            return ( Best ) ;
```

```
        end
end ;
```

SSS*.

Wszystkie prezentowane dotąd algorytmy działały wg. zasady "najpierw najgłębszy" (depth-first). Stanowią one znakomitą większość algorytmów minimaksowych stosowanych w praktyce. Algorytm SSS* wywodzi się z rodziny algorytmów A* działających wg. zasady "najpierw najlepszy" (best-first). W 1980-ym Nilson podał algorytm nazwany AO* będący adaptacją A* na potrzeby AND/OR grafów. SSS* (skrót od "State Space Search") podany został przez Stockmana w 1979 r. i jest on w zasadzie tożsamy z algorytmem AO*. Zasada działania algorytmu SSS* została opisana w rozdziale 3.4 przy okazji omawiania dynamicznego porządkowania rozpatrywanych posunięć.

SSS* jest "lepszy" od Alfabeta w następującym sensie: każdy węzeł odwiedzany przez SSS* musi być też odwiedzony przez algorytm Alfabeta, ale nie na odwrót. Dowód tego faktu podał Stockman, ale z błędem poprawionym przez Campbella (1981 r.).

Ze względu na wykładnicze wymagania pamięciowe algorytm SSS* nie jest stosowany w praktyce.

3.3. Inne algorytmy.

Jakkolwiek w praktyce programowania szachów używa się w zasadzie jedynie algorytmów minimaksowych, opracowano inne podejścia do problemu. Zasadniczą alternatywą do minimaksingu (z wszystkimi jego odmianami) jest algorytm B* .

B*.

Jak wspomniano w rozdziale o funkcjach oceniających zamykanie całej wiedzy szachowej w pojedynczej liczbie, jest słabą strona algorytmów minimaksowych. Na przykład w skomplikowanych pozycjach wartość funkcji może być obarczona znacznie większym błędem niż w prostych. Jeśli w takiej skomplikowanej pozycji z obopólnymi szansami funkcja oceniająca zwróci w wyniku 0, to oczywiście wymowa tego zera jest inna niż ten sam wynik funkcji oceniającej w końcowych pozycjach remisowych typu król przeciwko królowi.

Algorytm B* po raz pierwszy opublikowany został przez Hansa Berlinera w 1979-ym roku. Funkcja heurystyczna w algorytmie B* zwraca dwie wartości: pesymistyczną i optymistyczną ocenę pozycji, albo patrząc inaczej dolne i górne ograniczenia rzeczywistej wartości minimaksowej.

W przeciwieństwie do minimaksu B* nie ma za zadanie wartościowanie danej pozycji, a jedynie znalezienie najlepszego posunięcia. W tym celu B* posługuje się dwiema strategiami: PROVEBEST i DISPROVEREST . W przypadku obu strategii działanie algorytmu zmienia stopniowo pesymistyczne i optymistyczne oceny synów korzenia, posługując się przy tym zasadą minimaksu, ale oddzielnie dla tych dwu wartości. Działanie algorytmu

kończy się z chwilą uznania, że pesymistyczna wartość pewnego posunięcia jest większa lub równa optymistycznym wartościom pozostałych.

Strategia PROVEBEST próbuje udowodnić, że pewne wybrane posunięcie jest na pewno lepsze od pozostałych, natomiast DISPROVEREST próbuje dowieść, że wszystkie pozostałe posunięcia są na pewno gorsze od tegoż wybranego posunięcia . Na jedno wychodzi, ale różnica polega na wyborze węzła do dalszego rozpatrywania: PROVEBEST bada poddrzewo związane z założonym najlepszym posunięciem (za najlepsze posunięcie uznaje się to o najwyższej ocenie optymistycznej), DISPROVEREST bada poddrzewo związane z pewnym hipotetycznie słabym posunięciem, konkretnie tym, które daje największą nadzieję na szybkie odrzucenie. Teoretycznie B* może zatem nie przyjmować a priori założeń co do głębokości przeszukiwania.

Algorytm B* składa się z dwóch części: B* i B*_niskiego_poziomu. W algorytmie tym posunięcia w danej pozycji określone są przez potomków danej pozycji. Zamiennie używam też wyrazów "pozycja" i "węzeł". Procedura Rozwinięcie (P : pozycja) generuje potomków pozycji P i oblicza ich oceny optymistyczną i pesymistyczną. Procedura B*_niskiego_poziomu podana jest w formule negamaksowej.

```
Function B* ( P : pozycja ) : posunięcie ;
var
    W : set of pozycja  ;
begin
    W := rozwinięcie ( P ) ;
    best := Węzeł_o_max_górnej_ocenie( W ) ;
    alt_best := Węzeł_o_max_górnej_ocenie(W\{best});
    while best.ocena_dolna < alt_best.ocena_górna do
        begin
            strategia := wybierz_strategię ;
            if strategia = PROVEBEST then
                rozwijany_węzeł := best_node ;
            else
                rozwijany_węzeł :=  Węzeł_o_min_dolnej_ocenie(W\{best}) ;
            B*_niskiego_poziomu ( rozwijany_węzeł , strategia ) ;
            best := Węzeł_o_max_górnej_ocenie( W ) ;
            alt_best := Węzeł_o_max_górnej_ocenie(W\{best});
            return best_node
        end
end ;
Function B*_niskiego_poziomu ( P : pozycja , S : strategia ) ;
begin
    if  NOT rozwinięty ( P ) then
        rozwinięcie ( P )
```

```
    else
        begin
            węzeł := Wybierz_węzeł_do_rozwinięcia ( P , strategia ) ;
            B*_niskiego_poziomu ( węzeł ,strategia )
        end ;
    P.górne_ograniczenie :=  - Maximum_dolnych_ograniczeń_potomków ( P ) ;
    P.dolne_ograniczenie :=   - Maximum_górnych_ograniczeń_potomków ( P ) ;
end ;
```

W 1983-im A.J.Palay zaproponował użycie dystrybuant probabilistycznych w miejsce dwu wartości zwracanych przez funkcję oceniającą. W pracy [3] opublikował algorytm oparty na tej idei nazwany PB* (Probability-based B*).

3.4. Techniki programowania gry w szachy.

Przedstawione w poprzednich pracach algorytmy mogą być w różny sposób modyfikowane, wzbogacane różnymi technikami programistycznymi i heurystycznymi. Niżej podaję najczęściej stosowane.

Dobra kolejność rozpatrywania posunięć.

Jak wspomniano w poprzednim rozdziale, efektywność procedury Alfabeta zależy od kolejności rozpatrywania posunięć. Stosuje się więc różne metody dla ich uporządkowania. Najlepsze dzisiejsze programy potrafią przeszukiwać drzewa zaledwie trzykrotnie większe niż drzewo minimalne ! Trzeba jednak pamiętać, że gdybyśmy chcieli sortować potomków każdego węzła w drzewie, to koszt działania algorytmu trzeba przemnożyć przez koszt takiego sortowania.

Uporządkowanie statyczne (Slage , Dixon 1969) polega na obliczeniu funkcji oceniającej dla każdego badanego węzła drzewa. Następnie posunięcia sortowane są według tychże wartości. Ta metoda ma dwie wady: po pierwsze obliczenie funkcji oceniającej kosztuje, po drugie funkcja oceniająca obarczona jest pewnym błędem i w efekcie uporządkowanie takie może okazać się niedobre.

Ci sami autorzy zaproponowali więc uporządkowanie dynamiczne. Polega ono na następującej modyfikacji uporządkowania statycznego. Przy każdym kolejnym rozpatrywanym węźle obliczana jest od razu zmodyfikowana wartość minimaksowa na ścieżce od korzenia do tego węzła . Jeśli przy tym zmieni się uporządkowanie posunięć związanych z węzłami na tej ścieżce, to następuje powrót algorytmu do pierwszego miejsca, gdzie algorytm wybrał posunięcie prowadzące do pozycji o nienajwyższej ocenie i kontynuuje obliczenia biorąc pod uwagę zmodyfikowane wartości.

Tak naprawdę jest to zasada działania wspomnianego wcześniej algorytmu SSS*. SSS* pamięta przy tym dokładnie wszystkie oceny badanych dotąd węzłów, stąd jego wymagania pamięciowe rzędu wielkości rozpatrywanej przestrzeni stanów .

W praktyce, ze względu na koszty, korzysta się z innych metod. Przy programowaniu szachów, tańsze wydaje się sortowanie posunięć bez przyglądania się pozycjom do których one prowadzą. Przede wszystkim korzysta się z heurystyk związanych z konkretną dziedziną jaką jest gra w szachy. Bicia, posunięcia w kierunku centrum, ataki na bierki przeciwnika są posunięciami potencjalnie silnymi. Z drugiej strony korzysta się też z informacji zdobywanych w dotychczasowym procesie przeszukiwania.

Zasada analogii bierze się z obserwacji, że najsilniejsze możliwe w pewnej pozycji posunięcie, często bywa najsilniejszym w "podobnych" pozycjach. Na zasadzie analogii oparte są dwie metody. "Tablica morderców" ("killer table", Slate i Atkin, 1977) zawiera posunięcia, których rozpatrzenie doprowadziło do cięcia podczas działania procedury Alfabeta. Razem z posunięciem pamiętana jest głębokość w drzewie gry. Przy kolejnych pozycjach rozpatrywanych na tejże głębokości, posunięcia-mordercy, o ile tylko w danej pozycji są możliwe, mogą być rozpatrywane jako jedne z pierwszych. "Tablica historii" ("history table", Schaeffer, 1983) to tablica wszystkich możliwych posunięć, gdzie posunięcie określa się jednoznacznie polem źródłowym i docelowym. W tablicy takiej trzyma się pewne informacje o "przydatności" posunięć w dotychczasowym procesie przeszukiwania i wykorzystuje się do sortowania posunięć.

Należy tutaj wspomnieć o szeroko omawianych dalej tablicach transpozycji . W szachach na skutek możliwości "przestawiania posunięć" ta sama pozycja może być otrzymana z pewnej innej na kilka sposobów. Tym samym w procesie przeszukiwania te same pozycje mogą pojawiać się kilkakrotnie. Aby tego uniknąć trzyma się więc tablice transpozycji z informacjami z poprzednich obliczeń. Szeroko technika ta zostanie omówiona w następnych rozdziałach.

Iteracyjnie pogłębiana alfabeta.

W praktyce rozgrywania gier dysponuje się określonym limitem czasu do namysłu. Tymczasem działanie procedury Alfabeta tak silnie zależy od różnych czynników np. od uporządkowania rozpatrywanych posunięć, że nie da się w miarę dokładnie oszacować czasu jej działania. 1969-ym roku Scott zaproponował więc następujący schemat :

```
for i := 1 to N do
    wynik := Alfabeta ( P0 , αi , βi , i ) ;
```

przy asynchronicznym przerwaniu pętli po upłynięciu określonego limitu czasu.

Wprawdzie Scott rozważał jedynie uwarunkowania czasowe stawiane programom szachowym, jednak szybko okazało się, że iteracyjne pogłębianie przeszukiwania ma inne, zasadnicze zalety.

Przede wszystkim asymptotyczny koszt powyższego algorytmu jest równy zwykłej Alfabecie. Koszt ostatniej iteracji jest bowiem znacznie wyższego rzędu od poprzednich. Natomiast praktyczne implementacje iteracyjnie pogłębianej alfabety działają szybciej niż zwykłej !

Aby zrozumieć te zjawisko rozważmy na przykład prezentowane wyżej heurystyki uporządkowania rozpatrywanych posunięć. W początkowych iteracjach tablice posunięć-morderców zostaną wypełnione odpowiednią informacją. W efekcie w ostatniej iteracji dzięki lepszemu uporządkowaniu rozpatrywanych posunięć wzrośnie liczba cięć i rozpatrzone zostanie mniejsze drzewo.

Iteracyjnie pogłębianie przeszukiwania stosowane jest w niemal wszystkich dzisiejszych programach szachowych. Dalej prezentowane są kolejne techniki, które można stosować razem z iteracyjnie pogłębianą procedurą Alfabeta.

"Okno".

Częstość cięć w drzewie wzrasta przy zastosowaniu węższego przedziału (α , β). Procedura Alfabeta ma następującą własność: jeżeli rzeczywista wartość minimaksowa pozycji znajduje się wewnątrz inicjalnego przedziału (α , β) , to zwracana jest w wyniku dokładna wartość minimaksowa, jeśli rzeczywista wartość minimaksowa pozycji jest mniejsza od α, to w wyniku zwracana jest pewna wartość mniejsza lub równa α, jeżeli rzeczywista wartość minimaksowa jest większa od β, to zwracana jest wartość większa lub równa β. Stąd pojawia się idea, by zamiast początkowych wartości alfa i beta równych odpowiednio -MAXINT i +MAXINT , inicjalizować je do pewnych wartości α_0 i β_0 będących górnym i dolnym oszacowaniem rzeczywistej wartości minimaksowej pozycji wyjściowej. Jeśli oszacowanie okaże się prawdziwym zyskamy dzięki większej liczbie cięć, w przeciwnym razie trzeba powtórzyć przeszukiwanie.

```
α0 := oszacowanie_dolne_wartości_minimaksowej_P0 ;
β0 := oszacowanie_górne_wartości_minimaksowej_P0 ;
wynik := Alfabeta ( P0 , α0 , β0 ) ;
if wynik <= α0 then
    wynik := Alfabeta ( P0 , -MAXINT , α0 ) ;
else if wynik >= β0 then
    wynik := Alfabeta ( P0 , β0 , +MAXINT ) ;
```

Powyższy schemat nosi nazwę "Alfabeta z aspiracją" (Aspiration Alfabeta) i zdaje się być po raz pierwszy opisany przez Slate'a i Atkinsa w 1977. Przedział (α_0 , β_0) jest nazywany oknem. Oczywiście aspiracyjną alfabetę najlepiej stosować razem z iteracyjnym pogłębianiem, pozwala to na szacowanie wartości minimaksowej pozycji na podstawie wyniku poprzedniej iteracji.

Selektywne przeszukiwanie.

Schemat B Shannona postuluje branie pod uwagę jedynie "sensownych" posunięć w danej pozycji. Takie przeszukiwanie nazywamy selektywnym. Odrzucane posunięcia mogą być przy tym traktowane jak nieistniejące, a można też pozycje powstające po tych posunięciach uznać za końcowe i zastosować do nich funkcję oceniającą. Oczywiście zasadniczym problemem jest określenie kryteriów sensowności posunięcia.

Pierwszy historycznie program szachowy (Bernstein 1956) działał wg. zasady "best seven". W każdej pozycji, włącznie z początkową, wybierano siedem posunięć do rozpatrzenia. Niestety, często odrzucano przy tym pryncypialne kontynuacje. Odrzucenie najlepszego posunięcia na pryncypialnej ścieżce powoduje zmianę wartości minimaksowej rozpatrywanej pozycji i, co za tym idzie, wybór niewłaściwego posunięcia. Dzisiejsze programy skłaniają się więc do rozwiązań pośrednich pomiędzy przeszukiwaniem pełnym a selektywnym. Zazwyczaj do pewnej głębokości przeszukuje się pełne drzewo gry, a dalej stosuje się coraz węższą selekcję rozpatrywanych możliwości według pewnych opisanych dalej kryteriów, nie stosując przy tym jakiś stałych ograniczeń co do ich liczby, jak czynił to program Bernstaina. Takie stopniowanie selektywności przeszukiwania nazywa się podziałem przeszukiwanego drzewa gry na regiony: do głębokości N_1 mamy obszar pełnego przeszukiwania, od głębokości N_1+1 do N_2 pierwszy obszar przeszukiwania selektywnego, od N_2+1 do N_3 drugi obszar przeszukiwania selektywnego itd. Opis kryteriów stosowanych w obszarach przeszukiwania selektywnego zaczniemy od przedstawienia efektu horyzontalnego .

Funkcje heurystyczne z założenia nie dokonują żadnego przeszukiwania, oceniają pozycję jedynie na podstawie jej statycznego "wyglądu" . Przy tym w przypadku algorytmów minimaksowych wynik funkcji oceniającej jest zamknięty w jednej liczbie i nieznany jest dopuszczalny błąd tej oceny. Rozważmy następującą pozycję: białe mają tylko króla, czarne króla i wieżę, przy tym białe które są na posunięciu mogą zabić wieżę przeciwnika w jednym ruchu. Funkcja nie rozpatrująca dynamicznych własności przypisze tej pozycji bez wątpienia wartość odpowiadającą mniej więcej wygranej czarnych. Już minimaks o głębokości 1 wykazałby, że wyjściowa pozycja jest remisowa. Pierwsze programy oparte o schemat A Shannona, takie jak pierwotne wersje "Caissy", obserwowały powyższe zjawisko w wersji bardziej zakamuflowanej. Przykładowo, w jednej z partii ostatnim posunięciem na pryncypialnej ścieżce w przeszukiwanym drzewie gry było bicie wieży przeciwnika hetmanem. Posunięcie to było w rzeczywistości podstawką hetmana, gdyż wieża była broniona . W efekcie Caissa podjęła złą decyzję w pozycji wyjściowej. Zjawisko wahań między wartością przeszukiwania Minimaks (P, N) i Minimaks (P, N+1) nazywa się, analogicznie do odpowiedniego terminu w analizie numerycznej, niestabilnością funkcji oceniającej. Natomiast całe zjawisko podejmowania błędnych decyzji na skutek niewidoczności N+1-ego posunięcia nazywa się efektem horyzontalnym .

Z efektem horyzontalnym próbuje się walczyć stosując tak zwane warianty forsowne. Posunięciami zmieniającymi diametralnie wartość funkcji oceniającej są przede wszystkim bicia bierek. Potencjalnie groźnymi posunięciami mogą być też posunięcia zawansowanymi pionkami, szachy, posunięcia stwarzające groźby bicia itd. Stąd współczesne programy obliczają warianty forsowne biorąc pod uwagę coraz to węższe zbiory "niebezpiecznych" posunięć dzieląc drzewo gry na regiony według głębokości węzłów.

"Pusty ruch".

W szachach obie strony wykonują swoje posunięcia na przemian, nie ma możliwości rezygnacji z wykonania ruchu. Przy tym szachy są grą aktywną i w znakomitej większości pozycji prawo do wykonania posunięcia jest prawem korzystnym (istnieją wyjątki). "Pusty ruch" jest to posunięcie które nic nie zmienia na szachownicy. Taki "pusty ruch" jest niezgodny z przepisami gry w szachy.

Tym niemniej dopuszczenie "pustego ruchu", albo inaczej dwóch ruchów tej samej strony pod rząd, znajduje różne zastosowania w programach szachowych. Niżej podajemy jedno z nich.

Przed rozpatrzeniem "normalnych" posunięć rozpatruje się pusty ruch. Jeśli pusty ruch doprowadzi do cięcia alfa-beta, to traktujemy go tak, jak ruch zgodny z przepisami: następuje powrót sterowania do ojca węzła, rozpatrzenie następnego posunięcia itd. Ponieważ pusty ruch jest na ogół słabym ruchem, więc zapewne istniało przynajmniej jedno zgodne z zasadami posunięcie, które również prowadziło do cięcia (w grze jest to praktycznie zawsze prawdą, natomiast w końcówkach heurystyki tej lepiej nie stosować). Przy dużej ilości cięć w drzewie, oszczędza się czas, gdyż cięcie następuje bez uruchamiania generatora posunięć.

Stosując opisaną metodę należy pamiętać, że powinno się dopuścić tylko jedno puste posunięcie na dowolnej rozpatrywanej ścieżce.

Metody inkrementalno-dekrementalne.

W trakcie przeszukiwania drzewa gry trzeba wykonywać wiele procedur: generować posunięcia, oceniać pozycje itp. Pozycje powstające po wykonaniu jednego półruchu różnią się od siebie położeniem bierek zaledwie na dwóch polach (plus pewne dodatkowe statusy pozycji jak bicie w przelocie). Narzuca się więc rozwiązanie, w którym kolejne wywołania wyżej wymienionych procedur korzystają z wyników wywołań tychże procedur dla poprzedników i potomków danej pozycji. Rozważmy przykład. Funkcja oceniająca wywoływana jest dla wszystkich pozycji w drzewie gry. Jednym z podprogramów funkcji oceniającej jest ocena samej tylko struktury pionkowej pozycji. Jeśli więc w pewnej pozycji wykonano posunięcie figurą to w następnej pozycji wynik działania tego podprogramu będzie identyczny.

Z zasady metod inkrementalno-dekrementalnych korzysta funkcja oceniająca programu "Joanna". Inny, klasyczny przykład stanowi opisana w następnym rozdziale funkcja haszująca dla tablic transpozycyjnych.

Metody brutalne ("brute force").

Siła programów rozgrywających gry , zależy bardzo wyraźnie od głębokości przeszukiwania. Oczywiście katastroficzna, wykładnicza złożoność algorytmu, nie pozwala swobodnie zwiększyć tej głębokości. Współczesne komputery szachowe używają wiec specjalizowanych układów VLSI zaprojektowanych do wykonywania niektórych procedur algorytmu. Pozwala to uzyskać ogromną szybkość i zwiększyć głębokość przeszukiwania nawet do kilkunastu półruchów. Przy tym ze względu na niebezpieczeństwa związane z selektywnym przeszukiwaniem istnieje tendencja, do wykorzystywania, w miarę możliwości Schematu A Shannona. Innymi słowy "brute force" to:

prosty, kosztowny, ale gwarantujący poprawny wynik algorytm + potężna moc obliczeniowa.

Mistrz Świata Komputerów Szachowych z 1989 roku "Deep Thought" miał prędkość ok. $2 * 10^6$ węzłów / sek. Dla porównania: szachista podczas wyboru posunięcia prawdopodobnie nie rozważa więcej niż 50-u pozycji. Dzięki tej prędkości "Deep thought" potrafi liczyć warianty nawet do 18-u pólruchów. Pomimo że "Deep Thought" ma nieodbiegający zbytnio od standardów algorytm, zastosowanie tak ogromnej mocy obliczeniowej pozwoliło mu na pokonanie mm. Davida Levy'go i osiągnięcie innych liczących się wśród światowej czołówki sukcesów.

3.5 Tablice transpozycyjne.

Treść niniejszego podrozdziału, tematycznie należy do poprzedniego , ale ze względu na znaczenie techniki tablic transpozycyjnych dla niniejszej pracy omawiam ją szczegółowo. Tablice transpozycyjne zostały po raz pierwszy opisane przez Davida Slate'a i Lawrence Atkinsa w pracy [2] z 1977 r. poświęconej programowi "Chess 4.5" .

Idea opiera się na spostrzeżeniu, że ta sama pozycja w grze może powstać przy różnych przestawieniach posunięć. Co więcej, węzły rozpatrywane w trakcie przeszukiwania pewnej pozycji w grze, pojawią się w drzewach przeszukiwanych w następnej rozpatrywanej pozycji, ale o dwa półruchy bliżej korzenia.

W związku z tym autorzy programu "Chess" wpadli na następujący pomysł. Stworzyli tablicę z informacjami o już rozpatrywanych pozycjach (format rekordu w tej tablicy podano dalej). Ponieważ uzyskanie rekordu z tej tablicy musi być szybsze niż powtórzenie uprzednio wykonanej pracy, opracowano metody haszowania tej tablicy z pomocą szybkich funkcji. Tablice transpozycyjne stanowią bardzo silne narzędzie przy programowaniu gry w szachy. Przy korzystaniu z nich trzeba jednak rozwiązać jeden problem. Rozmiar tablicy ograniczony jest dostępnością pamięci. Zazwyczaj jest on porównywalny z liczbą węzłów w rozpatrywanym drzewie gry. Po całkowitym zapełnieniu tablicy trzeba określić strategię, które pozycje zasługują na zapamiętanie w tablicy transpozycji, a które nie.

Niżej podany jest "standardowy" format rekordu tablicy transpozycji. W rekordzie podanego niżej formatu trzymane są informacje, których wykorzystanie pozwala zmodyfikować procedurę Alfabeta, stąd jest to rekord ściśle powiązany z algorytmami minimaksowymi.

- górne ograniczenie wartości minimaksowej wierzchołka ;
- dolne ograniczenie wartości minimaksowej wierzchołka ;
- flaga
- posunięcie uznane za najlepsze ;
- wysokość przeszukanego poddrzewa ;
- wartość funkcji haszującej dla tej pozycji .

Konieczność przechowywania dwóch pól: górnego i dolnego ograniczenia wartości minimaksowej węzła, wynika stąd, że dla węzłów wewnętrznych drzewa Alfabeta na ogół nie zwraca dokładnego wyniku (na skutek cięć). Pole flaga określa, czy w rekordzie trzymane jest dolne, czy górne ograniczenie wartości minimaksowej, czy też dokładna wartość. W tym ostatnim przypadku dolne ograniczenie równe jest, rzecz jasna, górnemu.

Przy haszowaniu tablic trzeba rozwiązać problem kolizji, tzn. trafienia w tą samą pozycję tablicy. Jednak standardowe metody, np. umieszczanie rekordu na pierwszym wolnym miejscu za miejscem wystąpienia kolizji wydłużyłyby niepotrzebnie czas dostępu. Na ogół przyjmuje się zasadę, że w przypadku kolizji poprzednia wartość na danym miejscu w tablicy jest poprostu nadpisywana. Wynik funkcji haszującej musi być liczbą całkowitą z dostatecznie dużego przedziału, tak by kolizja wartości kluczy dwóch pozycji była wprawdzie teoretycznie możliwa, ale prawdopodobieństwo jej wystąpienia zaniedbywalne. W pracy [4] podano, że wynik funkcji haszującej powinien być liczbą conajmniej 32-u bitową. Wartość tej funkcji trzymana jest w rekordzie, natomiast jako adres rekordu brana jest liczba zapisana przez pierwszych kilka bitów klucza w zależności od rozmiaru tablicy (rozmiar ten jest na ogół potęgą dwójki). Oczywiście zwielokrotnia to liczbę trafień w to samo miejsce w tablicy, ale kolizja wykrywana jest dzięki trzymaniu w tablicy pełnego klucza.

Informacja zawarta w tablicy transpozycji może albo całkowicie zastąpić działanie procedury Alfabeta, albo zawęzić okno dalszego przeszukiwania, albo przynajmniej wskazać obiecujące posunięcie.

Szczególnie ważnym jest użycie dostatecznie szybkiej funkcji haszującej. Opisana niżej metoda pochodzi od Zobrista. Idea Zorbista opiera się na wykorzystaniu metod inkrementalno-dekrementalnych i własności funkcji "EXCLUSIVE OR" .

W grze w szachy bierze udział dwanaście różnych bierek. Bierki te mogą znajdować się na 64-ech różnych polach. Przyporządkujmy więc każdej możliwej kombinacji "bierka X na polu Y" 12 * 64 różnych losowo wybranych liczb całkowitych $\{ R_i \}_{i=1..12*64}$. Dodatkowo należy jeszcze przyporządkować liczby $\{ R_i \}_{i=12*64+1}$ kilku własnościom pozycji: kolejność posunięcia, status roszad, bicie w przelocie itp. Funkcję haszująca Zorbista oblicza się według wzoru:

$P_i = R_a \ \text{XOR} \ R_b \ \text{XOR} \ ... \ \text{XOR} \ R_x$.

Gdzie XOR oznacza bitową różnicę symetryczną liczb całkowitych, natomiast R_a oznacza liczbę przypisaną odpowiedniemu położeniu danej figury na danym polu szachownicy.

Jak łatwo pokazać, jeśli pozycja P_j powstaje z P_i przez posunięcie bierki z pola o przypisanej liczbie R_k na pole o przypisanej liczbie R_l , to zachodzi wzór :

$P_j = P_i \ XOR \ R_k \ XOR \ R_l$.

Jeśli natomiast przy posunięciu tym zbito dodatkowo na polu docelowym figurę a liczba przypisana docelowemu polu z tą figurą jest równa Rm , to zachodzi :

$P_j = P_i \ XOR \ R_k \ XOR \ R_l \ XOR \ R_m$.

Jest jeszcze kilka szczególnych przypadków dla promocji pionków, roszady i bicia w przelocie, ale podanie odpowiednich wzorów dla tych przypadków, pozostawiam czytelnikowi.

Niżej podany jest tekst procedury Alfabeta korzystającej z mechanizmu tablic transpozycyjnych, oraz opis zawartych w niej podprocedur.

Retrieve (P , height , value , flag , move) ;

Wyszukanie w tablicy transpozycyjnej rekordu związanego z pozycją P . Na kolejne parametry zwracane są kolejne wartości z tablicy. Jeśli odpowiedni rekord w tablicy jest pusty , na zmienną flag podstawiana jest wartość EMPTY.

Store (P , M , Best , flag , move) ;

Zapisanie w tablicy transpozycji rekordu. Przed zapisaniem wartość zmiennej M porównywana jest z istniejącą w tablicy. Zapis uzależniony jest od spełnienia warunku : height < M .

```
function Alfabeta ( P : Pozycja ; M : integer ; alfa, beta : integer ) : integer ;
begin
    Retrieve ( P , height , value , flag , move ) ;
    if height >= M then
        begin
            if ( flag = VALID ) then
                return ( value ) ;
            if ( flag = LBOUND ) then
                alfa := max ( alfa , value ) ;
            if ( flag = UBOUND ) then
                beta := min ( beta , value ) ;
            if ( alfa >= beta ) then
```

```
                return ( value ) ;
            end ;
        if M = 0 then
            return ( FunkcjaOceniająca( P ) ) ;
        else
            begin
                Best := -MAXINT ;
                while jest_kolejny_ruch_do_rozpatrzenia  do
                    begin
                        P' := pozycja_po_tym_ruchu ;
                        Value := -Alfabeta( P',M-1,-Beta,-max(Alfa,Best) ) ;
                        if  Value > Best then
                            begin
                                Best := Value ;
                                if  Best >= Beta then
                                    return ( Best ) ;
                            end
                    end
            end ;
    DONE :
        flag := VALID ;
        if ( Best <= alfa ) then
            flag := UBOUND ;
        if ( Best >= beta ) then
            flag := LBOUND ;
        if ( height <= M ) then
            Store ( P , M , Best , flag , move ) ;
        return ( Best ) ;
    end ;
```

4. Program "Joanna".

Pracę nad programem "Joanna" rozpocząłem w październiku 1992-go roku. Początkowo program był pisany z wykorzystaniem języka i systemu TURBO PASCAL 6.0 firmy Borland. W tym języku powstała pierwotna wersja generatora posunięć wraz z prostą funkcją oceniającą. Interfejs graficzny programu został napisany w języku MICROSOFT C 6.0 z wykorzystaniem biblioteki graficznej "ViDE". Ponieważ kompilatory firmy Borland i MICROSOFT nie są ze sobą zgodne, zdecydowałem się w lipcu 1993-go roku przepisać kod pascalowy na C w celu uzyskania jednolitego programu.

Już na początku roku 1993-go, jeszcze w języku PASCAL istniał moduł implementujący grę: zakodowanie "planszy", generator posunięć, itp. Po dodaniu prototypu funkcji heurystycznej oraz prostego modułu przeszukującego powstał w maju 1993-go roku prototyp programu, stopniowo usprawniany w następnych miesiącach. Takie stopniowe poprawianie programu ma oczywiste wady, rozwiązania przyjęte na początku powstawania projektu niejednokrotnie okazują się nieprzemyślane, a oddziaływują na powstające w późniejszym czasie części programu.

Idea wprowadzenia do programu tablic transpozycyjnych z algorytmem nazwanym przeze mnie "BeBe+" istniała od października 1992, ale decyzja o jej realizacji została podjęta w początkach kwietnia 1993.

Implementację tablic transpozycyjnych dodałem wraz z przepisaniem całości kodu na język C w lipcu 93. Po przepisaniu kodu okazało się że program zawiera ogromną liczbę błędów i kiedy na jesieni 93-go roku rozpocząłem testowanie programu, dziennie wykrywałem średnio dwa poważne błędy w programie. Sam program liczył już wtedy ok. ośmiu tysięcy linii dość skomplikowanego kodu. W styczniu roku 94-go, po gruntownym testowaniu, program został w końcu doprowadzony do "stanu używalności". Na jesieni 93-go roku rozpocząłem prace nad algorytmem "BeBe+". Algorytm rozpoczął swe działanie w marcu. Po dokonaniu testów programu w maju 1994, po półtorarocznej pracy program "Joanna" został ukończony.

Program składa się z sześciu modułów :

(a) generatora posunięć wraz z procedurami implementacji reguł gry,
(b) modułu przeszukiwania drzewa,
(c) funkcji oceniającej,
(d) modułu obsługi tablicy transpozycyjnej wraz z algorytmem "BeBe+",
(e) interfejsu użytkownika.

W następnych rozdziałach omówione zostaną poszczególne z nich.

4.1. Struktury danych, generator posunięć.

Szybkość działania generatora posunięć oraz szybkość uzyskiwania rozmaitych informacji poprzez poszczególne procedury programu, przede wszystkim przez funkcję oceniającą, mają krytyczne znaczenie dla prędkości przeszukiwania drzewa gry, a co za tym idzie dla siły gry programu (patrz rozdz. 3.4. "metody brutalne").

Działanie modułu opiera się na listowych strukturach danych. Wybór listowych struktur danych jest dla algorytmów grafowych naturalny, dla przykładu język "Lisp" powstał między innymi z myślą o przetwarzaniu języków naturalnych i programowaniu gier. Po części jednak z braku łatwego dostępu do kompilatora LISP-u, a także z innych, opisanych wcześniej przyczyn, program "Joanna" napisany został w języku C (ciekawe, że ogromna większość programów szachowych napisana została w języku C, w asemblerze lub wręcz w mikrokodach danych maszyn).

Generator posunięć programu przegląda szachownicę po kolei od lewego górnego rogu do prawego dolnego wierszami i przy napotkaniu figury gracza będącego na posunięciu generuje wszystkie możliwe ruchy tej figury.

Generator udostępnia trzy procedury: generator posunięć zgodnych z regułami gry, znacznie szybszy generator pseudoposunięć oraz generator pseudoposunięć-bić. Dla list posunięć generowanych przez poszczególne procedury generujące zdefiniowane są odpowiednie procedury sortujące.

(a) Generator posunięć zgodnych z regułami gry.

Procedura ta generuje wszystkie posunięcia zgodne z regułami szachów. Wykorzystywana jest jedynie przez funkcje logiczne określające, czy dana pozycja jest matem lub patem lub szachem.

(b) Generator pseudoposunięć.

Pseudoposunięcia to wszystkie zgodne z przepisami posunięcia, oraz takie, które pozostawiają króla pod biciem lub podstawiają króla pod bicie. Dopuszczenie podstawki króla czyni generator pseudoposunięć kilkakrotnie szybszym od generatora posunięć zgodnych z przepisami. Generator pseudoposunięć wykorzystywany jest w rejonie pełnego przeszukiwania drzewa gry. W regionie tym lista psuedoposunięć jest sortowana w kolejności:

- bicia w kolejności wartości materialnej zdobytego materiału,
- posunięcia z tablicy posunięć morderców,
- pozostałe posunięcia.

Technika "posunięć-morderców" opisana została w rozdz. 3.4. Technika ta służy do dynamicznego porządkowania rozpatrywanych posunięć. Stosowanie tej techniki bez wątpienia przyspiesza przeszukiwanie drzewa, jednak z przyczyn omówionych dalej, problematyczne jest stosowanie dynamicznego porządkowania posunięć razem z algorytmem "BeBe+". Dlatego "Joanna" przy "włączonym" algorytmie BeBe+ odłącza sortowanie posunięć z tablicy morderców. Bliżej problem ten omówiony zostanie w rozdz. 5.3.4.

Tablica posunięć-morderców jest zorganizowana dosyć prymitywnie. W tablicy tej znajdują się trzy posunięcia, które ostatnio w procesie przeszukiwania doprowadziły do cięcia. Nie ma rozróżnienia poziomu węzła w przeszukiwanym drzewie. Strategia wymiany zawartości tablicy opiera się na zasadzie FIFO.

Wszystkie bicia sortowane są według wartości zysku materialnego. Zyskiem materialnym związanym z danym posunięciem nazywać będę wartość materialną zbitej bierki, a w przypadku promocji wartość promowanej bierki minus wartość znikającego z szachownicy pionka.

(c) Procedura generująca pseudoposunięcia-bicia.

Procedura generująca psudoposunięcia-bicia wykorzystywana jest w regionie selektywnego przeszukiwania drzewa. Z procedurą tą związany jest jej parametr "deltaM" oraz stała TOLERANCJA".

Do listy pseudoposunięć-bić zdefiniowana jest procedura usuwająca wszystkie bicia nie dające zysku materialnego równego co najmniej: ("deltaM" - "TOLERANCJA"). Ma ona za zadanie wymusić, by w trakcie przeszukiwania ciągów wymian na zabicie np. skoczka, przeciwnik odpowiadał również zabiciem co najmniej skoczka. Parametr "deltaM" pokazuje dotychczasowy bilans materialny od rozpoczęcia przeszukiwania ciągów wymian (chodzi o przeszukiwanie selektywne, czyli wariant forsowny). Należy jednak przy tym narzucić pewną tolerancję, pozwalającą kontynuować przeszukiwanie, nawet jeśli nie da się wyrównać bilansu materialnego. Innymi słowy, dopuszczalne jest, by na zabicie wieży odpowiedzieć zabiciem tylko skoczka. W przeciwnym razie "Joanna" nie była w stanie obliczyć nawet prostych kombinacji z poświęceniem niewielkiej ilości materiału. Nie potrafiłaby się nawet pogodzić z wymianą gońca na skoczka, gdyż goniec ma w programie nieco wyższą wartość materialną niż skoczek. Aktualnie wartość stałej TOLERANCJA ustawiona jest na 125, czyli wartość materialną jednego i jednej czwartej pionka.

Dla trzech specyficznych sytuacji szachów: szacha, mata i pata, zdefiniowane sa odpowiednie funkcje logiczne. Zdefiniowane są także różne procedury niezbędne w programie: wykonywanie posunięcia, sprawdzanie czy dane posunięcie jest w danej pozycji możliwe, porównywanie dwóch pozycji itp.

Jak już napisałem, jakość całego modułu pozostawia wiele do życzenia. W skomplikowanych pozycjach z gry środkowej wykonanie przeszukiwania do głębokości 2 zajmuje "Joannie" ok. 5 sek. Popularne programy dostępne na rynku oprogramowania, np. "Chess Master" czy "GNU-Chess", w takim samym czasie przeszukują drzewo do głębokości 4.

Uwagi.

Moduł generowania posunięć jest niezbyt udaną częścią programu. Rozdział ten postanowiłem zakończyć kilkoma postulatami, których spełnienie pozwoliłoby na utworzenie lepszego generatora. Jest to znane wielu programistom zjawisko, że dopiero po napisaniu jakiegoś programu zdobywa się doświadczenie pozwalające napisać go po raz drugi, tym razem dobrze.

(1) W miarę możliwości nie powinno się alokować dynamicznie pamięci. Możliwie dużo struktur danych będących z natury strukturami dynamicznymi (listy, drzewa itp.) implementować w statycznie alokowanych tablicach.
(2) Generator posunięć nie powinien generować posunięć jedynie na podstawie zadanej pozycji. Powinno się zaprojektować dla niego strukturę danych pozwalającą na przyspieszenie generacji. Przykład takiej struktury opisano w [2] przy omówieniu bazy danych programu "Chess 4.5".
(3) Są dwa zasadnicze podejścia do generowania posunięć. Pierwsze z nich polega na generowaniu posunięć z danego pola szachownicy, drugie polega na generowaniu posunięć do danego pola. To drugie z nich ma kilka niewątpliwych zalet. Po pierwsze pozwala szybko wygenerować wszystkie bicia. Po drugie pozwala uporządkować posunięcia w kierunku ważnych strategicznie pól (np. centrum) już w trakcie generowania posunięć.
(4) Jak najpełniej korzystać z techniki "podręcznikowania", tzn. zapisywać wyniki poszczególnych funkcji programu (np. funkcji oceniającej) w haszowanych tablicach analogicznych do tablic transpozycyjnych opisanych w rozdz. 3.5.

4.2. Moduł przeszukujący.

Podział drzewa gry na regiony, ocena pozycyjna a ocena materialna.

Przeszukiwanie w programie jest oparte na standardowej zasadzie regionów (patrz. rozdz. 3.4. "przeszukiwanie selektywne"). Joanna dzieli drzewo na trzy regiony przeszukiwania. Pierwszy z nich to oczywiście region pełnego przeszukiwania drzewa gry. Po nim następują dwa regiony selektywnego przeszukiwania. Region przeszukiwania pełnego będę dalej nazywał bazowym.

W przeszukiwaniu "Joanna" korzysta z generatora pseudoposunięć. Podstawka króla wykrywana jest dopiero w trakcie dalszego przeszukiwania w momencie, gdy wykryta zostanie możliwość bicia króla. Sterowanie powraca wtedy do poprzedniej pozycji na ścieżce przeszukiwania i uruchamiana jest procedura rozpoznająca, czy dopuszczenie do bicia króla nastąpiło z powodu mata, pata czy też zwykłej podstawki.

Funkcja oceniająca programu (patrz rozdz. 4.3.) składa się z dwóch części: funkcji oceniającej materialnie, oraz funkcji oceniającej pozycyjnie. Przeszukiwanie działa według następującej zasady. W pozycjach końcowych regionu bazowego obliczana jest funkcja oceniająca pozycyjnie, następnie do wyniku tej funkcji dodawany jest wynik przeszukiwania selektywnego, przy czym przeszukiwanie selektywne korzysta z funkcji oceniającej materialnie.

Drugi z regionów przeszukiwania selektywnego, stanowi "wariant forsowny" programu (patrz. rozdz. 3.4.). Wariant forsowny nie może dać w wyniku mniej niż ocena materialna pozycji, w której rozpoczęto przeszukiwanie selektywne. W regionie tym brane są pod uwagę jedynie bicia.

Natomiast pierwszy region przeszukiwania selektywnego ma głębokość jednego półruchu i stanowi jakby płynną granicę między przeszukiwaniem pełnym a selektywnym. W zależności od danej pozycji w regionie tym dokonuje się albo przeszukiwanie pełne jak w regionie bazowym albo selektywne jak w wariancie forsownym.

Postanowiłem wprowadzić taką płynną granicę dla zmniejszenia efektu horyzontalnego. Idea jest następująca. Jeśli ostatnim posunięciem w regionie bazowym było stworzenie groźby "nie do odparcia" (np. popularnie zwanych przez szachistów "wideł") to zaatakowana strona musi nieuchronnie stracić materiał. Jednak wariant forsowny "Joanny" zwraca wynik zawsze nie mniejszy niż ocena materialna pozycji, w której rozpoczęto przeszukiwanie selektywne. Należy zatem niejako przedłużyć obszar pełnego przeszukiwania w celu wykrywania takich gróźb "nie do odparcia". Jeżeli w rozpatrywanej pozycji strona nie będąca na posunięciu nie dysponuje żadną groźbą mającą pozory "groźby nie do odparcia", przeszukiwanie przechodzi od razu do wariantu forsownego. W programie za "groźbę" przedłużającą pełne przeszukiwanie uznawane jest zagrożenie zbicia dowolnej figury nie licząc pionków.

Iteracyjne pogłębianie regionu bazowego, ograniczenia czasowe.

Jak niemal każdy współczesny program szachowy, "Joanna" korzysta z iteracyjnie pogłębianego przeszukiwania. Pogłębiana jest jednak jedynie głębokość regionu bazowego począwszy od głębokości 2. Maksymalne głębokości regionów selektywnych są stałe i wynoszą 1 dla regionu pierwszego i 6 dla drugiego. Pierwsza iteracja musi zawsze zostać wykonana, niezależnie od ewentualnego przekroczenia limitu czasu.

Przeszukiwanie kończy się z chwilą upłynięcia limitu czasowego, który jest jednym z parametrów programu wprowadzanym przez użytkownika.

"Okno".

"Joanna" stosuje metodę "okna" (patrz rozdz. 3.4.). Każda iteracja rozpoczyna się przy założeniu, że wartość minimaksowa z zmieści się w pewnym przedziale (α_0 , β_0). Wartości α_0 i β_0 obliczane są na podstawie wyniku z poprzedniej iteracji, zaś dla pierwszej iteracji na podstawie oceny pozycji wyjściowej. Niżej podany jest wzór na oszacowanie α_0 i β_0.

$\alpha_0 = W - (P/2 + (5 - G) * P / 10)$

$\beta_0 = W + (P/2 + (5 - G) * P / 10)$

dla $G <= 4$;

$\alpha_0 = W - P / 2$

$\beta_0 = W + P / 2$

dla $G > 4$;

Gdzie:

P : Wartość materialna pionka,

G : Głębokość regionu bazowego w przeszukiwanym drzewie.

W : wynik z poprzedniej iteracji lub z oszacowania dla pierwszej iteracji.

Oczywiście, jeżeli założenie okaże się fałszywe konieczne jest powtórne przeszukiwanie. "Joanna" poprawia przeszukiwanie najpierw w przedziałach:

$< \alpha_0 - 150 , \alpha_0 >$ lub $< \beta_0 , \beta_0 + 150 >$,

a jeżeli i ta druga próba zawiedzie, ostatecznie dokonuje przeszukiwania dla oszacowań:

$< -MAXINT , \alpha_0 - 150 >$ lub $< \beta_0 + 150 , +MAXINT >$.

4.3. Funkcja Oceniająca.

Funkcja oceniająca została opracowana w oparciu o artykuł D. Hartmana "Notions of Evaluation Functions" z pracy [5] oraz klasyczny artykuł "Chess 4.5" z pracy [2]. Punkt wyjścia do konstrukcji f.o. programu "Joanna" stanowił wzór z rozdziału 7.4 z [5]. Oczywiście wprowadziłem do tego wzoru kilka własnych modyfikacji. Ponadto Hartman pominął wartościowanie struktury pionowej. Przepisałem więc ją z [2]. Z [2] wzięta jest także specjalna funkcja oceniająca wykorzystywana podczas matowania.

Funkcja oceniająca korzysta z własnych struktur danych opisujących własności pozycji z aktualnego stanu przeszukiwania. Struktury te pozwalają na szybsze obliczenie wyniku f.o. Przykładem tych struktur są listy zawierające pozycje wszystkich poszczególnych typów bierek. Struktury te modyfikowane są w trakcie przeszukiwania za pomocą dwóch funkcji: "MoveFunOcen" oraz "UnMoveFunOcen". Jedna z nich służy do modyfikacji struktur danych f.o. przy wykonaniu posunięcia, druga przy cofnięciu. Tak więc schemat modyfikowania struktur danych funkcji oceniającej odpowiada zasadzie działania metod inkrementalno-dekrementalnych (patrz. rozdz. 3.4).

Sam wynik f.o. jest liczbą całkowitą z zakresu <-32768 , +32767>. Jak wspomniano wcześniej f.o. składa się zasadniczo z dwóch części: oceny materialnej i oceny pozycyjnej. F.o. jest funkcją "symetryczną", tzn. w pozycjach powstałych przez zamianę figur białych na czarne, czarnych na białe i odbiciu symetrycznym wzdłuż linii środkowej szachownicy funkcja oceniająca zmienia jedynie znak.

Jak wspomniano w poprzednim rozdziale ocena pozycyjna obliczana jest w pozycjach końcowych obszaru pełnego przeszukiwania, natomiast ocena materialna jest zadaniem przeszukiwania selektywnego. Prowadzi to do powstania efektu, z którego opisem nie zetknąłem się w czytanej literaturze i dlatego postanowiłem nazwać go "efektem odkładania gróźb". Otóż "Joanna" odkłada możliwie jak najdalej posunięcia wygrywające materiał, o ile tylko możliwość ich wykonania nie wykracza poza zasięg jej przeszukiwania selektywnego. Na przykład jeżeli w końcówce pionowej "Joanna" ma możliwość promowania hetmana, ale może to odłożyć o jedno posunięcie, to najpierw wykona posunięcie poprawiające ocenę pozycyjną. Rzecz w tym, że promując hetmana jeszcze w rejonie pełnego przeszukiwania "Joanna" pozbawiłaby się dodatkowych punktów za zaawansowaną pozycję pionka i dlatego preferuje wykonanie promocji dopiero w obszarze selektywnym.

Prowadzi to czasem do podejmowania błędnych decyzji. Dla przykładu w pewnym rodzaju końcówek pionowych zwanym przez szachistów "wyścigami", jak najszybsze promowanie hetmana ma decydujące znaczenie.

W programie walczyłem z tym zjawiskiem modyfikując nieco funkcję oceniającą pozycyjnie, co będzie opisane dalej.

Zakładam że czytelnik tego rozdziału jest zaznajomiony z terminologią szachową oraz komputerowo-szachową.

4.3.1. Ocena materialna.

Ocena materialna jest sumą wartości materialnej znajdujących się na szachownicy bierek. Wartości poszczególnych bierek ustaliłem na:

pionek = 100 ,
skoczek = 290 ,
goniec = 310 ,
wieża = 500 ,
hetman = 950 ,
król = 20.000 .

Oczywiście wartość czarnych bierek należy brać z minusem. Wartość materialna króla wykorzystywana jest jedynie w sytuacji, gdy w trakcie przeszukiwania selektywnego "Joanna" natrafi na pozycję z możliwością zabicia króla w jednym posunięcia. "Joanna" wartościuje taką pozycję na 20.000 i następuje powrót sterowania.

Ponadto zgodnie z sugestią autorów pracy [2] wprowadziłem dodatkowe specjalne wartościowania liczby pionków na szachownicy oraz wartościowanie przewagi materialnej w stosunku do sumy pozostałego na szachownicy materiału. Celem tych wartościowań jest z jednej strony zachęcenie "Joanny" do wymian w sytuacji posiadania przewagi materialnej (unikania wymian w przeciwnym wypadku), z drugiej strony "zniechęcenie" do wymian pionków w tejże sytuacji, gdyż pionki stanowią wtedy potencjalne hetmany. Wartościowanie to wyraża się poniższymi wzorami.

(1) LP * 30000 / (LP + 1) * M
(2) MA * 590 / M

We wzorach tych poszczególne identyfikatory oznaczają:

LP - liczba pionków strony mającej przewagę materialną

M - suma wartości materiału na szachownicy, biorąc czarne bierki wyjątkowo z plusem, nie licząc króli.

MA - wartość przewagi materialnej (oczywiście z plusem).

Jeśli przewagę materialną mają białe, to wartości powyższe dodaje się do wyniku funkcji oceniającej materialnie, a jeśli przewagę mają czarne odejmuje.

Liczba 590 we wzorze (2) została tak dobrana, aby z jednej strony wartość wyrażenia była jak największa, a z drugiej strony aby w pozycji: "K+G+S":"K+p" strona silniejsza nie wymieniła skoczka na pionka przeciwnika.

Powstaje oczywiście pytanie, czy nie należałoby traktować tych własności jako pozycyjnych i włączyć raczej w ocenę pozycyjną. Jednak takie rozwiązanie wzmogłoby jeszcze "efekt opóźniania gróźb".

4.3.2. Ocena pozycyjna.

Klasyfikacja pozycji wyjściowej.

Jak wspomniano wcześniej autorzy pracy [2] słusznie postulowali wybór funkcji oceniającej z bazy takich funkcji w zależności od klasyfikacji pozycji wyjściowej. Również "Joanna" klasyfikuje pozycję wyjściową. Nie ma jednak żadnej bazy funkcji oceniających, a jedynie bezpośrednio w kodzie programu poszczególne składniki oceny pozycyjnej zależą od wyniku klasyfikacji.

"Joanna" przyjmuje podział możliwych pozycji na pięć rodzajów:

(1) debiut,
(2) gra środkowa,
(3) wczesna końcówka,
(4) końcówka,
(5) matowanie.

(od.1) Pozycja jest klasyfikowana jako debiut, jeżeli co najmniej czternaście bierek, nie licząc króli, znajduje się na swoich pozycjach wyjściowych.
(od.2) Pozycja jest klasyfikowana jako gra środkowa, jeżeli nie jest debiutem, oraz suma materiału na szachownicy, nie licząc króli i licząc czarne bierki na plus, wynosi co najmniej 4 300.
(od.3) Pozycja jest klasyfikowana jako "wczesna końcówka", jeśli na szachownicy jest pomiędzy 4 300 a 2 800 p. materiału.
(od.4) Pozycja jest klasyfikowana jako końcówka, jeżeli na szachownicy pozostało mniej niż 2 800 materiału.
(od.5) Pozycja jest klasyfikowana jako matowanie, jeżeli jedna ze stron osiągnęła przewagę materialną co najmniej 450, ma co najmniej jednego hetmana lub wieżę, ewentualnie jeśli ich nie ma, to nie ma przy tym ani jednego pionka. Ten ostatni warunek wynika z założenia, że mając jedynie lekkie figury i piony, łatwiej jest najpierw promować hetmana, a dopiero potem matować.

Zaznaczam, że matowanie jest fazą posiadającą własną, specyficzną ocenę pozycyjną. Omawiane dalej składowe oceny pozycyjnej nie odnoszą się do fazy matowania.

Ze względu na oszczędność czasu, "Joanna" dokonuje jedynie klasyfikacji pozycji wyjściowej. Bywa to przyczyną błędnych ocen pozycji końcowych przeszukanego drzewa, jeżeli na ścieżce przeszukiwania nastąpiła zupełna zmiana sytuacji na szachownicy.

Położenie poszczególnych bierek.

Większość składników oceny pozycyjnej to wartościowanie położenia poszczególnych figur względem środka szachownicy oraz względem innych bierek. Działanie funkcji oceniającej jest takie, że dla każdej bierki liczone jest wartościowanie jej położenia i wynik dodawany jest do pewnej zmiennej globalnej.

Przed przejściem do omówienia wartościowania położeń poszczególnych typów bierek podaję używane dalej dwa wzory.

Bliskość_Centrum([i,j]) = 7 - (| 3.5 - i | + | 3.5 - j |)

We wzorze powyższym i oraz j oznaczają współrzędne pola na szachownicy. W programie "Joanna" przyjmuje się, że współrzędne szachownicy są liczbami z przedziału < 0 , 7 >. Tak więc np. pole "c-6", to w programie "Joanna" pole "2-5". Jak widać powyższy wzór przyjmuje wartości z przedziału < 0 , 6 >.

Bliskość_Pól([i1,j1],[i2,j2]) = 7 - (| i1 - i2 | + | j1 - j2 |)

gdzie [i,j] oznaczają współrzędne pola, podobnie jak w podanym wyżej wzorze. Wzór na bliskość pól przyjmuje wartości z przedziału < -7 , 6 > (nie ma sensu obliczanie odległości dwóch identycznych pól).

pionki

Jak wiadomo "struktura pionowa jest duszą szachów". "Joanna" rozpoznaje następujące cechy struktury pionowej.

(1) Pionki izolowane,
(2) Pionki zdwojone,
(3) Pionki w centrum,
(4) Dochodzące pionki,
(5) Specjalna ocena debiutowa.

(od.1) Za każdego izolowanego białego pionka "Joanna" odejmuje wartość 20. Oczywiście w przypadku czarnych bierek "Joanna" postępuje przeciwnie niż w przypadku białych, tzn. np. za izolowanego czarnego pionka "Joanna" dodaje 20 do oceny pozycji. W dalszym ciągu nie będę tego zaznaczał podając jedynie wartość związaną z daną cechą.
(od.2) Za każdego zdublowanego pionka "Joanna" odejmuje wartość 10.
(od.3) W debiucie i grze środkowej dla każdego pionka "Joanna" wartościuje jego odległość od centrum szachownicy wg. wzoru:

Bliskość_Centrum([i,j])

Dodatkowo za każdego pionka znajdującego się na jednym z pól "d4", "d5", "e4", "e5" "Joanna" dolicza do oceny 15, a za pionki znajdujące się na polach "c4", "c5", "d3", "d6", "e3", "e6", "f4", "f5" dolicza 10.
(od.4) W końcówkach dla każdego pionka obliczane jest jego zawansowanie wg. wzoru:

WSPÓŁCZYNNIK * $(j - 1)^2$ dla białych pionków
WSPÓŁCZYNNIK * $(6 - j)^2$ dla czarnych pionków

gdzie j oznacza współrzędną poziomą pionka na szachownicy. Powyższa wartość dodawana jest do oceny pozycyjnej. WSPÓŁCZYNNIK przyjmuje wartość 2 we wczesnej końcówce i 4 w końcówce.

Ponadto "Joanna" oblicza liczbę dokonanych na aktualnej ścieżce promocji pionków i za każdą promocję dodaje/odejmuje do/od oceny wartość:

30 * WSPÓŁCZYNNIK

W przeciwnym razie "Joanna" odkładałaby możliwe jak najdalej promowanie pionków, aby nie stracić pozycyjnego zysku z pionka na siódmej linii ("efekt opóźniania gróźb").

(od.5) Jeżeli w debiucie, oba z pionków sprzed hetmana i króla nie zostały rozwinięte, to "Joanna" odejmuje od oceny 50. Ponadto dla obu tych pionków oddzielnie, jeżeli nie są rozwinięte, odliczane jest od oceny jeszcze 10. Rzecz w tym, że "Joanna" wartościuje rozwój debiutowy w oparciu o przedstawiony dalej wzór D.Levy'go. Jednak wzór Levy'go nic nie mówi o rozwoju pionków i dlatego programy wartościujące rozwój w oparciu o niego często rozpoczynają partię od wyprowadzenia obu skoczków.

skoczki

"Joanna" rozpoznaje tylko jedną własność położenia skoczka.

(1) Odległość od centrum.

Własność jest badana we wszystkich fazach gry (oprócz matowania oczywiście) i do oceny dodaje się wartość:

4 * Bliskość_Centrum

gońce

Dla gońców liczona jest podana przez Hartmana i nieco przeze mnie zmodyfikowana ocena położenia. Ocena ta wyraża się wzorem (zob. [5]):

ABB = EM + OM + DL + OW

gdzie:

EM : punkty za obecność bierek przeciwnika na diagonalach gońca: Skoczek=3, Goniec=4, Wieża=5, Hetman=9, Król=10

OM : punkty za obecność własnych bierek na diagonalach gońca: Skoczek=1, Goniec=1, Wieża=2, Hetman=3

DL : długość diagonali gońca (Liczba możliwych posunięć gońca z danego pola na pustej szachownicy) - 7

OW : punkty za własne pionki na diagonali gońca

Za każdego pionka z tyłu gońca: +1

Za każdego pionka z przodu gońca:

jeśli pionek jest na własnej połowie: -5

jeśli pionek jest na połowie przeciwnika: +1

Tak obliczona ocena związana z danym gońcem mnożona jest przez współczynnik i dodawana do oceny całkowitej. Współczynnik zależy od fazy gry i wynosi: 3 dla gry środkowej i 1 dla wczesnej końcówki. W końcówkach i w debiucie pozycja gońca nie jest wartościowana.

wieże

Dla wieży rozpatrywane są następujące elementy oceny pozycyjnej.

(1) Wieża na otwartej linii.
(2) Bliskość króla przeciwnika.
(3) Mobilność wieży.
(4) Wieże złączone.
(5) Wieża na siódmej linii.

(od.1) Dla każdej wieży na otwartej linii dodawane jest do oceny 10, a dla wieży na półotwartej linii 4. Własność nie jest brana pod uwagę w końcówkach.
(od.2) W grze środkowej i końcówce wartościowana jest odległość wieży od króla przeciwnika zgodnie ze wzorem:

2 * Bliskość_Pól (Wieża , Król_Wroga)

Wartość ta dodawana do oceny pozycji (oczywiście, wartość powyższa może być ujemna). Własność nie jest brana pod uwagę w debiucie.
(od.3) Mobilność wieży oblicza się ze wzoru:

2 * Liczba atakowanych przez wieżę pól

Wartość powyższa dodawana jest do oceny.
(od.4) Za każdą parę złączonych wież doliczane jest 20 punktów (oczywiście rzadko kiedy na szachownicy jest więcej niż jedna para wież).

(od.5) Dla wieży na siódmej linii dodaje się do oceny 27 punktów. Jeżeli wieża na siódmej linii odcina króla na ósmej to dodaje się jeszcze 13.

hetmany

Dla hetmanów rozpatrywane są następujące elementy oceny pozycyjnej.

(1) Hetman w centrum.
(2) Bliskość króla przeciwnika.

(od.1) Punkty za odległość od centrum liczone są według wzoru

WSPÓŁCZYNNIK * Bliskość_Centrum

i dodawane do oceny, przy czym w debiucie WSPÓŁCZYNNIK przyjmuje wartość -2, w grze środkowej 0, we wczesnej końcówce 2, i w końcówce 4.
(od.2) Odległość hetmana od króla przeciwnika liczona jest według wzoru

3 * Bliskość_Pól(Hetman , Król_przeciwnika)

Wartość powyższa dodawana jest do oceny w grze środkowej i w końcówce.

król

Rozpatrywane są następujące elementy oceny pozycji króla.

(1) Bezpieczeństwo króla.
(2) Centralizacja króla.

(od.1) Bezpieczeństwo króla jest bardzo ważnym czynnikiem w debiucie i w grze środkowej. Nie jest ono wartościowane w końcówkach, gdzie znaczenia nabiera centralizacja króla.

"Joanna" ocenia bezpieczeństwo króla na podstawie dwóch własności: położenia króla i obecności pionków przed królem. Niżej podane są wartości dodawane przez "Joannę" do oceny w zależności od położenia białego króla (dla czarnego króla należy brać przeciwne wartości dla przeciwległych pól).

b1,g1 : 25
b2,a1,g2,h1 : 15
c1,a2,h2 : 10

f1 : 5

d1 : -5

c2-f2 : - 40

a3-h3 : - 100

a4-h4 : - 200

pozostałe : - 400

Ponadto za każdego pionka znajdującego się na jednym z trzech pól bezpośrednio przed królem (lub dwóch, jeżeli król stoi na bandzie) "Joanna" dolicza do oceny 15.

Jeżeli jednak wszystkie trzy pola przed królem znajdującym się na pierwszej linii są zajęte przez pionki. to "Joanna" odlicza od oceny 45, niwelując punkty za obecność tych pionków. Chodzi o uniknięcie słabości pierwszej linii i wymuszenie na "Joannie" zrobienia królowi "furtki".

(od.2) W końcówkach miejsce wartościowania bezpieczeństwa króla zajmuje wartościowanie odległości od centrum szachownicy. Dodawana do oceny wartość obliczana jest według wzoru:

WSPÓŁCZYNNIK * Bliskość_Centrum

gdzie WSPÓŁCZYNNIK przyjmuje wartość 3 dla wczesnej końcówki i 8 dla końcówki.

Inne składniki oceny pozycyjnej.

Oprócz wartościowania pozycji poszczególnych bierek bada się także pewne ogólne własności pozycji. Są to: kontrola centrum, mobilność i wzór Levy'go. Wzór Levy'go służy wartościowaniu rozwoju figur w debiucie a także w grze środkowej. Natomiast kontrola centrum i mobilność figur stanowią bardzo dobrą heurystykę wartościowania pozycji bez odwoływania się do znanych z praktyki gry w szachy pojęć. Niestety, struktury danych "Joanny" nie pozwalają obliczać tych wartości precyzyjnie dostatecznie małym kosztem, stąd zmuszony byłem obliczenia tych wartości zmodyfikować. Dla przykładu, aby obliczyć liczbę atakowanych pól przez jedną ze stron "Joanna" odwołuje się do listy posunięć możliwych do wykonania w tej pozycji i oblicza liczbę różnych pól docelowych dla kolejnych ruchów. Jednak taki algorytm nie uwzględnia kontroli własnych bierek poprzez inne własne bierki, a jest to przecież bardzo ważna własność pozycji. W dodatku aby nie generować w pozycjach końcowych posunięć dla gracza nie będącego na posunięciu (zbyt duży koszt) "Joanna" bierze do oceny dla tego gracza wartości z pozycji o jeden półruch wstecz.

kontrola centrum

Za każdą kontrolę jednego z sześciu centralnych pól szachownicy: "c4", "c5", "d4", "d5", "e4", "e5", "f4", "f5" "Joanna" dolicza do oceny pozycji wartość 10.

mobilność

W skład oceny mobilności wchodzą dwie własności:

(1) Liczba posunięć każdej ze stron.
(2) Liczba kontrolowanych pól przez każdą ze stron.

(od.1) Za każde możliwe pseudoposunięcie "Joanna" dolicza do oceny pozycji wartość 3.
(od.2) Za każde kontrolowane pole na szachownicy "Joanna" dolicza do oceny pozycji wartość 3 .

wzór Levy'go

Wzór Levy'go ma za zadanie wartościować rozwój figur w debiucie. Warto obliczać go także w pozycjach klasyfikowanych jako gra środkowa. Wzór ten przedstawia się następująco.

20*D - (15*U + k*C)

gdzie:

D : liczba rozwiniętych lekkich figur;

U : 0 jeśli hetman stoi na wyjściowym polu lub nie ma go już na szachownicy, albo liczba nierozwiniętych skoczków, gońców i wież w przeciwnym przypadku.

C : 8 jeśli przeciwnik ma hetmana, albo liczba posiadanych przez przeciwnika skoczków, gońców i wież, w przeciwnym przypadku.

k : 0 jeśli roszada została wykonana;
3 , jeśli roszada nie została jeszcze wykonana, ale zostało zachowane prawo do obu roszad;
5 , jeśli roszada nie została jeszcze wykonana, ale zostało zachowane prawo do krótkiej roszady;
10 , jeśli roszada nie została jeszcze wykonana, ale zostało zachowane prawo do długiej roszady;
20 , jeśli roszada nie została wykonana, i zostało utracone prawo do obu roszad;

Wartość wzoru Levy'go liczona jest dla obu stron i dodawana do oceny pozycyjnej.

Praktyka potwierdza dużą przydatność wzoru dla debiutu. Postanowiłem jednak wprowadzić modyfikację tego wzoru dla gry środkowej. Otóż w grze środkowej wartość 15*U nie jest liczona. Składowa ta została tak pomyślana, aby jak najdłużej nie wykonywać posunięć hetmanem. Jednak w grze środkowej powinno się w końcu umożliwić jakieś jego rozwinięcie.

ocena pozycyjna przy matowaniu

W fazie matowania "Joanna" korzysta ze specjalnej funkcji oceniającej pozycyjnie. Funkcja ta nastawiona jest na stopniowe zapędzenie matowanego króla w róg oraz na trzymanie figur matujących możliwie blisko matowanego króla.

Odległość matowanego króla od centrum wartościowana jest wzorem:

9 * Bliskość_Centrum

Powyższa wartość jest dodawana do oceny na korzyść strony matowanej.

Ten prosty wzór wywołuje poprawne manewry ciężkich figur i gońców natomiast dla skoczków i króla strony matującej niezbędne jest przybliżenie ich do matowanego króla. Uzyskuje się to odejmując od oceny, na niekorzyść strony matowanej, wartość:

2 * Bliskość_Pól(Matująca_Bierka , Matowany_Król).

Ponadto za położenie króla strony matującej na jednej z linii "c", "f", "3", "5" odejmuje się do oceny n a niekorzyść strony matowanej wartość 2. Zdaniem autorów [2] ułatwia to matowanie króla w końcówce "K+G+G : K".

4.4. Tablica transpozycji.

Jak większość programów szachowych "Joanna" korzysta z tablicy transpozycji zwanej też czasem "podręcznikiem" ("cache", patrz rozdz. 3.5). Tablica transpozycji spełnia w programie kilka funkcji, przede wszystkim swoją tradycyjną funkcję zapamiętywania wyników przeszukiwania w celu uniknięcia ich powtórzenia przy przestawieniach posunięć prowadzących do identycznych pozycji. Ponadto rozpoznawanie powtarzających się pozycji z rozgrywanej aktualnie partii także odbywa się przez mechanizm tablic transpozycji. Na technice tablic transpozycji oparty jest przedstawiony w trzeciej części pracy algorytm "BeBe+".

"Dane techniczne" rekordu tablicy transpozycji

Rekord z tablicy transpozycji ma wielkość 6 słów 16-o bitowych. Niżej podany jest format tego rekordu. Kolejne słowa oznaczono w1, w2 itd. Bity w poszczególnych słowach numeruję od zera, więc ósmy bit w trzecim słowie oznaczono w3.7

Format rekordu w tablicy transpozycji jest następujący:

w1,w2 - posunięcie uznane za najlepsze
w3 - wynik minimaksu z przeszukiwania
w4.0-3 - flaga (patrz dalej)
w4.4-7 - głębokość regionu pełnego przeszukiwania
w4.14 - flaga określająca rekord algorytmu "BeBe+"
w4.15 - flaga określająca rekord "systemowy"
w5,w6 - klucz, wartość funkcji haszującej dla danej pozycji

Posunięcie uznane za najlepsze trzymane jest w formacie :

w1.0-3 - pole wyjściowe - linia pionowa
w1.4-7 - pole wyjściowe - linia pozioma
w1.8-11 - pole docelowe - linia pionowa
w1.12-15 - pole docelowe - linia pozioma
w2.0-3 - promowana bierka
w2.4-7 - bierka wykonująca posunięcie
w2.8-11 - bita bierka
w2.12 - czy posunięcie jest biciem w przelocie

Znaczenie poszczególnych pól jest przystające do pól rekordu definiującego posunięcie w programie "Joanna".

Pole w3 to wynik minimaksu z przeszukiwania. Dla węzłów wewnętrznych przeszukiwanych drzew jest on czasem jedynie górnym bądź dolnym ograniczeniem rzeczywistej wartości. Jest to skutek działania algorytmu Alfabeta w sytuacji, gdy wynik minimaksowy leżał poza inicjalnym przedziałem $< \alpha_0 , \beta_0 >$. Dla rekordów będących zapisem wyniku przeszukiwania, pole w4.0-3 ("flaga") przyjmuje jedną z trzech wartości: OGRANICZENIE_DOLNE, OGRANICZENIE_GÓRNE, DOKŁADNA_WARTOŚĆ. Inicjalnie pole to przyjmuje wartość "PUSTY REKORD". Jeżeli rekord pełni pewną funkcję w programie inną niż rekordu transpozycji, na przykład służy ocenianiu powtarzających się w aktualnej partii pozycji jako remisów (pole w3 ma wtedy wartość 0), to flaga przyjmuje wartość określającą do czego dany rekord służy.

Pole w4.4-7 zawiera głębokość przeszukanego poddrzewa. Chodzi tu o głębokość regionu pełnego przeszukiwania. Jak widać narazie "Joanna" zakłada, że głębokość przeszukanego drzewa nie przekroczy piętnastu.

Pole w4.14 określa rekord algorytmu "BeBe+". Rekord taki jest traktowany w specjalny sposób. "Zwykłe" rekordy nie mogą nadpisać rekordów "BeBe+", natomiast zapis rekordów "BeBe+" musi zakończyć się sukcesem.

Jeśli pole w4.15 przyjmuje wartość 1, to rekord jest traktowany jako "systemowy". Rekordy "systemowe" są traktowane analogicznie jak rekordy "BeBe+", chociaż pełnią inne funkcje w programie.

Pola w4.8-13 nie są wykorzystywane.

Na polach w5 i w6 trzymany jest 32-u bitowy klucz będący wynikiem funkcji haszującej dla danej pozycji (w polu w6 trzymane są starsze bity). Prawdopodobieństwo powtórzenia się klucza dla dwóch różnych pozycji wynosi 1/4.294.967.294 .

Rozmiar, adresowanie i problem kolizji.

Rozmiar tablicy transpozycji zależy od wielkości dostępnej pamięci. Jest on zawsze potęgą dwójki. Ułatwia to adresowanie. Jako adres w tablicy brane jest obcięcie 32-bitowego klucza danej pozycji do log N początkowych bitów klucza, gdzie N jest rozmiarem tablicy. W praktyce, dla programu chodzącego w systemie MS DOS bez dostępu do pamięci rozszerzonej, rozmiar tablicy transpozycji wynosi 2^{12} (4096) rekordów.

Można mówić o trzech poziomach kolizji w programie "Joanna". Pierwszy polega na odwzorowaniu dwóch różnych kluczy w to samo miejsce tablicy na skutek obcięcia bitów klucza przy adresowaniu. Ta kolizja jest dosyć częsta w trakcie działania programu (im mniejszy rozmiar tablicy, tym częstsze kolizje), ale można ją bardzo łatwo wykryć porównując klucz danej pozycji z zapisanym w rekordzie (dokładnie po to umieszcza się w rekordzie tablicy klucz pozycji). Należy przyjąć jakąś strategię postępowania w przypadku kolizji przy zapisywaniu rekordu. W "Joannie" przyjęto, że nowy rekord nadpisuje poprzedni, o ile ten nie jest rekordem "systemowym" (patrz dalej). W przypadku próby zapisania rekordu związanego tą samą co rekord w tablicy pozycją, decyduje wysokość przeszukanego poddrzewa (pole w4.4-7). Im wyższe poddrzewo, tym wyższy priorytet.

Drugi poziom kolizji polega na odwzorowaniu dwóch pozycji w ten sam klucz 32-u bitowy. Zdarza się to raz na 4 294 967 294 razy. Najczęściej taka kolizja jest wykrywane przez program, gdy zapisane w rekordzie posunięcie okazuje się niemożliwe do wykonania w wyjściowej pozycji.

Wreszcie trzeci poziom kolizji to ten, w którym wystąpił drugi poziom kolizji, a zapisane w tablicy posunięciu okazało się możliwe. Trzeba jednak zaznaczyć, że nawet w tym wypadku konsekwencje kolizji niekoniecznie muszą okazać się katastrofalne, poprostu "Joanna" przyjmie dane z tablicy za miarodajne i podejmie przypadkową decyzję. Jeśli owa przypadkowa decyzja zostanie podjęta gdzieś głęboko w przeszukiwanym drzewie na niepryncypialnej ścieżce, to kolizja może pozostać niezauważona.

Jak wspomniano wcześniej dla rekordów "systemowych" wymaga się pozytywnego zakończenia operacji zapisu. Trzeba zatem rozwiązać oddzielnie przypadek kolizji rekordów systemowych. "Joanna" trzyma

dodatkową tablicę 50 rekordów z przeznaczeniem na wypadek kolizji rekordów systemowych. W przypadku kolizji zapisu rekordów systemowych zapisywane jest pierwsze wolne miejsce w tablicy dodatkowej. W przypadku kolizji przy odczycie, tablica ta przeglądana jest liniowo w poszukiwaniu właściwego rekordu.

Funkcja haszująca obliczana jest według metody Zorbista opisanej w punkcie 3.5. Dla każdej możliwej konfiguracji (bierka x pole) wygenerowana została pewna losowa 32-u bitowa liczba. Każde dwie z tych 12 * 64 = 768 liczb są różne. Oczywiście dla pustego pola brane jest zero.

Klucz danej pozycji to totalny XOR wszystkich liczb związanych z położeniem bierek szachownicy. W trakcie przeszukiwania klucz kolejnych badanych pozycji obliczany jest zgodnie z zasadą działania metod inkrementalno-dekrementalnych, co było opisane w rozdziałach 3.4. i 3.5.

Funkcje tablicy transpozycji.

Podstawową funkcją tablicy transpozycji jest zapisywanie wyników z przeszukiwań w celu uniknięcia ich powtórnego wykonania na wypadek ewentualnych transpozycji, to jest powtórzeń pozycji.

Przy odczycie dane z rekordu tablicy transpozycji mogą zastąpić wykonanie przeszukiwania, zawęzić przedział $< \alpha_0, \beta_0 >$, lub przynajmniej wskazać obiecujące posunięcie.

Z mechanizmu tablic transpozycji korzysta także wartościowanie z wynikiem zero powtarzających się w aktualnie rozgrywanej partii pozycji. Wszystkie pozycje w grze zapisywane są do tablicy transpozycji jako rekordy "systemowe" z flagą "REKORD_Z_PARTII" i wynikiem przeszukiwania zero. Flaga "REKORD_Z_PARTII" traktowana jest identycznie jak "DOKŁADNA_WARTOŚĆ". Po wykonaniu w partii posunięcia pionkiem lub zmianie sytuacji materialnej na szachownicy wszystkie rekordy systemowe z flagą "REKORD_Z_PARTII" są usuwane z tablicy, gdyż powtórzenie zapisanej w nich pozycji staje się niemożliwe.

Trzecią funkcją tablicy transpozycji jest obsługa algorytmu "BeBe+". Ta funkcja zostanie opisana w trzeciej części pracy.

4.5. Interfejs użytkownika.

"Joanna" ma wzorowany na standardzie SAA/CUA obsłudze anglojęzyczny interfejs użytkownika. Interfejs ten został napisany przy użyciu biblioteki graficznej "ViDE".

menu

Menu programu "Joanna" ma następującą strukturę.

File
- New
- Load
- Save
- Exit

Options
- Play
- Learn
- Screen

BeBePlus
- Merge swapped data
- Autoplay

About

(1) Kolejne opcje podmenu "File" oznaczają:

- New: zakończenie aktualnie rozgrywanej partii i rozpoczęcie nowej;
- Load: wgranie partii zapisanej na dysku;
- Save: zapisanie partii na dysku;
- Exit: wyjście z programu.

(2) Kolejne opcje podmenu "Options" oznaczają:

- Play: wybór opcji "Play" powoduje otwarcie formatki do wprowadzenia parametrów programu. Są tylko dwa parametry.
(a) Kolor bierek, którymi gra program. Opcja ta może być także zmieniana w trakcie rozgrywki.
(b) Limit czasu na jedno posunięcie w sekundach. Można wprowadzić dowolną wartość z przedziału < 1 , 250 >. Trzeba jednak pamiętać, że nawet przy limicie 1sek. na posunięcie "Joanna" musi wykonać minimalne przeszukiwanie drzewa, co w skomplikowanych pozycjach i na komputerze o małej mocy może trwać nawet do jednej minuty.

- Learn : wybór opcji "Learn" pozwala ustawić status funkcjonowania algorytmu uczenia się "BeBe+" (algorytm "BeBe+" zostanie omówiony w trzeciej części programu). Algorytm "BeBe+" może funkcjonować na cztery sposoby:

(a) W ogóle nie funkcjonować (OFF).
(b) Przesyłać jedynie dane z bazy danych do programu (Only LTM to STM).
(c) Przesyłać dane w przeciwnym kierunku co w poprzednim punkcie. Innymi słowy program w tym stanie pozwalałby jedynie gromadzić doświadczenie, natomiast nie korzystałby z doświadczenia już zgromadzonego (Only STM to LTM).
(d) Funkcjonować "w obie strony", tzn. zarówno korzystać z już zdobytego doświadczenia, jak i gromadzić doświadczenie nabywane w trakcie działania programu (ON).

- Screen : W trakcie namysłu nad posunięciem program wyświetla informację o aktualnej sytuacji w procesie przeszukiwania. Ponieważ wyświetlanie tej informacji na ekranie zmniejsza prawie dwukrotnie prędkość przeszukiwania, więc można zamienić taką "pełną" informację na krótką informującą tylko o tym, że program jest właśnie w stanie "namysłu".

(3) Podmenu "BeBePlus" ma dwie opcje, obie są związane z algorytmem "BeBe+".
- Autoplay : Po wybraniu tej opcji program przełączy status algorytmu "BeBe+" na "ON" i wykona serię dziesięciu ruchów począwszy z aktualnej pozycji. Czas do namysłu nad każdym z tych ruchów będzie taki, jak nastawiono w opcji <Options><Play> ale będzie on wynosić conajmniej 10 sek. Po wykonaniu serii "Joanna" połączy utworzone pliki tymczasowe (patrz rozdz. 5.2.) z bazą danych algorytmu "BeBe+".
- Merge swapped data : Łączy zawartość zapisanych na dysku plików tymczasowych z bazą danych "BeBe+".

(4) Wybór opcji "About" powoduje wyświetlenie krótkiej informacji o programie.

rozgrywka

Rozgrywka, polega na naprzemiennym wykonywaniu posunięć przez "Joannę" i użytkownika. Wykonanie posunięcia przez użytkownika polega na wskazaniu za pomocą myszki pola wyjściowego (zostaje one zaznaczone na ekranie obwódką) i następnie docelowego. Następuje wykonanie posunięcia użytkownika i "Joanna" przystępuje do "namysłu" nad swoim posunięciem. Następnie wykonuje go na szachownicy, następuje kolej użytkownika itd.

"Joanna" rozpoznaje w trakcie rozgrywki sytuacje mata, pata, trzykrotnego powtórzenia posunięć oraz pięćdziesięciu posunięć bez ruchu pionkiem i bicia bierki. W przypadku wystąpienia jednej z nich "Joanna" wyświetla odnośny komunikat i kończy aktualną partię. Nową można rozpocząć wybierając z menu opcję <"File"><"New">.

"gorące" klawisze

Jedyny "gorący klawisz" to <Ctrl>-<Escape> kończący natychmiast działanie programu. Klawisz ten jest szczególnie przydatny w sytuacji, gdy program uruchomiono przy nie zainstalowanej w systemie myszy.

minimalne wymagania wobec sprzętu

Program został napisany na komputery klasy IBM PC z systemem operacyjnym MS DOS. Minimalne wymagania programu wobec sprzętu to: procesor co najmniej 80286, minimum 640 K pamięci operacyjnej, karta graficzna co najmniej EGA, oraz mysz.

Uwaga! Program w ogóle nie wykorzystuje klawiatury, obsługa programu bez myszy nie jest możliwa.

5. Algorytm BeBe+.

Algorytm "BeBe+" jest moim samodzielnym opracowaniem. Idea została zaczerpnięta od autorów znanego programu "BeBe" i stąd taka a inna nazwa algorytmu.

Algorytm "BeBe+" należy zaklasyfikować jako algorytm uczenia się na podstawie własnego doświadczenia. Termin ten wymaga jednak konkretniejszego wyjaśnienia. Otóż "BeBe+" ma za zadanie zapamiętywać wyniki wszystkich dokonywanych przez program przeszukiwań, zapisywać je w "pamięci trwałej" i przywoływać te wyniki w miejsce powtórzeń tych samych przeszukiwań.

Algorytm "BeBe+" funkcjonuje razem z mechanizmami iteracyjnego pogłębiania przeszukiwania i tablicy transpozycji. Zastosowanie algorytmu wymaga zrezygnowania z dynamicznych metod porządkowania posunięć.

5.1. Algorytm BeBe.

Algorytm BeBe+ jest rozwinięciem algorytmu uczenia się na podstawie własnego doświadczenia zwanego "BeBe", opisanego w [6] na str.197-216 i zaimplementowanego w programie o tej samej nazwie.

Ideą algorytmu BeBe było zapamiętywanie wyników przeszukiwań dla pozycji pojawiających się w rozgrywanych przez "BeBe" partiach. Wyniki kolejnych przeszukiwań "BeBe" zapisywała na pliku w rekordach o formacie zbliżonym do formatu rekordów z tablicy transpozycji. Wielkość pliku była ograniczona, tak że pojawienie się nowej pozycji (zamiast mówić o rekordach związanych z daną pozycją będę mówił poprostu o pozycji) powodowało nadpisanie innej. Przed rozpoczęciem każdego przeszukiwania odbywała się transmisja zawartości pliku do tablicy transpozycji.

Działanie algorytmu "BeBe" można przyrównać do funkcjonowania pamięci LTM (Long Term Memory) i STM (Short Term Memory). Funkcję LTM pełni plik dyskowy a STM tablica transpozycji. (Odnośnie pamięci LTM i STM patrz [2] str.34-53.)

Jak pokazali autorzy programu "BeBe" przy serii 60-u gier przeciwko temu samemu przeciwnikowi (w prezentowanym przez autorów teście przeciwnikiem "BeBe" był inny program szachowy) "siła" programu wzrosła o około 150 p. ELO. Jednak dalszy wzrost siły gry nie był obserwowany. W każdym jednak razie niewątpliwą korzyścią ze stosowania algorytmu jest to, że program na ogół nie powtarza dwukrotnie tych samych - przegranych partii.

5.1.1. Założenia do algorytmu "BeBe+".

Analiza algorytmu "BeBe" nasunęła mi ideę jego rozwinięcia. Za cel postawiłem trzy następujące postulaty:

(1) Nie należy ograniczać "pojemności" LTM. Należy stworzyć bazę danych z mechanizmami błyskawicznego dostępu do poszukiwanych pozycji. Rekord, który raz pojawi się w bazie pozostaje w niej "na

zawsze", może co najwyżej zostać zastąpiony rekordem z "dokładniejszą" informacją o tej samej pozycji, tzn. pochodzącą z przeszukiwania "wyższego" drzewa.

(2) Autorzy "BeBe" przegrywali całą LTM do STM. Przy moim rozwiązaniu z LTM do STM wgrywane są jedynie rekordy z informacją o pozycjach znajdujących się w "otoczeniu" rozpatrywanej pozycji (chodzi o transmisję tylko tych pozycji, które mogą pojawić się w trakcie przeszukiwania). Należy zatem przechowywać w LTM informację o "otoczeniach" poszczególnych pozycji.

(3) "Bebe" zgrywał z STM do LTM jedynie rekord związany z korzeniem przeszukanego drzewa. Przy założeniach (1) i (2) naturalnym staje się przegranie z STM do LTM także węzłów wewnętrznych przeszukanego drzewa i równocześnie zapamiętanie, że węzły te stanowią otoczenie rozpatrzonego korzenia.

Niewątpliwie założenia powyższe nakładają dosyć ambitne wymagania. Przede wszystkim, ponieważ w szachy gra się z ograniczonym czasem do namysłu (na ogół ok. trzy minuty na posunięcie), niemożliwa jest transmisja z STM do LTM tak dużej ilości informacji przy zachowaniu odpowiedniej struktury bazy danych. Wymyśliłem więc poniższe rozwiązanie.

W trakcie partii dokonuje się jedynie transmisja z LTM do STM przed rozpoczęciem każdego przeszukiwania, przy czym dokonywana jest jedynie transmisja rekordów z "otoczenia" danej pozycji. Po zakończeniu każdego przeszukiwania interesująca nas część STM zapisywana jest na plik roboczy. Po zakończeniu partii na życzenie operatora można dokonać złączenia danych z pliku roboczego z LTM.

Odmiennym problemem jest zaprojektowanie dostatecznie szybkiej transmisji z LTM do STM "otoczenia" rozpatrywanej pozycji. Zostało to rozwiązane dzięki opisanej dalej odpowiedniej strukturze bazy danych LTM.

Dzięki omówionemu dalej algorytmowi udało mi się uzyskać efekt dokładnego odtwarzania sytuacji, w której poprzednio przerwane zostało iteracyjnie pogłębiane przeszukiwanie. Oznacza to, że jeśli "Joanna" natrafi w pewnej rozgrywanej partii na pozycję, którą rozpatrywała już wcześniej, to odtworzy sytuację w której przeszukiwanie zostało poprzednio przerwane i będzie kontynuować dalej od tego punktu.

5.2. Struktura plików LTM dla algorytmu BeBe+.

Aby mieć pełną kontrolę nad działaniem bazy danych postanowiłem sam zaprogramować obsługę LTM korzystając jedynie z języka C i systemu plików systemu MS DOS.

W skład bazy danych algorytmu BeBe+, stanowiącej LTM algorytmu "BeBe+", wchodzi plik "tt.dat" wraz z kluczem "tt.key" oraz rodzina plików "KLUCZ.ltm". Rodzina plików została zaimplementowana jako katalog plików. Poszczególne pliki "KLUCZ.ltm" zawierają informację o otoczeniu pozycji o kluczu "KLUCZ". Ponieważ w programie "Joanna" klucz pozycji jest liczbą 32-bitową, więc przyjąłem, że "KLUCZ" jest poprostu szesnastkowym zapisem tej liczby, np. "118fa2g4.ltm".

Zakłada się istnienie dysku twardego o nazwie "C:". LTM przechowywana jest w następującej strukturze plików:

C:\BEBEPLUS.DBS\

tt.dat,

tt.key,

LTM.DAT\

klucz1.ltm,

klucz2.ltm,

...

Dalej omawiane są poszczególne pliki wchodzące w skład struktury LTM.

(1) tt.dat

Plik "tt.dat" zawiera rekordy przepisane z tablicy transpozycji. Format rekordu pliku "tt.dat" jest identyczny z formatem rekordu tablicy transpozycji.

(2) tt.key

Jest to klucz pliku "tt.dat". Rekord pliku "tt.key" zawiera dwa pola: klucz i adres.

Pole "klucz" oznacza klucz pozycji, natomiast pole "adres" jest fizycznym adresem rekordu z pliku "tt.dat" o kluczu "klucz".

Plik "tt.key" jest fizycznie posortowany względem pola klucz. Umożliwia to bardzo szybkie odnalezienie algorytmem wyszukiwania słownikowego rekordu o zadanym kluczu.

(3) "KLUCZ.ltm"

Pliki "KLUCZ.ltm" zawierają informację o otoczeniach pozycji, które kiedykolwiek pojawiły się w partiach "Joanny". Jak wspomniano wcześniej "KLUCZ" oznacza szesnastkowy zapis klucza pozycji, której dotyczy dany plik.

Kolejne rekordy pliku to fizyczne adresy rekordów z pliku "tt.dat". Rekordy w plikach ".ltm" są fizycznie posortowane dla ułatwienia transmisji otoczenia pozycji o kluczu "KLUCZ" do STM.

5.3. Algorytm "BeBe+".

Algorytm BeBe+ polega na wymianie informacji pomiędzy LTM (bazą danych na dysku) a STM (tablicą transpozycji w pamięci).

"Joanna" korzysta z iteracyjnie pogłębianej Alfabety. Jeśli "Joanna" rozgrywając pewną partię natrafi na pozycję, którą rozpatrywała już wcześniej, to algorytm BeBe+ pozwali odtworzyć dokładnie sytuację, w której poprzednio zakończyło się przeszukiwanie. Dalsze przeszukiwanie kontynuowane jest od tego punktu. Dalej pokazane zostanie, że zużycie pamięci dyskowej na potrzeby algorytmu "BeBe+" jest proporcjonalne do

iloczynu liczby posunięć w rozegranych kiedykolwiek przez program partiach i logarytmu z wielkości przeszukanej przestrzeni stanów.

Algorytm składa się z trzech części: transmisji danych z LTM do STM, transmisji w przeciwnym kierunku oraz obsługi algorytmu BeBe+ w trakcie przeszukiwania przez manipulację rekordami w tablicy transpozycji.

Jak wspomniano wcześniej transmisja z STM do LTM nie odbywa się bezpośrednio, ale z wykorzystaniem plików przejściowych, których zawartość łączona jest z LTM po zakończeniu rozgrywania przez "Joannę" partii.

5.3.1. Obsługa tablicy transpozycji przez algorytm BeBe+.

Pole w4.14 z rekordu w tablicy transpozycji (patrz rozdz. 5.4.) zostało przeznaczone na flagę określającą, że dany rekord jest wykorzystywany przez algorytm BeBe+. Rekord z zapaloną flagą BeBe+ nie może zostać nadpisany przez "zwykłe" rekordy. W dodatku zapis rekordu z flagą BeBe+ musi się zawsze zakończyć pomyślnie. Ze względu na niebezpieczeństwo kolizji wprowadziłem dodatkową tablicę przeznaczoną dla rekordów BeBe+ obsługiwaną analogicznie jak opisana w rozdz. 4.4. tablica rekordów systemowych.

Weźmy pewien algorytm przeszukiwania "Przeszukiwanie". Parametrami algorytmu niech będą: m - głębokość przeszukiwania oraz P - pozycja. Obsługa rekordów BeBe+ w trakcie przeszukiwania odbywa się według podanego niżej schematu.

```
(* Przeszukiwanie do głębokości m związane z pozycją P *)

procedure Przeszukiwanie( P : pozycja , m : integer );
begin
    if m = 0 then
        przeszukiwanie := ocena( P );
    else
        begin
            Lista := Potomkowie( P );
            while  JestNierozpatrzonaPozycja( Lista )
                begin
                    P' := Kolejna_Pozycja( Lista );
                    Przeszukiwanie( P', m-1 );
                end ;
            if m > 1
                KasujFlagęBeBe+DlaListyPozycji( Lista );
            ZaznaczFlagęBeBePlusDlaPozycji( P );
        end
end;
```

Uwaga! Przed rozpoczęciem przeszukiwania wygaszane są flagi algorytmu BeBe+ we wszystkich rekordach tablicy transpozycji.

5.3.2. Transmisja danych z LTM do STM.

Przed rozpoczęciem każdego przeszukiwania "Joanna" sprawdza, czy LTM zawiera już informację o danej pozycji. Jeśli tak, to do STM ładowana jest informacja pozwalająca odtworzyć sytuację, w której poprzednio przerwano iteracyjnie pogłębiane przeszukiwanie.

Schemat algorytmu zgrywania zawartości LTM do STM przedstawia się następująco.

```
...
k := klucz( Pozycja );
p := plik_z_informacją_o_otoczeniu_pozycji_o_kluczu( k );
if p <> NIL then
    for all ( adres : adres w LTM odczytany z pliku "p" )
        ładuj_z_LTM_do_STM_rekord( adres );
...
```

Formaty rekordów w pliku "tt.dat" i rekordu w tablicy transpozycji są identyczne dzięki czemu transmisja z LTM do STM jest bardzo prosta, a co ważniejsze szybka.

5.3.3. Transmisja danych z STM do LTM.

Jak wspomniano wcześniej, niemożliwe jest złączenie zawartości STM z LTM w trakcie rozgrywania partii. Dlatego podczas partii zawartość STM po każdym przeszukiwaniu zapisywana jest na pliku roboczym. Oczywiście zapisywane są jedynie rekordy z flagą BeBe+.

Kolejne pliki robocze nazywane są poprostu "0.dat", "1.dat", "2.dat"... Pliki te umieszczane są w katalogu:

C:\BEBEPLUS.DBS\BEBESWAP.DAT\

Pierwszym rekordem w pliku roboczym jest zawsze rekord korzenia przeszukanego drzewa. Jak nietrudno zauważyć rekord ten ma w trakcie przeszukiwania zawsze zapaloną flagę BeBe+. Zmienia się jedynie jego zawartość po zakończeniach kolejnych iteracji.

Zapisanie tego rekordu na początku pliku roboczego w prosty sposób określa pozycję, której dany plik dotyczy.

Zapisywanie danych na plik roboczy odbywa się według schematu:

```
Przeszukiwanie( P );
k := klucz( P );
p := kolejny_plik_roboczy;
r := rekord_z_STM( k );
Zapisz( p, r );
for all( r : r - rekord z STM , r jest rekordem BeBe+ ) do
    Zapisz( p , r );
```

Zasadniczą częścią algorytmu jest scalanie plików roboczych z LTM. Scalanie odbywa się kolejno dla pojedynczych plików roboczych.

Niżej podany jest algorytm scalania oraz omówienie wchodzących w jego skład procedur, o ile z samej nazwy nie wynika jasno ich przeznaczenie.

function Klucz(r : Rekord): KluczPozycji ;

Typ "Rekord" oznacza rekord w formacie z tablicy transpozycji. Funkcja zwraca klucz rekordu pozycji

function PlikTypuOtoczenie(k : KluczPozycji) : Plik ;

Funkcja zwraca w wyniku plik zawierający informację o otoczeniu pozycji o kluczu k (patrz rozdz. 5.2. p..3).

function PierwszyRekord(p : Plik) : Rekord ;

Zwraca w wyniku pierwszy rekord z pliku p.

function IstniejeRekord(p : Plik , k : Klucz) : Typ_Logiczny ;

W założeniu parametrem p może być jedynie plik główny LTM, jednak dla elegancji zapisu podaję go jako parametr funkcji. Funkcja stwierdza czy w pliku głównym znajduje się rekord o kluczu k. Odszukanie rekordu odbywa się oczywiście poprzez użycie klucza pliku głównego ("tt.key") algorytmem wyszukiwania słownikowego, a zatem w koszcie log log n, gdzie n jest liczbą rekordów w pliku p.

procedure Porównaj_i_Zapisz(p : Plik ; r : Rekord);

Procedura zakłada, że w pliku p istnieje rekord o identycznym kluczu, co klucz z rekordu r. Procedura porównuje zawartości rekordu r i rekordu z pliku p i jeśli rekord r zawiera dokładniejszą informację o pozycji, to zapisuje go w miejscu poprzedniego. podobnie jak wyżej parametrem p może być jedynie plik "tt.dat".

function Adres(p : Plik ; k : Klucz) : Adres_Plikowy ;

Funkcja zwraca adres rekordu o kluczu k w pliku p. Parametr p tak samo jak wyżej.

procedure ScalajKluczPlikuGłównegoZTablicą(T) ;

Po dopisaniu do pliku głównego nowych rekordów należy uaktualnić zawartość klucza tego pliku (patrz rozdz. 5.2. p.2). Ponieważ klucz pliku głównego jest zawsze posortowany, więc uaktualnienie zawartości wymaga jedynie posortowania nowo dodanych rekordów w tablicy i scalenia tej tablicy z plikiem zawierającym klucz.

```
(* Algorytm scalania pliku roboczego z LTM *)

VAR
    p1,p2,p3 : Plik ;
    k : KluczPozycji ;
    T1,T2 : tablica typu - Adres_Plikowy ;

p1 := OtwórzPlik( CZYTANIE , KolejnyPlikRoboczy ) ;
p2 := OtwórzPlik( CZYTANIE+PISANIE , PlikGłównyLTM ) ;
k := Klucz( PierwszyRekord( p1 ) );
p3 := OtwórzPlik( PISANIE , PlikTypuOtoczenie( k ) ) ;

for all ( r : r jest rekordem z pliku p1 ) do
    begin
        if IstniejeRekord( p2 , Klucz( r ) );
            Porównaj_i_Zapisz( p2 , r );
        else
            begin
                DopiszNaKońcuPliku( p2 , r );
                DodajDoTablicy( Adres_Ostatniego_Rekordu( p2 ), T1 );
            end ;
        DodajDoTablicy( adres( p2 , Klucz( r ) ) , T2 );
    end;
Sortuj( T1 );
Sortuj( T2 );
ZapiszTablicęRekordówNaPlik( p3 , T2 );
ZamknijPliki( p1 , p2 , p3 );
ScalajKluczPlikuGłównegoZTablicą( T1 );
```

5.3.4. Analiza algorytmu BeBe+.

Przyjmijmy dla uproszczenia, że dla każdej pozycji w szachach liczba możliwych posunięć jest stała. W podanych dalej wzorach kolejne litery mają następujące znaczenia:

K: liczba pozycji, które program kiedykolwiek od początku pracy algorytmu BeBe+ rozpatrywał jako pozycje wyjściowe przeszukiwania,
L: liczba rekordów w pliku "tt.dat",
M: stopień drzewa,
N: wysokość przeszukiwanego drzewa,
T: rozmiar pliku tymczasowego.

Koszt algorytmu BeBe+ trzeba podać dla poszczególnych jego, omówionych wcześniej, składowych z rozbiciem na dwa kryteria: czas i zajętość pamięci dyskowej.

(1) Koszt pamięciowy zapamiętania otoczenia pozycji.

Po pierwsze przed rozpoczęciem przeszukiwania kasowane są wszystkie flagi bebeplus w tablicy transpozycji.

Po drugie przy każdym powrocie przeszukiwania do poziomu N kasowane są wszystkie flagi potomków właśnie rozpatrzonego węzła (patrz rozdz.5.3.3).

Zatem w dowolnym momencie przeszukiwania flagę bebe+ mogą mieć ustawione jedynie rekordy znajdujące się na aktualnej ścieżce przeszukiwania oraz bracia tych rekordów.

Stąd można oszacować liczbę rekordów w tablicy transpozycji z zapaloną flagą BeBe+ przez:

$KP_1 <= M * N$

Jest to tym samym oszacowanie rozmiaru pliku zawierającego informację o otoczeniu pozycji.

Trzeba przy tym zwrócić uwagę na następujący fakt. Otóż powyższe oszacowanie jest prawdziwe tylko wtedy, gdy program rozpatruje posunięcia w kolejności zależnej jedynie od "statycznych" własności rozpatrywanych pozycji. Dla przykładu, omawiana wcześniej technika "posunięć-morderców" jest techniką dynamicznego porządkowania rozpatrywanych posunięć. Kolejność rozpatrywania zależy tam od wcześniejszego przebiegu przeszukiwania drzewa. Jednak przy dynamicznym porządkowaniu posunięć "Joanna" po wczytaniu z LTM danych o otoczeniu mogłaby na skutek innej zawartości tablicy "posunięć-morderców" rozpocząć przeszukiwanie w zupełnie innym "kierunku" i w efekcie liczba rekordów z flagą "BeBe+" w tablicy transpozycji zaczęłaby się sumować przy każdym kolejnym przeszukiwaniu.

Z tego właśnie powodu "Joanna" korzysta z tablicy "posunięć morderców" jedynie przy wyłączonym algorytmie "BeBe+" (patrz rozdz. 4.1. p.(b)).

(2) Koszt pamięciowy zapamiętania danych w LTM.

Z poprzedniego punktu wprost wynika, że liczba rekordów pamiętanych zarówno w pliku głównym LTM ("tt.dat"), jak i suma zajętości wszystkich plików z otoczeniami jest ograniczona przez:

$KP_2 <= M * N * K$

gdzie K oznacza liczbę rozpatrywanych kiedykolwiek pozycji a c jest pewną stałą.

A zatem, jak wspomniano wcześniej, zużycie pamięci dyskowej na bazę danych algorytmu "BeBe+" (LTM) jest proporcjonalne do iloczynu logarytmu z wielkości przeszukanej przestrzeni stanów i liczby rozpatrzonych pozycji wyjściowych (M jest stałą). Wynika to wprost z powyższego wzoru.

(3) Koszt czasowy transmisji danych z LTM do STM.

Ponieważ w systemie MS DOS odnalezienie pliku w katalogu odbywa się algorytmem wyszukiwania liniowego, więc koszt tego odnalezienia jest liniowy względem K. Warto zaznaczyć, że możnaby sprowadzić wyszukiwanie pliku z otoczeniem do wyszukiwania słownikowego i tym samym zmniejszyć koszt do log log K. Następnie dokonywana jest transmisja kolejnych rekordów z LTM na podstawie ich fizycznych adresów. Całość kosztu czasowego transmisji z LTM do STM wynosi zatem:

$KC_1 <= K + M * N$

operacji dyskowych. Przez operacje dyskowe rozumiem tutaj swobodny dostęp do rekordu według adresu oraz operację czytania.

(4) Koszt czasowy transmisji z STM do LTM.

Koszt transmisji danych z STM na plik tymczasowy jest oczywiście liniowy względem wielkości tablicy transpozycji. Trudniejszy do obliczenia jest koszt scalania jednego pliku tymczasowego z LTM.

Dla każdego rekordu z pliku tymczasowego trzeba odnaleźć jego pozycję w pliku "tt.dat" o ile oczywiście rekord ten jest już w pliku "tt.dat". W wyszukiwaniu tym korzysta się z posortowanego indeksu "tt.key". Ponieważ rozkład kluczy pozycji jest jednostajny, więc algorytm wyszukiwania słownikowego rekordu w indeksie ma koszt log log L.

Po dopisaniu nowych rekordów do pliku tt.dat konieczne jest odbudowanie klucza. Dokonuje się to przez scalenie pliku "tt.key" z posortowaną względem klucza pozycji tablicą wskaźników do nowo dopisanych rekordów (tablica ta ma oczywiście identyczny format rekordu co plik "tt.key"). Na koszt tego scalania składa się sortowanie nowo dopisanych rekordów oraz samo scalanie.

Należy także doliczyć koszt zapisania pliku z informacją o otoczeniu. Koszt ten jest równy sumie kosztów: sortowania tablicy wskaźników rekordów dopisanych i uaktualnionych (operacje pamięciowe) i zapisania posortowanej tablicy na dysku (operacje dyskowe). Koszt całkowity scalania pliku tymczasowego z LTM wynosi:

$KC_2 = 2 * T \log T + T * \log \log L + (K + T)$

Przy czym należy zaznaczyć, że składnik $2 * M \log M$ dotyczy operacji pamięciowych (sortowanie) a dwa pozostałe dotyczą operacji dyskowych.

5.4. Programy usługowe dla implementacji algorytmu "BeBe+".

Część obsługi algorytmu "BeBe+" znajduje się w samym programie "Joanna". Funkcje algorytmu dostarczane przez program opisane zostały w rozdz. 4.5.

Oprócz tych funkcji w skład systemu wchodzi także zestaw programów obsługujących bezpośrednio samą bazę danych LTM. W systemie każdy z tych programów wywoływany jest przez plik wsadowy. Wszystkie pliki wsadowe umieszczono w katalogu:

C:\BEBEPLUS.DBS\PROG.BAT

Niżej podane krótko omówione jest przeznaczenie tych programów.

(1) BBMERGE.BAT

Program łączy pliki tymczasowe znajdujące się w katalogu "C:\BEBEPLUS.DBS\BEBESWAP.DAT\" z LTM. Jest on odpowiednikiem opcji <BeBePlus><Merge swapped data> z menu programu "Joanna".

(2) BBCHECK.BAT

Program sprawdza zawartość bazy danych. Jest on w stanie wykryć i usunąć niektóre rodzaje uszkodzeń w bazie danych.

(3) BACKUP.BAT

Tworzy kopię LTM w katalogu "C:\BEBESWAP.DBS\BACKUP".

(4) UNBACKUP.BAT

Odtwarza LTM z kopii ostatnio utworzonej programem "BACKUP". Niszczy aktualną zawartość LTM.

5.5. Test algorytmu "BeBe+".

Testu dokonano z użyciem komputera IBM AT 286 16 MHz. Algorytm testowano na następującej pozycji:

Białe: Kh2, Hh4, Gf6, Se5
Czarne: Kg8, Wh8, Sf8, Sg6, d7, e6, f7, g7, h7

W tej pozycji białe dają forsownego mata w pięciu półruchach po posunięciu: 1.Hh6! (1...gh6 2.Sg4 dowol. 3.Sh6 mat lub 1...gf6 2.Sg4 dowol. 3.Sf6 mat).

Joanna" myślała nad tą pozycją dziesięć razy po 30 sekund, przy włączonym algorytmie BeBe+. Po każdym namyśle łączono zawartość pliku tymczasowego z bazą danych.

Wyniki są następujące:

nr partii	posunięcie
1	1.Hb4
2-5	1.Hd4
6-10	1.Hh6

Posunięcie z pierwszej partii najprawdopodobniej prowadzi do porażki. Posunięcie grane w partiach 2-5 prowadzi do remisu (1.Hd4 gf6 2.Se8 itd.). Jak wspomniałem wcześniej posunięcie grane przez "Joannę" w następnych partiach prowadzi do mata najdalej w piątym półruchu.

Test potwierdza działanie algorytmu zgodnie z oczekiwaniami.

Bibliografia.

Programowaniu szachów poświęcono bardzo wiele książek i artykułów. Wiele z nich to prace, które niewiele wniosły do rozwoju dziedziny. Proponowane w nich rozwiązanie były albo "ślepymi uliczkami", albo poprostu nie spotkały się z szerszym zainteresowaniem. Natomiast artykuły takie jak "Programing Computer for playing chess" C.Shannona czy "Chess Skill in man and machine" P.Frey'a to "kamienie milowe" programowania gry w szachy. Przystępując do pisania niniejszej pracy nie posiadałem rozeznania, które książki rzeczywiście należy wyselekcjonować do przeczytania, a które pominąć, bo nie jest przecież możliwe przeczytanie wszystkich. Z tego powodu pragnąc oszczędzić czasu przyszłym entuzjastom programowania gry w szachy postanowiłem podzielić się moim doświadczeniem i wyselekcjonować w ramach tej bibliografii pozycje o szczególnym, moim zdaniem, znaczeniu .

Wszystkim pragnącym zapoznać się z dziedziną programowania gry w szachy polecam rozpocząć od przeczytania następujących pozycji: [7], [2], [4], [5] str.91-115, [6] str.3-9 55-159 197-217.

Przedstawiona niżej bibliografia składa się z dwóch części:

(1) Książki z których korzystałem bezpośrednio przy pisaniu niniejszej pracy.
(2) Książki do których istnieją odwołania w tekście tej pracy.

Część 1.

[1] - G.Owen: "Teoria gier", PWN 1975
[2] - Ed. P.Frey: "Chess Skill in Man and Machine", Springer-Verlag, 1977 r.
[3] - A.J.Palay: "Searching with probabilities", Pitman Advanced Publishing Program, 1983 r.
[4] - S.C.Shapiro: "Encyclopedia of Artificial Inteligence" Wiley-Interscience, 1987 r. str. 159-171
[5] - Ed. D.F.Beal: "Advances in Computer Chess 5", Nort-Holland, 1989 r.
[6] - Ed. T.A.Marsland, J.Schaeffer: "Computers Chess and Cognition", Springer-Verlag, 1990 r.

Część 2.

[7] - C.E.Shannon: Programming computer for playing chess", Edinburgh University Press pp.255-265, 1950 r.
[8] - A.M.Turing: "Digital computers aplied to games" z pracy "Faster then thought", B.V.Bowden, 1953 r.
[9] - D.E.Knuth, R.W.Moore: "An Analisys of Alpha-Beta Prunning", Artificial Inteligence, vol 6. str. 293 - 326, 1975 r.
[10] - Ed. M.R.B.Clarke: "Advances in Computer Chess 3", Pergamon Press, 1982 r.

[11] - ed. M.A.Bramer: "Computer Game-Playing - theory and practice", Ellis Horwood Series in Artficial Inteligence,

[12] - R.A.Finkel, J.P.Fishburn: "Improooved speedup bounds for alfabeta search" IEEE Trans. PAMI, 1983 r., vol.1, nr.1, str.89-91, 1983 r.

[13] - M.M.Botvinnik: "Computers in Chess", Springer-Verlag, 1984 r.

[14] - Adelson-Velsky , V.L.Arlazarov, M.V.Donskoy: "Algorithms for games", Springer-Verlag, 1988 r.

[15] - L.Bolz,J.Cytowski: "Metody przeszukiwania heurystycznego" tom 2, PWN , 1991 r.

[16] - Ed. D.F.Beal: "Advances in Computer Chess 6", Nort-Holland, 1991 r.

[17] - Ed. Alan Bundy: "Catalogue of artificial inteligence Techniques", Springer-Verlag 1990