Politechnika Łódzka
Instytut Informatyki

Praca magisterska

Praktyczne aspekty programowania gier logicznych

Piotr Beling

nr. albumu: 110341

Promotor: dr inż. Tadeusz Łyszkowski

Łódź, 2006

Instytut Informatyki
90-924 Łódź, ul. Wólczańska 215, budynek B9
tel. 042 631 27 97, 042 632 97 57, fax 042 630 34 14 email: office@ics.p.lodz.pl

Spis treści

Rozdział 1

Wstęp

1.1 Dlaczego warto pisać gry logiczne?

Gry logiczne stanowią jedno z zagadnień poruszanych w ramach badań nad sztuczną inteligencją. Ich dobrze określone reguły, łatwość porównywania różnych rozwiązań (np. poprzez bezpośredni pojedynek pomiędzy programami) oraz dostępność wiedzy ekspertów sprawia, iż są one świetnym poligonem doświadczalnym dla wielu algorytmów, np. przeszukiwania z wykorzystaniem heurystyk, algorytmów optymalizacji i metod uczenia maszynowego.

Dodatkowo, model prezentowanego w tym opracowaniu rozwiązania jest na tyle ogólny, że jego elementy mogą być adaptowane na potrzeby innych problemów[1].

Wszystko to sprawia, iż wielu naukowców, a także programistów (w tym programistów amatorów) zajmuje się konstruowaniem własnych komputerowych graczy. Jako efekt ich pracy powstaje pokaźna ilość gier i programów do analizy cieszących wielu zwolenników intelektualnych rozrywek. Część z tych programów gra na poziomie mistrzowskim i arcymistrzowskim.

1.2 Co zawiera to opracowanie?

Istnieje ogromna ilość bardzo dobrej literatury, na temat programowania gier logicznych. Wiele pozycji omawia jednak dość szczegółowo pojedyncze jego aspekty, stosowane pomysły lub heurystyki, nie dając często dobrego obrazu całości zagadnienia.

W tym opracowaniu, spróbuję opisać najczęściej stosowane algorytmy i nakreślić nieco bardziej ogólnie jak wygląda praktyka programowania gier logicznych. Aby

[1] np. występujących przy konstruowaniu systemów wspomagających podejmowanie decyzji czy też systemów sterowania

praca nie była jednak zbyt ogólna, skupię się głównie na grach dwuosobowych, deterministycznych, z pełną informacją, o zerowej sumie wypłat (definicja tej klasy gier znajduje się w pkt. 2.1).

Dla każdego z opisywanych algorytmów, opierając się zarówno na doświadczeniach własnych jak i licznej literaturze, postaram się nakreślić jego ideę, motywację jaka przyświecała jego twórcy, oraz podać informacje niezbędne do poprawnej implementacji.

Opracowanie to powinno więc stanowić swoisty podręcznik na temat programowania gier logicznych.

Dodatkowym celem mojej pracy jest napisanie, bazując na opisywanych algorytmach, programu grającego w warcaby klasyczne oraz zaprezentowanie szczegółów jego implementacji (szczególnie tych, efektywnych rozwiązań, których idea jest na tyle ogólna, że z powodzeniem mogą zostać zaadaptowane na potrzeby innych gier, jak np. sposób reprezentacji planszy w pamięci komputera, sposób numerowania sytuacji na potrzeby przechowywania ich w bazie końcówek, itd).

Na przykładzie napisanego programu, sprawdzę, jak skuteczne są zastosowane w nim, powszechnie używane heurystyki. Spróbuję także zmierzyć wkład poszczególnych z nich w prezentowane przez komputerowego gracza umiejętności i zbadać jakie związki zachodzą pomiędzy nimi (które i jak się uzupełniają).

Rozdział 2

Wstęp do teorii gier

2.1 Podstawowe pojęcia teorii gier

Teoria gier to dział matematyki zajmujący się badaniem optymalnego zachowania w przypadku konfliktu interesów.

Gra to dowolna sytuacja konfliktowa. Gracz to dowolny uczestnik gry, który postępując według pewnej strategii, próbuje osiągnąć określony cel. Zależnie od strategii własnej oraz innych uczestników, każdy gracz otrzymuje wypłatę w tzw. jednostkach użyteczności (która jest miarą zwycięstwa, zysku gracza).

Definicja 1 (gry)**.** *Gra składa się z:*

- *zbioru graczy (uczestników gry)*
- *zbioru możliwych dla każdego z graczy strategii*
- *funkcji wypłaty, przyporządkowującej każdej kombinacji strategii przyjętych przez poszczególnych graczy wypłatę dla każdego z uczestników*

Definicja 2 (strategii)**.** *Strategia gracza jest jednoznacznie opisana przez określenie decyzji jaką powinien podjąć ten gracz dla każdej możliwej sytuacji w grze (dla każdego stanu gry).*

Definicja 3 (gry o sumie stałej)**.** *Gra o sumie stałej, to gra w której suma wypłat wszystkich graczy jest stała (tj. niezależna od strategii poszczególnych graczy).*

Definicja 4 (gry o sumie zerowej)**.** *Gra o sumie zerowej, to gra o sumie stałej w której suma wypłat wszystkich graczy jest równa zero.*

Z definicji 4 bezpośrednio wynika twierdzenie 1.

Twierdzenie 1. *W grze dwuosobowej o sumie zerowej, wartość wypłaty jednego z graczy jest równa co do wartości bezwzględnej i przeciwna co do znaku do wypłaty drugiego z graczy.*

Ponieważ grę o sumie stałej łatwo sprowadzić do gry o sumie zerowej, to pojęcia te często są utożsamiane.

Bardziej rozbudowane wprowadzenie do teorii gier można znaleźć np. w [7].

2.2 Gry logiczne w kontekście teorii gier

Większość powszechnie znanych gier logicznych to gry dwuosobowe o sumie zerowej. Przykładami takich gier są: szachy, warcaby, GO, brydż[1], otello, backgammon (tryktrak), poker, kółko i krzyżyk.

Typowy przebieg rozgrywki wyżej wymienionych gier jest następujący: przepisy gry określają początkowy stan gry[2]. Następnie gracze wykonują ruchy (zazwyczaj na przemian). Wykonanie ruchu polega na przeprowadzeniu stanu gry z bieżącego, do następnego, zgodnie z zasadami gry. Gra kończy się, gdy zostanie osiągnięty jeden z (określonych w regulaminie) stanów końcowych (regulamin określa też wypłatę każdego z graczy w osiągniętej sytuacji, której wartość w większości gier może przyjąć tylko trzy wartości: dla wygranej lub przegranej konkretnego gracza, albo remisu).

Jak wynika z powyższego opisu, przepisy gry tak naprawdę charakteryzują pewien skierowany graf (dalej nazywany *grafem gry*). Węzłami tego grafu są stany gry (punkty w pewnej przestrzeni stanowej $\mathbb{S}$). Następnikami każdego węzła są stany osiągalne (poprzez wykonanie jednego ruchu) ze stanu reprezentowanego przez ten węzeł. Jeden z węzłów jest wyróżniony jako stan początkowy gry. Stany końcowe charakteryzuje brak następników.

W niektórych grach stan początkowy jest wybierany losowo (np. w brydżu czy pokerze poprzez rozdanie kart). Dodatkowo, gracze często nie mają pełnej informacji (np. nie znają kart przeciwnika) o stanie, w którym znajduje się aktualnie rywalizacja (są to tzw. *gry z niepełną informacją*), nie wiedzą po której części grafu „się poruszają". Wraz z przebiegiem gry (ujawniania się pewnych faktów, np. kart oponenta) mogą sie oczywiście tego domyślać. Nie rzadko wyciągają też wnioski z poczynań samego rywala (występują pewne aspekty psychologiczne).

[1] mimo, iż w brydża fizycznie grają 4 osoby, to z punktu widzenia teorii gier jest to gra dwuosobowa - konflikt interesów zachodzi pomiędzy obiema parami

[2] nieformalnie, stan gry w danym momencie można by określić jako całość informacji pozwalającej wznowić przerwaną w tym momencie grę; w szachach np. jest określony przez ustawienie bierek na szachownicy, wskazanie gracza do którego należy ruch i pewne dodatkowe dane, np. o tym która ze stron wykonała już roszadę, itd.

W pewnych grach (przykładem może być backgammon), zbiór ruchów jakie może wykonać gracz w danej sytuacji wyznaczony jest przez czynnik losowy (w backgammonie rzut kostkami). O grach, w których występuje czynnik losowy mówimy, że są *niedeterministyczne*[3].

[3]w przeciwieństwie do gier *deterministycznych*, tj. niezależnych od czynnika losowego

Rozdział 3

Algorytmy minimaksowe

3.1 Jak komputerowy gracz podejmuje decyzje? Algorytm Min-Max

Jednym z podstawowych problemów przy pisaniu komputerowego gracza jest napisanie dobrego algorytmu podejmowania przez niego decyzji w trakcie rozgrywki.

Załóżmy, że gra znajduje się w pewnym stanie S i nasz zawodnik, o imieniu Max, ma wykonać ruch, tj. wybrać jeden ze stanów bezpośrednio osiągalnych z S (oznaczmy zbiór takich stanów przez $nast(S)$). Powinien on oczywiście wybrać posunięcie, które daje mu największe szanse na wygraną, czyli takie które doprowadzi do najkorzystniejszego (z jego punktu widzenia) stanu $S_B \in nast(S)$. Przy czym, musi pamiętać, że kolejny ruch, w sytuacji S_B, należy do jego oponenta (nazwijmy go Min). Ten zaś, gra dobrze i będąc w sytuacji S_B wybierze najgorszy dla Maksa stan z $nast(S_B)$. Dlatego, Maks może ocenić każdą z sytuacji $S_B \in nast(S)$ tak samo jak najgorszą z sytuacji $nast(S_B)$. Krótko mówiąc, dobry ruch to taki, po którym przeciwnik nie ma dobrej odpowiedzi. Oczywiście każda z potencjalnych jego (Mina) odpowiedzi doprowadzi do sytuacji w której znów Max będzie mógł wybrać i zmaksymalizować ocenę kolejnego węzła grafu gry. Następnie zaś decydował (i minimalizował) będzie Min, itd.

Takie przewidywanie ruchów (przeszukiwanie grafu gry) musi się kiedyś skończyć. Może oczywiście w stanach końcowych gry, które nie mają następników i są banalne w ocenie (regulamin określa wielkość wypłaty dla nich).

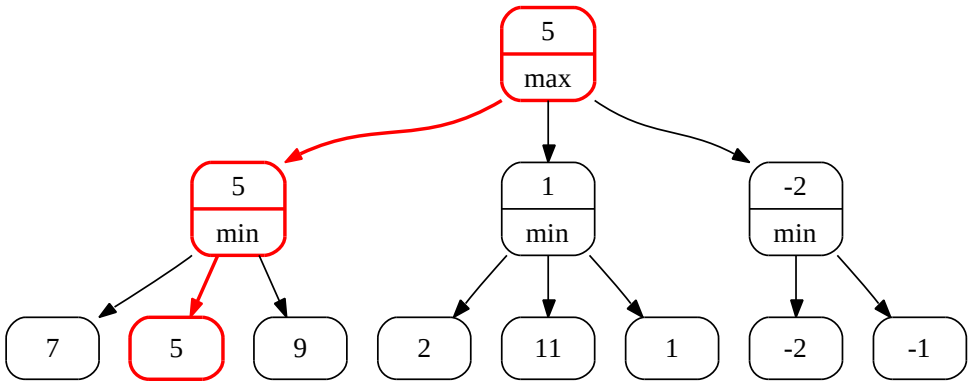
Wydruk 3.1 przedstawia algorytm skonstruowany zgodnie z powyższym rozumowaniem (zanotowany w pseudokodzie wzorowanym na C++). W linii 4 funkcja $nast(S)$ (dalej zwana *generatorem ruchów*[1]) zwraca następniki stanu S w grafie gry.

[1] dla uproszczenia przyjęto, że generator ruchów przekazuje swój wynik jako wektor stanów. Równie dobrze sprawdzi się każda struktura, po której można iterować. Można też przechowywać w niej opisy posunięć zamiast całych stanów i generować kolejne sytuacje tylko na czas ich sprawdzenia.

Listing 3.1: algorytm Min-Max

```
1  //Zwraca wielkość wypłaty gracza G
2  //S - aktualny stan gry
3  int MinMax(Stan S) {
4      vector<Stan> N = nast(S);
5      if (N == ∅)
6          return wyplata(S, G);
7      int result = MinMax(N[0]);
8      if (S.czyj_ruch == G) { //ruch należy do G
9          //szukamy maksimum z ocen kolejnych pozycji
10         for (int i = 1; i < |N|; i++) {
11             int val = MinMax(N[i]);
12             if (val > result)
13                 result = val;
14         }
15     } else {     //ruch należy do rywala G
16         //szukamym minimum z ocen kolejnych pozycji
17         for (int i = 1; i < |N|; i++) {
18             int val = MinMax(N[i]);
19             if (val < result)
20                 result = val;
21         }
22     }
23     return result;
24 }
```

Rysunek 3.1: Przykładowe drzewo poszukiwań algorytmu Min-Max z zaznaczonym głównym wariantem gry



Jeżeli S jest stanem końcowym, zwracana jest wypłata[2] gracza Max zdefiniowana dla S (linie: 5-6). W przeciwnym wypadku, funkcja MinMax jest wywoływana rekurencyjnie dla każdego następnika S, zaś ze zbioru wyników tych wywołań oblicza się maksimum lub minimum (w zależności od tego czy wybór ruchu należy do Maxa czy też do jego rywala).

Widać więc, że Min-Max przeszukuje graf gry w głąb, w wyniku dając wartość wypłaty przepisaną z pewnego liścia (przewidywanego stanu końcowego) drzewa poszukiwań. Ścieżkę łączącą tego liścia z korzeniem nazywać będziemy *głównym wariantem gry*. Rysunek 3.1 przedstawia przykładowe drzewo poszukiwań. Węzły w których oblicza się maksimum (na rysunku, korzeń) nazywamy *węzłami typu maksimum*, zaś węzły w których oblicza się minimum, *węzłami typu minimum* (na rysunku dzieci korzenia).

Ponieważ algorytm Min-Max w postaci 3.1 nie używa dodatkowej pamięci w celu sprawdzenia które stany już odwiedził, a jego jedynym warunkiem stopu jest dotarcie do węzła w grafie który nie ma następników, to do jego poprawności wymagane jest by graf gry był acykliczny (w przeciwnym razie algorytm może błądzić po grafie w koło, bez końca). Mimo, iż grafy większości gier spełniają ten warunek[3], to użyty warunek stopu w praktyce i tak jest za słaby. Dzieje się tak ze względu na ogrom przestrzeni stanowej większości gier. Zauważmy, że Min-Max ma złożoność wykładniczą względem wysokości drzewa poszukiwań i jedyną znaną powszechnie grą, dla której miał by szanse skończyć pracę w rozsądnym czasie (wywołany dla stanu początkowego) jest kółko i krzyżyk. Dlatego w praktyce stosuje się dodatkowe warunki stopu. Najczęściej stosowanym z nich, jest ograniczenie głębokości wywołań rekurencyjnych funkcji MinMax, do pewnej wartości przekazywanej jako dodatkowy parametr jej wywołania. Po osiągnięciu założonej głębokości nie rozpatruje się następników stanu S, zaś jako wynik wywołania MinMax (konkretnie MinMax(S, 0)) zwraca się statyczną ocenę S, czyli wartość tzw. *funkcji oceniającej* dla stanu S. Funkcja oceniająca $ocena_G : \mathbb{S} \rightarrow \Re$ (gdzie $\mathbb{S}$ to przestrzeń stanowa gry) powinna możliwie dokładnie aproksymować wartość MinMax(S) dla każdego $S \in \mathbb{S}$ i jednocześnie powinna dać się obliczyć możliwie szybko. Zazwyczaj, konstruuje się ją w oparciu o wskazane przez ekspertów (w dziedzinie danej gry) cechy stanu gry (np. w warcabach może ona uwzględniać przewagę w ilości posiadanych pionów lub damek, odległość pionów od linii promocji, itd.). Więcej szczegółów znajduje się w rozdziale 5.

Zmodyfikowaną wersję Min-Max przedstawia wydruk 3.2. W przeciwieństwie do wersji 3.1 nie zawsze zwraca on dokładny wynik jakim zakończy się partia po optymalnej grze obu stron, ale za to nadaje się do praktycznego użycia.

Złożoność czasowa tej wersji Min-Max wynosi $O(\mathbb{B}^d)$, gdzie d to głębokość na jaką eksplorowany jest graf gry (wysokość drzewa poszukiwań), zaś wartość $\mathbb{B}$ (dalej zwana

[2] dla uproszczenia przyjęto że jest ona wyrażona jako liczba całkowita

[3] np. w szachach czy warcabach regulamin przewiduje remis przez kilkukrotne powtórzenie ruchów lub sytuacji. W praktyce, sprawdzenie tego pkt. przepisów w trakcie przeszukiwania grafu gry, wymaga jednak dodatkowej pamięci, por. roz. 4.2.

Listing 3.2: algorytm Min-Max z ograniczeniem głębokości wywołań

```
1  //Zwraca przewidywaną wielkość wypłaty gracza G
2  //S - aktualny stan gry, Depth - głębokość poszukiwań
3  int MinMax(Stan S, int Depth) {
4      if (Depth == 0)
5          return ocena_G(S);
6      vector<Stan> N = nast(S);
7      if (N == ∅)
8          return wyplata(S, G);
9      int result = MinMax(N[0], Depth-1);
10     if (S.czyj_ruch == G) { //ruch należy do G
11         //szukamy maksimum z ocen kolejnych pozycji
12         for (int i = 1; i < |N|; i++) {
13             int val = MinMax(N[i], Depth-1);
14             if (val > result)
15                 result = val;
16         }
17     } else {     //ruch należy do rywala G
18         //szukamym minimum z ocen kolejnych pozycji
19         for (int i = 1; i < |N|; i++) {
20             int val = MinMax(N[i], Depth-1);
21             if (val < result)
22                 result = val;
23         }
24     }
25     return result;
26 }
```

średnim czynnikiem rozgałęzienia - ang. average branching factor) zależy od gry i jest średnią ilością następników dowolnego stanu gry. Złożoność pamięciowa wynosi $O(\mathbb{B}d)$ (lub nawet $O(d)$ dla generatora ruchów konstruującego stany tylko na czas ich przeszukania).

Dokładność samego wyniku rośnie wraz ze wzrostem głębokości poszukiwań i jakości użytej funkcji oceniającej.

3.2 Nega-Max czyli Min-Max w uproszczonej notacji

Zapis algorytmu Min-Max można ujednolić korzystając ze spostrzeżenia, że:

$$\forall a_1, a_2, \ldots, a_N \in \Re\colon \min(a_1, a_2, \ldots, a_N) = -\max(-a_1, -a_2, \ldots, -a_N) \quad (3.1)$$

Listing 3.3: algorytm Nega-Max

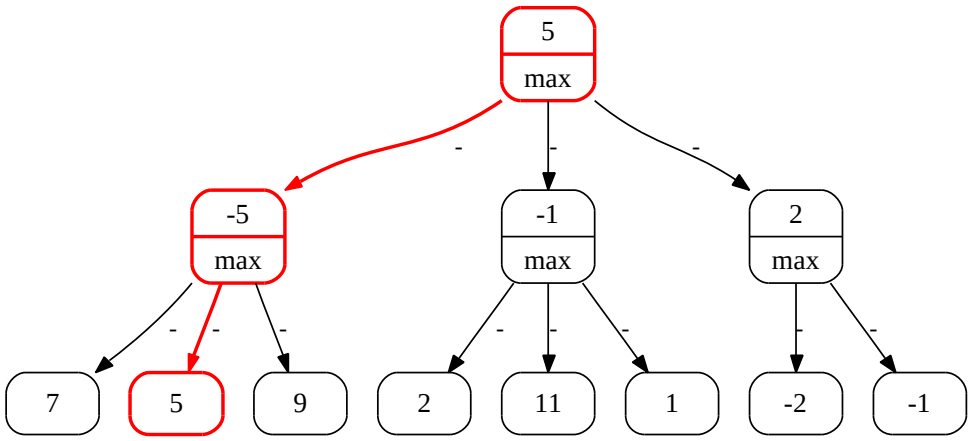
```
1  //Zwraca przewidywaną wypłatę gracza do którego należy ruch w stanie S
2  //S - aktualny stan gry, Depth - głębokość poszukiwań
3  int NegaMax(Stan S, int Depth) {
4      if (Depth == 0)
5          return ocena(S);
6      vector<Stan> N = nast(S);
7      if (N == ∅)
8          return wyplata(S);
9      int result = −∞;
10     //szukamy maksimum z ocen kolejnych pozycji
11     for (int i = 0; i < |N|; i++) {
12         int val = - NegaMax(N[i], Depth-1);
13         if (val > result)
14             result = val;
15     }
16     return result;
17 }
```

Można więc we wszystkich węzłach typu minimum obliczać maksimum, biorąc jednak do obliczeń wartości przeciwne do tych zwróconych przez podrzędne wywołania rekurencyjne (jak na listingu 3.3). Wywołanie nadrzędne (które jest dla węzła typu maksimum) dołoży minus do zwróconego wyniku. Trzeba jednak pamiętać, że Nega-Max dla węzła typu minimum oblicza wynik przeciwny do tego obliczonego przez Min-Max.

W związku z powyższym i zgodnie z twierdzeniem 1 wynik $NegaMax(S,\ d)$ dla dowolnych $d \in \mathbb{N}$ i $S \in \mathbb{S}$, można interpretować jako przewidywaną wypłatę gracza do którego należy ruch w stanie S. Analogicznie muszą być zdefiniowane funkcje $wyplata$ i $ocena$. Pierwsza z nich musi zwracać wypłatę dla węzła S dla gracza, do którego należałby ruch w tym stanie (tzn. dla przeciwnika gracza który wykonał ostatnie posunięcie). Zaś $ocena(S)$ musi aproksymować wartość $NegaMax(S, +\infty)$.

Listing 3.3 przedstawia algorytm Nega-Max, zaś rysunek 3.2 jego przykładowe działanie (dla tej samej sytuacji co na rysunku 3.1). W zasadniczej części funkcja $NegaMax$ wywoływana jest rekurencyjnie dla każdego następnika stanu S, zaś wyniki tych wywołań są negowane (co daje przewidywaną wypłatę po wybraniu każdego z badanych następników z punktu widzenia wykonującego ruch w stanie S). Z otrzymanego zbioru liczb, idący wybiera tę największą (wynik najkorzystniejszy z jego punktu widzenia).

Rysunek 3.2: Przykładowe drzewo poszukiwań algorytmu Nega-Max z zaznaczonym głównym wariantem gry



3.3 α-β - podstawowy algorytm cięć

Jak już wspomniano, głównym czynnikiem, który utrudnia nam skonstruowanie naprawdę dobrego komputerowego gracza, jest brak dostatecznej ilości czasu jaką ma on na podjęcie decyzji. Dla dużych głębokości poszukiwań ilość węzłów odwiedzonych przez algorytm Nega-Max jest ogromna. Okazuje się jednak, że nie ma potrzeby sprawdzenia wszystkich gałęzi drzewa poszukiwań, by wyznaczyć wartość Nega-Max dla jego korzenia.

W 1975 roku Knuth i Moore w artykule „An Analysis of Alpha-Beta Pruning" opisali algorytm, który stał się podstawową metodą obcinania gałęzi w drzewach minimaksowych (zarówno stosowany bezpośrednio, jak i również jako część innych metod). Nad ideą strzyżenia drzew minimaksowych wcześniej pracowali McCarthy (1956) i Brudno (1963). Sam algorytm α-β cięć odkryli w 1958 r. trzej naukowcy: Allen Newell, John Shaw i Herbert Simon.

By nakreślić zasadę działania α-β cięć, rozważmy sytuację przedstawioną na rysunku 3.3 (por. też rys. 3.2). Załóżmy, że pierwszy (licząc od lewej) następnik korzenia został przeszukany i dał (po zanegowaniu) wartość $d_1 = 5$, z czego wynika, że wartość dla korzenia wyniesie:

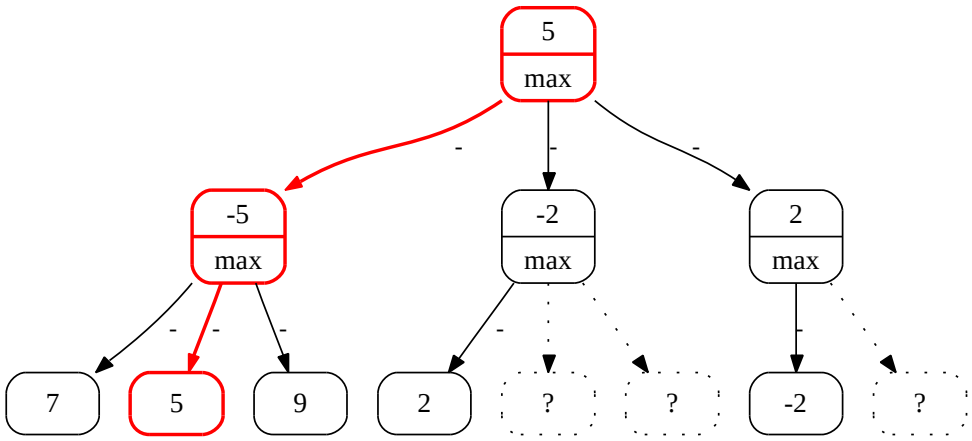
$$r = \max(5, d_2, d_3) \geqslant 5$$

gdzie d_2 i d_3 to zanegowane wartości uzyskane dla kolejnych dzieci korzenia.

Teraz przeszukiwany jest kolejny (drugi od lewej) jego następnik. Pierwsze jego dziecko dało wartość (po zanegowaniu) -2. Oznacza to, że do korzenia zostanie zwrócona wartość:

$$-d_2 \geqslant -2$$

Rysunek 3.3: Przykładowe drzewo poszukiwań algorytmu alfa-beta z zaznaczonym głównym wariantem gry



czyli (po zanegowaniu):

$$d_2 \leqslant 2 \leqslant 5$$

Dlatego też:

$$r = \max(5, d_2, d_3) = \max(5, d_3)$$

niezależnie od tego ile dokładnie wynosi d_2 (ważne jest, że nie wynosi ono więcej niż 5). Kolejne następniki drugiego dziecka korzenia nie muszą być zatem w ogóle odwiedzone, zaś jako wartość tego dziecka możemy zwrócić[4] 2, albo nawet $d_1 = 5$. Mimo, iż nie jest ona zgodna z tą policzoną przez Nega-Max (por. rys. 3.2) to takie postępowanie nie ma wpływu na ostateczny wynik, tj. wartość r. Na podobnej zasadzie może zostać obcięty jeden z następników jego trzeciego dziecka.

Listing 3.3 ukazuje algorytm α-β. Jak widać w stosunku do Nega-Max (wydruk 3.2) nasza funkcja przyjmuje dwa dodatkowe parametry. Jeśli oznaczymy przez G zawodnika do którego należy ruch w stanie S to:

- α jest największą wartością jaką osiągną dotychczas G, tzn. mógł on wcześniej wykonać posunięcie, które w efekcie doprowadziłoby grę do stanu o ocenie nie mniejszej niż α
- β jest najmniejszą wartością (z punktu widzenia G) do jakiej mógł doprowadzić rywal gracza G

Jeżeli bieżąca wartość oceny stanu S osiągnie lub przekroczy β (linia 14), to przeszukanie kolejnych następników S nie ma sensu, gdyż rywal G będzie wolał wykonać wcześniej ruch, który doprowadzi grę do stanu o ocenie β zamiast do S. Może nastąpić odcięcie (linia 15) i jako wartość stanu S można zwrócić β.

[4] tak czynimy w przypadku wersji fail-soft algorytmu α-β

Listing 3.4: algorytm α-β

```
1  //Zwraca przewidywaną wypłatę gracza do którego należy ruch w stanie S
2  //S - aktualny stan gry, Depth - głębokość poszukiwań
3  //α, β - największa i najmniejsza znaleziona wcześniej wartość
4  int AlfaBeta(Stan S, int Depth, int α, int β) {
5      if (Depth == 0)
6          return ocena(S);
7      vector<Stan> N = nast(S);
8      if (N == ∅)
9          return wyplata(S);
10     sort(N);   //krok opcjonalny
11     //szukamy maksimum z ocen kolejnych pozycji
12     for (int i = 0; i < |N|; i++) {
13         int val = - AlfaBeta(N[i], Depth-1, -β, -α);
14         if (val ⩾ β)
15             return β;
16         if (val > α)
17             α = val; //poprawa maksimum
18     }
19     return α;
20 }
```

Przekazując α i β do wywołań rekurencyjnych (linia 13) trzeba pamiętać, iż interpretacja tych liczb wewnątrz tego wywołania następuje z punktu widzenia rywala G, dlatego też należy te wartości zanegować i zamienić miejscami.

Opcjonalny krok w linii 10 omówię przy okazji analizy złożoności α-β.

Twierdzenie 2 (o związku α-β z Nega-Max)**.** *Niech $d \geqslant 0$, $\alpha < \beta$, $S \in \mathbb{S}$. Oznaczmy ponadto $ab = AlfaBeta(S, d, \alpha, \beta)$ oraz $n = NegaMax(S, d)$. Wtedy możemy wyróżnić trzy sytuacje:*

- *(sukces)*
$$\alpha < ab < \beta \quad \Rightarrow \quad ab = n \tag{3.2}$$
- *(failing low)*
$$\alpha \geqslant ab \quad \Rightarrow \quad ab \geqslant n \tag{3.3}$$
- *(failing high)*
$$\beta \leqslant ab \quad \Rightarrow \quad ab \leqslant n \tag{3.4}$$

Bezpośrednio z implikacji 3.2 wynika:

$$\forall S \in \mathbb{S}, d \geqslant 0: \qquad AlfaBeta(S, d, -\infty, \infty) = NegaMax(S, d) \tag{3.5}$$

co określa możliwość bezpośredniego wykorzystania α-β do znalezienia wartości negamaksowej stanu gry.

Ilość odcięć jaką wykona taka metoda jest ściśle uzależniona od kolejności rozpatrzenia węzłów. W najgorszym wypadku nie nastąpi żadne odcięcie (warunek w linii 14 nigdy nie zajdzie), α-β odwiedzi dokładnie te same węzły co Nega-Max o złożoność czasowej $O(\mathbb{B}^d)$ (d to głębokość poszukiwań, zaś $\mathbb{B}$ to średni czynnik rozgałęzienia). Najmniej czasochłonny przypadek nastąpi, gdy dla każdego rozważanego stanu, najlepszy (dla idącego w tym stanie) z możliwych ruchów zostanie sprawdzony jako pierwszy. Wtedy złożoność wyniesie $O(\mathbb{B}^{d/2})$. Dla ruchów wykonanych w losowej kolejności można oczekiwać złożoności $O(\mathbb{B}^{3d/4})$.

Opcjonalnie, można uporządkować (wg. pewnej wstępnej oceny) następniki rozważanego stanu S przed ich przejrzeniem (linia 10 listingu 3.4) tak, by zwiększyć szansę na wcześniejsze przejrzenie dobrych ruchów. Można wtedy liczyć na szybsze odcięcia i zbliżenie się do dolnej granicy ($O(\mathbb{B}^{d/2})$) złożoności czasowej α-β.

3.4 Fail-soft α-β

Algorytm α-β w klasycznej postaci (roz. 3.3) zwracał wartości z przedziału $[\alpha, \beta]$, gubiąc czasami informacje za pomocą jakiej dokładnie wartości nastąpiło odcięcie. W wielu wypadkach informacja ta nie ma znaczenia, ale niektóre algorytmy umieją z niej skorzystać (jest ona często dokładniejszym ograniczeniem górnym lub dolnym prawdziwej wartość węzła).

Przedstawiony na wydruku 3.5 fail-soft α-β może zwracać wyniki spoza przedziału $[\alpha, \beta]$. Różnice w stosunku do algorytmu z listingu 3.4, to zwracanie wartości węzła powodującego cięcie zamiast β (linia 16) oraz szukanie wartości badanego stanu bez ograniczenia jej od dołu przez α (co wymagało wprowadzenia dodatkowej zmiennej „best").

Twierdzenie 2 pozostaje prawdziwe dla tej wersji α-β.

3.5 W poszukiwaniu głównego wariantu... PVS

Wydruk 3.6 przedstawia algorytm PVS (ang. Principal Variation Search) znany też pod nazwą NegaScout, który w istocie rzeczy jest zmodyfikowaną wersją metody α-β cięć. Flaga fFoundPv (linia 11) określa, czy znaleziono kandydata na najlepszy następnik badanego stanu S. Zakłada się, iż następnikiem tym jest pierwszy węzeł o wartości wyższej niż α (linie 22 i 24). Od momentu znalezienia takowego, warunek w linii 14 będzie prawdziwy i zamiast przeszukiwać kolejne następniki S w pełnym oknie (α, β)[5] (jak w linii 19) próbuje się je w oknie $(\alpha, \alpha+1)$[6] (linia 15), co zazwyczaj

[5] z punktu widzenia idącego, zaś $(-\beta, -\alpha)$ z punktu widzenia jego rywala

[6] z punktu widzenia idącego, zaś $(-\alpha - 1, -\alpha)$ z punktu widzenia jego rywala

Listing 3.5: algorytm fail-soft α-β

```
1  //Zwraca przewidywaną wypłatę gracza do którego należy ruch w stanie S
2  //S - aktualny stan gry, Depth - głębokość poszukiwań
3  //α, β - największa i najmniejsza znaleziona wcześniej wartość
4  int AlfaBetaFS(Stan S, int Depth, int α, int β) {
5      if (Depth == 0)
6          return ocena(S);
7      vector<Stan> N = nast(S);
8      if (N == ∅)
9          return wyplata(S);
10     sort(N); //krok opcjonalny
11     //szukamy maksimum (best) z ocen kolejnych pozycji
12     int best = -∞;
13     for (int i = 0; i < |N|; i++) {
14         int val = - AlfaBetaFS(N[i], Depth-1, -β, -α);
15         if (val ⩾ β)
16             return val;
17         if (val > α)
18             α = val; //poprawa ograniczenia
19         if (val > best)
20             best = val;  //poprawa maksimum
21     }
22     return best;
23 }
```

jest wyraźnie szybsze. Takie postępowanie pozwala (na podstawie implikacji 3.3 twierdzenia 2) efektywnie zweryfikować naszą tezę, iż najlepszy następnik S ma wartość α. Jeśli weryfikacja nie powiedzie się (wiersz 16), musimy ponownie przeszukać badany węzeł, w pełnym[7] oknie (α, β), co może (a nawet powinno) doprowadzić do poprawienia α (ustalenia nowego kandydata). Powtórne przeszukiwanie jest oczywiście czasochłonne, dlatego zaleca się uporządkowanie (linia 10) następników stanu S, tak, by zwiększyć szansę na wcześniejsze przejrzenie dobrych ruchów i na prawdziwość heurystycznej tezy.

W praktyce, przy dobrym uporządkowaniu ruchów, PVS okazuje się o kilka procent szybszy od „czystego" α-β.

[7] tak naprawdę, szczególnie gdy używana jest wersja fail-soft α-β, można użyć mniejszego okna (val, β), trzeba jednak uważać na skutki ewentualnych niestabilności, por. roz. 4.9

Listing 3.6: algorytm PVS

```
1  //Zwraca przewidywaną wypłatę gracza do którego należy ruch w stanie S
2  //S - aktualny stan gry, Depth - głębokość poszukiwań
3  //α, β - aktualne „okno"
4  int PVS(Stan S, int Depth, int α, int β) {
5      if (Depth == 0)
6          return ocena(S);
7      vector<Stan> N = nast(S);
8      if (N == ∅)
9          return wyplata(S);
10     sort(N);   //krok opcjonalny, zalecany
11     bool fFoundPv = false;
12     for (int i = 0; i < |N|; i++) {
13         int val;
14         if (fFoundPv) {
15             val = - PVS(N[i], Depth-1,  -α-1, -α);
16             if ((val > α) && (val < β)) // błąd w założeniach?
17                 val = - PVS(N[i], Depth-1, -β, -α);
18         } else
19             val = - PVS(N[i], Depth-1, -β, -α);
20         if (val ⩾ β)
21             return β; //cięcie
22         if (val > α) {
23             α = val; //poprawa maksimum
24             fFoundPv = true; //N[i] jest najlepszy(?)
25         }
26     }
27     return α;
28 }
```

3.6 Iteracyjne pogłębianie

Jednym z problemów, jaki napotkamy programując grę logiczną w oparciu o algorytmy opisane w poprzednich rozdziałach, będzie odpowiednie dobranie głębokości na jaką powinniśmy przeszukiwać graf gry.

Zbyt mała głębokość eksploracji oznacza mało dokładny wynik (i słabą grę), zaś zbyt duża, długi czas oczekiwania na ten wynik i możliwość naruszenia przez naszego gracza ograniczeń czasowych[8].

Iteracyjne pogłębianie jest niezłym i powszechnie stosowanym rozwiązaniem opisanego problemu. W pierwszej iteracji graf gry przeszukiwany jest dość płytko i dzięki temu szybko. Dopóty, dopóki pozostanie dostatecznie dużo czasu, graf jest przeglądany ponownie, coraz głębiej. Gdy czas się skończy, zwracana jest ostatnia znaleziona odpowiedź.

W [26] autor zaproponował rozszerzenie algorytmu, przydatne, gdy czas na rozegranie partii jest ograniczony: jeżeli wyniki otrzymane w kilku kolejnych iteracjach nie różnią się, lub różnią się nieznacznie, ale wciąż jeden ruch jest typowany jako najlepszy, to można domniemać, że sytuacja jest „prosta", wykonać „narzucający się" ruch i zaoszczędzić w ten sposób czas, który może sie przydać później.

Narzuca się jednak pytanie, ile dodatkowego czasu kosztuje takie wielokrotne przeszukiwanie? Okazuje się, że nie tak dużo.

Po pierwsze, zwiększenie głębokości przeglądania o $k \in \mathbb{N}$, zwiększa drzewo poszukiwań około $\mathbb{B}^k$ razy (gdzie $\mathbb{B}$ - średni czynnik rozgałęzienia). Oznacza to, że płytsze sprawdzenia trwają ułamki czasu tego najgłębszego.

Po drugie, można wykorzystać informacje pochodzące z wcześniej wykonanego przeszukiwania, by przyspieszyć to kolejne, głębsze. Najprostszym sposobem dokonania tego, jest próba zwiększenia ilości cięć algorytmu α-β[9] poprzez uporządkowanie dzieci korzenia według ostatnio uzyskanych ocen, tak, by najbardziej obiecujące ruchy były sprawdzane na początku. Dodatkowo można zapamiętać w trakcie przeszukiwania główny wariant gry, po to, by przy głębszym przeglądaniu, ruchy wyznaczone przez niego wykonać jako pierwsze[10].

3.7 Algorytm aspirującego okna

Algorytm aspirującego okna (przedstawiony na wydruku 3.7) jest rozszerzoną wersją iteracyjnego pogłębiania. Ulepszenie polega na zawężeniu okna poszukiwań (α, β) kolejnej iteracji do pewnego otoczenia wyniku uzyskanego w poprzedniej. Takie postępowanie, powinno zwiększyć ilość odcięć wykonanych przez algorytm α-β. Jeżeli wynik nieszczęśliwie nie zmieści się w „aspirującym" oknie, przeszukiwanie na daną

[8] częstą praktyką (szczególnie na różnych turniejach) jest ograniczanie czasu jaki przysługuje każdemu z graczy na wykonanie ruchu lub rozegranie partii

[9] lub jakiejś jego pochodnej, np. PVS

[10] w ogólniejszym wymiarze można to uzyskać stosując tablicę transpozycji (por. roz. 4.2)

Listing 3.7: algorytm aspirującego okna z porządkowaniem ruchów

```
1  //Zwraca „najlepszy" następnik stanu gry S
2  Stan aspwindow(Stan S) {
3      vector<Stan> N = nast(S);
4      if (N == ∅) throw Exception("brak ruchów");
5      int values[|N|];      //wektor ocen ruchów
6      int α = −∞, β = ∞, depth = 1;
7      do {
8          int bestval = α;
9          for (int i = 0; i < |N|; i++) {
10             values[i] = − AlphaBeta(N[i], depth−1, −β, −bestval);
11             if (values[i] ⩾ β) { //nie zmieściliśmy się w oknie
12                 bestval = β;
13                 break;
14             }
15             if (values[i] > bestval)
16                 bestval = values[i];   //poprawa maksimum
17         }   //bestval = AlphaBeta(S, depth, α, β);
18         if ((bestval ⩽ alpha) || (bestval ⩾ beta)) {
19             α = −∞; //nie zmieściliśmy się w aspirującym oknie,
20             β = ∞;  //zwiększamy okno,
21             continue;     //szukamy ponownie na głębokość depth
22         }
23         sort(N, values);//porządkujemy następniki po ich ocenach
24         depth++;          //zwiększamy głębokość o 1 (można więcej)
25         //ustawiamy nowe aspirujące okno (δ > 0 to stała):
26         α = bestval − δ;
27         β = bestval + δ;
28     } while (is_time_to_search(...));
29     return N[0];
30 }
```

głębokość musi zostać powtórzone w większym oknie, np. $(-\infty, \infty)$. Nie powinno się to jednak zdarzać zbyt często, gdyż jeden ruch bardzo rzadko radykalnie zmienia ocenę stanu gry.

Listing 3.8: algorytm MTD

```
1  //Zwraca przewidywaną wypłatę gracza do którego należy ruch w stanie S
2  //S - aktualny stan gry, Depth - głębokość poszukiwań,
3  //FirstGuess - pierwsze przybliżenie rozwiązania
4  int MTD(Stan S, int Depth[, int FirstBeta]) {
5      int lowerbound = -∞; //dolna granica rozwiązania
6      int upperbound = ∞; //górna granica rozwiązania
7      int β = FirstBeta;   //górna granica okna poszukiwań
8      int val;             //wynik ostatniego testu
9      do {
10         val = AlfaBetaFS(S, Depth, β-1, β);
11         if (val < β)
12             upperbound = val;
13         else
14             lowerbound = val;
15         β = nextBeta(val, lowerbound, upperbound);
16     } while (lowerbound < upperbound);
17     return val;
18 }
```

3.8 Algorytmy z rodziny MTD

3.8.1 Wstęp

Algorytmy z rodziny MTD (ang. Memory-enhanced Test Driver) polegają na stopniowym zawężaniu przedziału potencjalnych wartości rozważanego węzła za pomocą serii wywołań fail-soft α-β z zerowym oknem (tj. takich, że $\beta - \alpha = 1$). Zauważmy, że każde takie wywołanie, na podstawie tw. 2, pozwala określić nowe dolne (równanie 3.4) albo górne (3.3) ograniczenie szukanej wartości. Gdy granice się zrównają, otrzymamy dokładną wartość węzła.

Na wydruku 3.8, funkcja $nextBeta$ ustala okno (konkretnie wartość β, zaś $\alpha = \beta - 1$) poszukiwań dla kolejnej iteracji. $FirstBeta$ to stała lub parametr funkcji (zależnie od konkretnego algorytmu) od której może zależeć pierwsze okno poszukiwań.

Jeśli interesuje nas jedynie jaki jest najlepszy ruch, zaś niekoniecznie jaka jest jego wartość, możemy zmienić warunek stopu (z linii 16), na następujący:

$$\forall N \in nast(S) \setminus B\colon \quad lowerbound_B \geqslant upperbound_N$$

gdzie:

$lowerbound_S$ (dla każdego $S \in \mathbb{S}$) dolna znaleziona dotychczas (podczas przeszukiwań grafu gry) granica wartości S.

$upperbound_S$ (dla każdego $S \in \mathbb{S}$) górna znaleziona dotychczas granica wartości S.

B najlepszy, znaleziony dotychczas ruch (tzn. taki że $lowerbound_B = max_{N \in nast(S)}(lowerbound_N)$)

S badany stan

Ponieważ algorytmy MTD wielokrotnie przeszukują tą samą część grafu gry, to bardzo ważne jest, by ich implementację (konkretnie to implementację używanego przez nie algorytmu α-β) wzbogacić o tablicę transpozycji[11] (por. roz. 4.2).

Niestety, algorytmy MTD trudno jest pogodzić z niektórymi heurystykami, np. z techniką odcięć w oparciu o pusty ruch (por. roz. 4.6). Dodatkowo, są one mało odporne na niestabilność przeszukiwania (por. roz. 4.9).

Dokładny opis algorytmów MTD (wraz z badaniami ich efektywności) zawiera raport [22] oraz [21].

3.8.2 MTD(f)

Algorytm MTD(n, f) (ang. Memory-enhanced Test Driver with node n and value f) lub krócej MTD(f) (dokładniej opisany w [20]) jest najpopularniejszym (i powszechnie uważanym za najlepszy) przedstawicielem rodziny MTD. Jest on średnio o kilka procent szybszy od PVS z tablicą transpozycji.

Do ustalenia okna kolejnego eksperymentu używa on wartości zwróconej przez fail-soft α-β (por. roz. 3.4) w poprzedniej iteracji (szczegóły przedstawia listing 3.9).

Listing 3.9: nextBeta dla MTD(f)

```
1 int nextBeta(int val, int lowerbound, int upperbound) {
2     return (val == lowerbound) ? val + 1 : val;
3 }
```

$FirstBeta$ ($-\infty < FirstBeta \leqslant \infty$) jest parametrem podawanym na wejściu algorytmu i powinien być jak najlepszym przybliżeniem szukanej wartości. Najczęściej wraz z MTD(f) używa się (nadrzędnie) algorytmu iteracyjnego pogłębiania. Wtedy, za $FirstBeta$ przyjmuje się wartość znalezioną podczas poprzedniej, płytszej iteracji albo (w pierwszej iteracji) 0.

3.8.3 MTD$+\infty$ czyli SSS*

SSS* (ang. State Space Search) to algorytm typu „najpierw najlepszy". Zaczyna od sprawdzenia jaką maksymalną wypłatę może osiągnąć gracz wykonujący ruch tzn. od badania w oknie $(+\infty - 1,\ +\infty)$ (stała $FirstBeta = +\infty$). Taki test powoduje rozwinięcie wszystkich węzłów typu maksimum i tylko po jednym typu minimum

[11] wywołanie takiego α-β z zerowym oknem to właśnie *Memory-enhanced Test*

(tj. zakłada, że rywal idącego gracza będzie zawsze wykonywał pierwszy wygenerowany ruch). Jego wynik jest nowym górnym ograniczeniem na wartość korzenia i wyznacza wartość β kolejnego testu (por. wydruk 3.10). Później, okno poszukiwań jest dalej konsekwentnie (i analogicznie) stopniowo przesuwane ku mniejszym wartościom, tzn. w każdej kolejnej iteracji jest ono ograniczone od góry przez (coraz mniejszą) wartość zwróconą przez algorytm fail-soft α-β w poprzedniej iteracji.

Listing 3.10: nextBeta dla SSS*

```
1 int nextBeta(int val, int lowerbound, int upperbound) {
2     return val;
3 }
```

3.8.4 MTD$-\infty$ czyli DUAL*

DUAL* działa analogicznie do SSS* ale „od drugiej strony", tzn. jest to algorytm typu „najepierw najlepszy" z punktu widzenia rywala gracza wykonującego ruch. W pierwszej iteracji przeszukiwanie następuje w oknie $(-\infty, \ -\infty + 1)$ (tj. $FirstBeta = -\infty + 1$). Kolejne (coraz większe) wartości zwracane przez fail-soft α-β wyznaczają α dla kolejnych poszukiwań (zaś $\beta = \alpha + 1$, por. wydruk 3.11).

Proszę zauważyć, że dla nieparzystej głębokości poszukiwać, w pierwszej iteracji DUAL* rozwinie średnio $\mathbb{B}$ razy mniej węzłów niż SSS* (gdyż w przypadku DUAL* tylko pierwszy następnik korzenia będzie przeszukany). Badania (wg. raportu [22]) pokazują, iż istotnie DUAL* ma w praktyce pewną przewagę nad SSS* (na ogół przegląda mniej węzłów).

Listing 3.11: nextBeta dla DUAL*

```
1 int nextBeta(int val, int lowerbound, int upperbound) {
2     return val + 1;
3 }
```

3.8.5 MTD-step

Ta wersja jest bardzo podobna do SSS* (także $FirstBeta = +\infty$) ale „przesuwa" okno poszukiwań szybciej o stałą $stepsize$ (listing 3.12).

Analogiczny algorytm można też stworzyć wzorując się na DUAL*.

Listing 3.12: nextBeta dla MTD-step

```
1 int nextBeta(int val, int lowerbound, int upperbound) {
2     return max(lowerbound + 1, val - stepsize);
3 }
```

3.8.6 MTD-bi czyli C*

W C* okno poszukiwań w kolejnych iteracjach znajduje się w środku przedziału potencjalnych wyników (listning 3.13), zaś stała $FirstBeta = (-\infty + \infty + 1)/2$. Algorytm ten, działa więc analogicznie do przeszukiwania połówkowego (binary search).

Listing 3.13: nextBeta dla MTD-bi

```
1 int nextBeta(int val, int lowerbound, int upperbound) {
2     //średnia arytmetyczna zaokrąglona do góry
3     return (lowerbound + upperbound + 1) / 2;
4 }
```

Rozdział 4

Jak efektywniej przeszukiwać graf gry?

4.1 Czas jest zasobem krytycznym

Co można zrobić, by przeszukać graf gry głębiej bez dodatkowych nakładów czasowych?

Przede wszystkim można przeglądać jego węzły w odpowiedniej kolejności[1] tak, by odciąć jak największą jego część np. przy pomocy algorytmu α-β lub pochodnych (por. roz. 3.3, 3.4 i 3.5).

Po drugie dobrze jest zajrzeć głębiej w ciekawsze gałęzie, szczędząc czasu na te mniej obiecujące. W szczególności nie należy przerywać rozwijania drzewa poszukiwań w sytuacjach, w których można wykonać ruchy znacznie zmieniające stan gry.

Cenny czas można też zaoszczędzić unikając redundantnych obliczeń. W wielu grach różne sekwencje ruchów prowadzą przecież do tej samej sytuacji, co wcale nie musi oznaczać konieczności wielokrotnego jej eksplorowania.

W dalszej części rozdziału opisano metody, najczęściej używane w celu osiągnięcia wyżej wymienionych celów. Temat ten, bardzo często poruszany jest także w literaturze, np. w [9, 12, 13, 15, 16, 18, 26].

4.2 Tablica transpozycji

4.2.1 Motywacja

Naturalną częścią algorytmu przeszukiwania grafu w głąb (a to w istocie robią algorytmy minimaksowe) jest zapamiętanie dla każdego wierzchołka grafu, czy był on już odwiedzony. Dzięki temu unika się wielokrotnego przeglądania tych samych węzłów

[1] najpierw te najbardziej obiecujące

(w przypadku gier zdarza się ono, gdy: z danej sytuacji można osiągnąć inną różnymi sekwencjami ruchów, w przypadku używania algorytmu iteracyjnego pogłębiania lub aspirującego okna, a także w przypadku stosowania innych algorytmów, które mogą wielokrotnie badać ten sam węzeł, np. na różne głębokości lub w różnych oknach[2]).

4.2.2 Zasada działania

Przestrzeń stanowa większości gier jest ogromna, co uniemożliwia zapisanie jakiejkolwiek informacji dla każdego ze stanów (na ogół nie dysponujemy pamięcią o wielkości rzędu $|\mathbb{S}|$). Jednak zazwyczaj interesuje nas tylko niewielka część grafu gry (konkretnie stany osiągalne z aktualnego sytuacji poprzez wykonanie co najwyżej kilku ruchów).

Najbardziej naturalnym podejściem wydaje się więc zapisywanie każdej ze sprawdzanych sytuacji zaraz po jej zbadaniu do pewnego zbioru (dalej zwanego tablicą transpozycji lub tablicą przejść).

Tablica transpozycji, oprócz podstawowej roli jaką jest unikanie redundantnych obliczeń, może spełnić też dodatkową (warunkiem jest zapamiętanie w niej najlepszych następników zapisanych stanów). W przypadku, gdy w tablicy znajduje się wpis dotyczący stanu S, jednak nie opisuje on dostatecznie dokładnie jego wartości, możemy przypuszczać, że mimo wszystko znaleziony wcześniej najlepszy następnik S i tym razem (przy innym oknie lub głębokości) okaże się najlepszy. Co za tym idzie, sprawdzenie tego następnika w pierwszej kolejności powinno zwiększyć liczbę odcięć.

Technika ta jest szczególnie cenna w połączeniu z iteracyjnym pogłębianiem, gdyż w kolejnych iteracjach najpierw próbowane są stany należące do głównego wariantu gry znalezionego w poprzedniej iteracji (por. roz. 3.6).

Tablicę przejść można też wykorzystać do znalezienia powtórek sytuacji, które w wielu grach oznaczają remis[3].

Kluczami identyfikującymi wpisy w tablicy transpozycji będą oczywiście same stany lub jakieś ich identyfikatory (np. wartości funkcji skrótu). Każdy wpis powinien dodatkowo zawierać:

- wartość stanu gry S, którego dotyczy wpis
- informację, czy zapisana wartość jest dokładna, czy też jest górną lub dolną granicą (wynikłą z obcięć) tej dokładnej; alternatywnie można zapisać okno (α, β) w jakim badany był stan albo (szczególnie w przypadku algorytmów z rodziny MTD, por. roz. 3.8) górną i dolną znalezioną granicę wartości stanu
- głębokość na jaką badany był stan S

[2] przykładem może być algorytm PVS (por. roz. 3.5) lub rodzina MTD (por. roz. 3.8)

[3] nawet jeżeli do remisu wymagane jest więcej niż jednokrotne powtórzenie sytuacji, jak jest np. w szachach, to już pierwsze jej powtórzenie implikuje kolejne i w konsekwencji remis, o ile żadna ze stron nie zmieni taktyki (a tak będzie gdy obie strony stosują tą samą, „optymalną" strategie)

- opcjonalnie: najlepszy następnik S lub jego identyfikator (np. numer w wygenerowanym wektorze ruchów)
- opcjonalnie: informacja czy dany węzeł drzewa poszukiwań leży na ścieżce od korzenia do węzła aktualnie sprawdzanego (umożliwia to wykrywanie powtórek sytuacji)

Listing 4.1 przedstawia przykład wzbogacenia o tablicę transpozycji algorytmu α-β. Metoda `load` wczytuje z TT rekord opisujący stan S. Jeśli takowy nie znajduje się w tablicy, zostaje do niej dodany (przy czym dla nowego rekordu: `type==NEW` i `is_repeated()==false`). Gdy wczytany rekord opisuje dostatecznie dokładnie wartość odwiedzanego węzła (tzn. w odpowiednim oknie i głębokości nie mniejszej niż aktualnie wymagana, por. wydruk 4.2), to wartość z TT jest zwracana. W przeciwnym razie stan jest badany tak jak w oryginalnym α-β, po czym otrzymany wynik zostaje ewentualnie zapamiętany (przy pomocy metody `save`, listing 4.3). W zależności od przyjętej strategii (wyniku działania `should_save` w metodzie `save`), można zawsze zapisywać nową wartość (`should_save` zawsze zwraca `true`) lub np. tylko wtedy gdy wynika ona z nie płytszego badania niż ta wcześniej zapisana w TT.

4.2.3 Realizacja

Wpisy w tablicy transpozycji są dość niewielkie, zaś jedyne operacje (słownikowe) jakie na niej wykonujemy, to dodawanie nowego wpisu (ewentualnie nadpisywanie istniejącego) i wyszukiwanie rekordu o zadanym kluczu[4]. Dlatego strukturą odpowiednią do jej zaimplementowania wydaje się być tablica haszująca z adresowaniem otwartym.

W swojej podstawowej postaci struktura ta jest zwykłą tablicą o stałym rozmiarze R[5], w której o miejscu lub ciągu miejsc, w które należy wstawić rekord decyduje tzw. funkcja mieszająca (zwana też haszującą). Każdemu kluczowi identyfikującym rekord przyporządkowuje ona indeks lub ciąg indeksów (tzw. *ciąg kontrolny*, który jest permutacją zbioru wszystkich indeksów). Początkowo wszystkie wpisy są puste (specjalna wartość oznacza pusty rekord). Nowy rekord wstawiany jest w pierwszą (w kolejności przypisanych do jego klucza indeksów) pustą komórkę. Wyszukanie klucza polega na sprawdzeniu jego obecności w kolejnych, wyznaczonych przez odpowiadający mu ciąg kontrolny, komórkach (napotkanie pustego miejsca w tablicy oznacza brak rekordu o zadanym kluczu).

Więcej informacji o tablicach haszujących można znaleźć w [4]. Znajduje się tam również (wraz z dowodem) następujące twierdzenie:

Twierdzenie 3 (o efektywności tablicy z haszowaniem)**.** *Jeśli współczynnik zapełnienia tablicy z haszowaniem wynosi* $\alpha = n/R < 1$ *(*n *to ilość zapisanych*

[4] w szczególności nigdy nie usuwamy rekordów

[5] przeważnie dużo mniejszym niż uniwersum kluczy, w naszym przypadku $R \ll |\mathbb{S}|$

Listing 4.1: algorytm alfa-beta z tablicą transpozycji

```
1  //Zwraca przewidywaną wypłatę gracza do którego należy ruch w stanie S
2  //S - aktualny stan gry, Depth - głębokość poszukiwań
3  //α, β - okno poszukiwań
4  int AlfaBeta(Stan S, int Depth, int α, int β) {
5      Record* TT_entry = TT.load(S);   //wczytujemy opis stanu z TT
6      int val = TT_entry->to_return(depth, α, β);
7      if (val ≠ UNKNOWN)
8          return val;
9      if (Depth == 0) {
10         val = ocena(S);
11         TT_entry->save(val, Depth, EXACT);
12         return val;
13     }
14     vector<Stan> N = nast(S);
15     if (N == ∅)
16         return wyplata(S);
17     TT_entry->set_repeated(true);     //zaznacz wystąpienie sytuacji
18     TTRecordType record_type = UPPER_BOUND;
19     sort(N); //krok opcjonalny
20     for (int i = 0; i < |N|; i++) {
21         int val = - AlfaBeta(N[i], Depth-1, -β, -α);
22         if (val ⩾ β) {
23             TT_entry->save(β, Depth, LOWER_BOUND);
24             return β;
25         }
26         if (val > α) {
27             α = val; //poprawa maksimum
28             record_type = EXACT;      //wynik będzie dokładny
29         }
30     }
31     TT_entry->save(α, Depth, record_type);
32     return α;
33 }
```

Listing 4.2: tablica transpozycji: metoda to_return

```
1  //Zwraca wartość z TT (jeśli rekord dostatecznie dokładnie opisuje stan)
2  //lub specjalną wartość UNKNOWN (w przeciwnym wypadku)
3  int Record::to_return(int depth, int alpha, int beta) {
4      if (TT_entry->is_repeated())      //sytuacja powtórzona
5          return 0;                     //oznacza np. remis
6      if (this->depth ⩾ depth)      //czy głębokość wystarczająca?
7          switch (this->type) {
8              case EXACT: //mamy dokładną wartość
9                  return this->value;
10
11         case UPPER_BOUND: //mamy granicę górną
12             if (this->value ⩽ alpha)
13                 return alpha;
14             break;
15
16         case LOWER_BOUND: //mamy granicę dolną
17             if (this->value ⩾ beta)
18                 return beta;
19             break;
20     }
21     return UNKNOWN;
22 }
```

Listing 4.3: tablica transpozycji: metoda save

```
1  void Record::save(int value, int depth, TTRecordType type) {
2      this->set_repeated(false);
3      //metoda should_save ocenia czy warto nadpisać wartość dla danego stanu
4      if (this->type ≠ NEW &&
5          !this->should_save(value, depth, type))
6          return;
7      this->value = value;
8      this->depth = depth;
9      this->type = type;
10 }
```

w tablicy niepustych rekordów), to oczekiwana liczba porównań kluczy w czasie wyszukiwania elementu, który nie występuje w tablicy, jest nie większa niż $1/(1-\alpha)$*, o ile spełnione jest twierdzenie o równomiernym haszowaniu.*

Twierdzenie o równomiernym haszowaniu dla tablic z adresowaniem otwartym zachodzi, gdy dla każdego klucza wszystkich z $R!$ permutacji zbioru indeksów jest równie prawdopodobne jako jego ciągi kontrolne.

W praktyce, przy odpowiednio dużej tablicy haszującej i funkcji mieszającej, choć w przybliżeniu spełniającej twierdzenie o równomiernym haszowaniu, oczekiwany czas każdej z wykonywanych na niej operacji wynosi $O(1)$.

Sam klucz obliczany jest na podstawie stanu gry, którego ma dotyczyć wpis. Niech $key\colon \mathbb{S} \to \mathbb{N}$ będzie funkcją za to odpowiedzialną (dla uproszczenia dalej zakładam, że klucze są liczbami naturalnymi). W idealnym przypadku, dla dowolnych $S_1, S_2 \in \mathbb{S}$ powinna zachodzić implikacja:

$$key(S_1) = key(S_2) \quad \Rightarrow \quad S_1 = S_2 \tag{4.1}$$

W praktyce jednak, ze względu na oszczędność pamięci, nie rzadko rezygnuje się z jej spełnienia. Zauważmy, że prawdziwość 4.1 wymaga istnienia $|\mathbb{S}|$ różnych kluczy, czyli na zapisanie każdego potrzeba średnio $log_2(|\mathbb{S}|)$ bitów i to przy idealnym, często trudnym do znalezienia i efektywnego zaimplementowania odwzorowaniu. Powszechnie stosowaną praktyką jest więc użycie kluczy mniejszych niż $log_2(|\mathbb{S}|)$ bitów i obliczanie ich tak, by równości $key(S_1) = key(S_2)$ oznaczała dużą szansę (ale nie pewność) na $S_1 = S_2$. Dopuszcza się tym samym występowanie (choć sporadyczne) pomyłek, polegających na odczytaniu i wykorzystaniu do obliczeń wpisu dotyczącego innego stanu niż aktualnie badany.

Przykładowo, w programach grających w szachy powszechnie używa się 64-bitowe *klucze Zobrista* (ang. *Zobrist key*). Każdej figurze białych i czarnych i dla każdego pola planszy przypisuje się ustaloną losową (najczęściej 64-bitową) wartość (zwykle w kluczu uwzględnia się również pole bicia w przelocie i flagi roszady). Klucz Zobrista to różnica symetryczna (xor) wszystkich wartości odpowiadających figurom na polach planszy. Proszę zauważyć, że gdy gdy dla sytuacji S mamy policzony klucz (k), to bardzo szybko możemy policzyć klucz dla każdej sytuacji S' podobnej do S, np. uzyskanej z S poprzez wykonanie ruchu. Wystarczy policzyć xor klucza k, z odpowiednimi wartościami dla figur postawionych i zdjętych z pól, których zawartość różni się pomiędzy S i S'. Dzieje się tak, ze względu na własności operacji xor ($\oplus$), min. (dla dowolnych k, d i d_2): $(k \oplus d) \oplus d = k$, $(k \oplus d) = (d \oplus k)$ i $(k \oplus d) \oplus d_2 = k \oplus (d \oplus d_2)$. Zatem, postawienie pewnej figury, oraz jej zdjęcie z danego pola, powodują takie same zmiany w kluczu.

4.2.4 Enhanced transposition cutoffs

Załóżmy, że badamy pewien stan S i dla jednego z jego następników, w tablicy przejść znajduje się wartość taka, że zostanie ona zwrócona jak tylko zaczniemy

badać ten węzeł. Jeśli w wyniku tego miało by nastąpić β-odcięcie w S, to nie ma sensu przeszukiwanie innych jego następników.

ETC (ang. Enhanced transposition cutoffs) polega na próbie wykonania β-cięcia w oparciu o wartości zapisane w tablicy transpozycji (zgodnie z powyższymi spostrzeżeniami) przed rozpoczęciem właściwego przeszukiwania potomków badanego węzła.

Ponieważ ETC wymaga dodatkowych obliczeń (odwołań do tablicy transpozycji), to należy go stosować tylko tam, gdzie potencjalny zysk (z wykonanego szybciej odcięcia) jest największy, czyli blisko korzenia drzewa poszukiwań.

4.3 Quiescence Search - unikanie efektu horyzontu

Jak już wspomniałem w roz. 3.1, z braku czasu, przeszukujemy jedynie fragment grafu gry (do określonej głębokości). Można by powiedzieć, że liście drzewa poszukiwań stanowią horyzont, czyli kres tego fragmentu grafu gry, który „widzimy" i na podstawie którego podejmujemy decyzję. Wartość korzenia zaś jest przepisana (wzdłuż głównego wariantu gry) z któregoś z liści.

Efekt horyzontu wynika z różnicy ocen stanu gry (liścia) pomiędzy tą obliczoną przez funkcję oceniającą, a jego prawdziwą wartością (wypłatą jaka po nim nastąpi przy optymalnej grze obu stron). Przy czym, znaczna zmiana oceny sytuacji może nastąpić tuż za horyzontem (o takim stanie mówimy że jest *niestabilny* lub *niecichy*). Przykładem dla szachów lub warcabów mogą być sytuacje, w których strona idąca ma bicie.

Można zmniejszyć negatywny wpływ efektu horyzontu na jakość wyniku, poprzez unikanie statycznej oceny niestabilnych stanów.

Przykładem algorytmu postępującego w ten sposób jest przedstawiony na wydruku 4.4 *quiescence search*[6] („szukający spokój"). Polega on na sprawdzeniu, czy dany stan jest cichy przed obliczeniem wartości funkcji oceniającej. Jeśli nie jest, stan ten przeszukiwany jest głębiej (aż do osiągnięcia stabilności). Przy czym, sprawdzane mogą być jedynie ruchy, które mogą powodować niestabilność (np. bicia). Jednak wtedy wynik należy obliczać na podstawie ocen stanów z całego dodatkowo odwiedzonego fragmentu grafu gry, a nie jedynie liści, gdyż statyczna ocena nieterminalnych stanów reprezentuje po prostu ocenę ich pominiętych, stabilnych następników.

4.4 Heurystyka historyczna

Heurystyka historyczna opiera się na prostym (i prawdziwym dla większości gier) spostrzeżeniu, że posunięcia najlepsze w wielu sytuacjach, są też całkiem dobre w innych

[6] Funkcje QuiescenceSearch(S, α, β) należy wywołać zamiast ocen(S) w algorytmie α-β i pochodnych

Listing 4.4: algorytm Quiescence Search

```
1  //Zwraca przewidywaną wypłatę gracza do którego należy ruch w stanie S
2  //S - aktualny stan gry
3  //α, β - największa i najmniejsza znaleziona wcześniej wartość
4  int QuiescenceSearch(Stan S, int α, int β) {
5      α = max(ocena(S), α);
6      if (α ⩾ β)
7          return β;
8      vector<Stan> N = nastU(S);   //generuje niestabilne następniki
9      for (int i = 0; i < |N|; i++) {
10         int val = - QuiescenceSearch(N[i], -β, -α);
11         if (val ⩾ β)
12             return β;
13         if (val > α)
14             α = val; //poprawa maksimum
15     }
16     return α;
17 }
```

(w których są dopuszczalne). Można zatem prowadzić statystykę, ile razy każde posunięcie okazywało się być najlepsze (lub też powodowało polepszenie oceny stanu lub β-odcięcie), zaś jej wyniki wykorzystać do porządkowania następników w algorytmie α-β (tak by najpierw przeszukiwać ruchy, które często okazywały się najlepsze).

Tablica historii ruchów (służąca do prowadzenia wspomnianej statystyki) dla większości gier może być indeksowana bezpośrednio (co zapewnia szybkość jej działania), np. w warcabach[7], rozgrywanych na 32 (lub 50)[8] polach istnieją tylko $32^2 = 1024$ (lub $50^2 = 2500$) różne posunięcia (konkretniej mówiąc różniące się polem z którego lub na które przesuwana jest bierka).

Im głębiej przeszukamy następniki danego węzła grafu gry, tym większa szansa, że wyznaczymy jego w rzeczywistości najlepszy następnik, co warto uwzględnić zwiększając licznik „dobroci" danego posunięcia w tablicy historii ruchów. Często zwiększa się go zatem np. o d^2 lub nawet o 2^d, gdzie d jest głębokością eksploracji rozpatrywanego stanu gry.

[7] w których tablicę historii możemy indeksować parą: pole z którego/na które przesuwamy warcab, tzn. może mieć ona postać: int HT[32][32]

[8] w przypadku warcabów 100 polowych

4.5 Heurystyka ruchów morderców

Heurystyka ruchów zabójców jest o oparta na tych samych przesłankach, co heurystyka historyczna (por. roz. 4.4).

Polega ona na zapamiętaniu dla każdego poziomu (głębokości) drzewa poszukiwań kilku (najczęściej dwóch) posunięć, które ostatnio spowodowały β-cięcie. Na danym poziomie, posunięcia te, o ile są możliwe do wykonania, próbowane są jako pierwsze.

4.6 Heurystyka odcięć w oparciu o pusty ruch

Heurystyka odcięć w oparciu o pusty ruch (ang. null-move pruning) opiera się na założeniu, że w większości sytuacji graczowi nie opłaca się rezygnować z ruchu. Jest ono zasadne tylko w niektórych grach[9], np. w szachach (w których opisywana metoda sprawdza się nadzwyczaj dobrze).

Sama metoda polega zaś na wykonaniu pustego ruchu (tj. zrezygnowaniu z ruchu) przed wykonaniem innych posunięć (nie przejmujemy się przy tym, iż zasady danej gry, np. szachów, mogą nie dopuszczać takiego postępowania). Powstałą pozycję przeszukuje się na głębokość $depth - 1 - c$, gdzie $depth$ - aktualna głębokość (por. listing 3.4), $c >= 0$ to stała (zazwyczaj $c = 2$ lub $c = 3$). Wynik tego sprawdzenia używamy jedynie do ewentualnego β-cięcia[10] (licząc na to, że skoro pusty ruch jest wystarczająco dobry by spowodować cięcie, to któryś z dopuszczalnych ruchów jest od niego nie gorszy i też by je spowodował). Jeśli ono nastąpi, w ogóle nie musimy generować „prawdziwych" następników aktualnego stanu.

Trzeba jednak pamiętać, że jeśli założenie heurystyczne nie będzie prawdziwe, to stosując odcięcia w oparciu o pusty ruch, możemy nieprawidłowo ocenić badany stan (o takich sytuacjach mówimy, że są typu *zugzwang*). Dlatego warto w jakiś sposób weryfikować otrzymywane przy jej użyciu rezultaty lub sprawdzać, czy jest zagrożenie nieprawdziwości wspomnianego założenia przed jej użyciem.

4.7 Baza debiutów

W początkowych fazach gry przeszukiwanie jej grafu w celu podjęcia decyzji przeważnie niewiele daje, gdyż większość ruchów zostaje oceniona niemalże tak samo, zaś oceny te obarczone są dużym błędem. Lepiej jest zatem grać na podstawie (istniejących dla większości gier) wypracowanych przez wieloletnie doświadczenia ekspertów lub wyliczonych[11] wcześniej schematów.

[9] w pozostałych (przykładem mogą być warcaby) nie należy korzystać z opisywanego algorytmu, gdyż może on spowodować błędne ocenianie stanów

[10] jeśli będzie nie mniejszy od β

[11] np. na podstawie zapisu dużej ilości gier dobrych graczy. W takiej bazie można znaleźć, które debiuty prowadziły do zwycięstwa częściej niż pozostałe i je stosować.

Wystarczy do tego prosta baza danych z zapisanymi pozycjami i najlepszymi posunięciami jakie można z nich wykonać.

4.8 Baza końcówek

Baza końcówek zawiera pozycje, które mogą pojawić się w końcowej fazie gry. Z każdą z nich powinna być zapisana wypłata, jaką zakończy się gra po bezbłędnej grze zawodników i (opcjonalnie) najlepsze posunięcie, jakie można w niej wykonać (lub alternatywnie, ilość ruchów do końca gry).

Jeżeli w trakcie przeglądania grafu napotkamy stan znajdujący się w bazie, to możemy natychmiast zwrócić odczytaną z bazy, prawdziwą wartość jego wypłaty.

Jeśli zaś gra znajduje się w opisanym w bazie stanie S, to wystarczy wykonać zapisany dla niego ruch (jeśli tego typu informacja znajduje się w bazie).

Jeśli dysponujemy informacją o ilość ruchów do końca gry, to najlepszy ruch jaki można wykonać w znajdującym się w bazie stanie S, możemy wyznaczyć za pomocą następującego algorytmu:

1. Wygeneruj zbiór N następników S ($N = nast(S)$).

2. Znajdź w N taki stan B, którego zapisana w bazie wartość wypłaty w_B jest najmniejsza.

   Jeśli w N znajduje się kilka stanów z wartością wypłaty w_B, to jako B można wybrać ten z nich, dla którego ilość ruchów do końca gry była minimalna (jeśli $-w_B > 0$) lub maksymalna (jeśli $-w_B \leqslant 0$), tzn. postępować zgodnie z zasadą że do wyniku korzystnego chcemy doprowadzić jak najszybciej, zaś remis lub porażkę chcemy jak najbardziej odwlec, dając przeciwnikowi więcej szans na pomyłkę.

3. Zwróć w wyniku B.

Implementując bazę końcówek będziemy musieli rozwiązać wiele problemów związanych z efektywnym przechowywaniem i przeszukiwaniem wielkich zbiorów danych. Zaś dla wielu gier (o zbyt wielu „końcówkach", np. Othello czy Go) w ogóle nam się to nie uda (baza końcówek nie ma dla nich zastosowania). Przykładowo, dla warcabów, których bliskie końca stany gry składają się zaledwie z kilku kamieni (których pozycje wyznaczają dany stan) jest ponad $3 * 10^9$ końcówek, w których na planszy stoi dokładnie 6 kamieni.

Z powyższych powodów, ilość informacji (szczególnie tej przechowywanej w pamięci operacyjnej) związanej z każdym rekordem w bazie, należy ograniczyć do niezbędnego minimum. Nie warto np. przechowywać opisu samego stanu. Zamiast tego lepiej jest ponumerować wszystkie końcówki, tzn. skonstruować funkcję[12] $i : \mathbb{S}_k \rightarrow$

[12] przykład takiej funkcji, dla 64-polowych warcabów, znajduje się w rozdziale 6.5.5

$\{0, 1, \ldots, M-1\}$ (gdzie $\mathbb{S}_k$ to zbiór końcówek, zaś $M \geqslant |\mathbb{S}_k|$) przyporządkowującą końcówką numery rekordów w bazie (najlepiej takie, aby $M = |\mathbb{S}_k|$). Nie ma też potrzeby przechowywania w pamięci całej bazy, np. informacja o tym, który ruch jest najlepszy w danej sytuacji może być wczytana dopiero, gdy do niej dojdzie.

Dla wielu gier baza końcówek została już przez kogoś wygenerowana. Jeśli jednak przyjdzie nam stworzyć ją samemu, dobrze jest zacząć jej wypełnianie od rozważenia stanów bliskich końcowi gry (np. w warcabach mogą być to sytuacje, w których każdy z graczy dysponuje tylko jednym kamieniem). Po jego uzupełnieniu, możemy poszerzyć naszą bazę o kolejny fragment (posługując się tym, co już w niej się znajduje). Ważne jest, by wszystkie następniki, każdego ze stanów składających się na dodawany fragment F (czy to do pustej, czy też częściowo wypełnionej bazy), znajdowały się w bazie lub w F. Obliczenia kontynuujemy aż do uzyskania zadowalającej ilości wpisów.

Samo poszerzenie bazy o F można uczynić w następujący sposób (zakładam, że w każdym jej rekordzie zapisana jest wypłata gracza posiadającego ruch w sytuacji opisywanej przez ten rekord oraz ilość ruchów do końca gry):

1. Inicjalizacja. W każdym rekordzie opisującym stan końcowy z F, ustaw wartość wypłaty na określoną w przepisach gry i ilość ruchów do końca gry na 0. Dla wszystkich pozostałych nowych rekordów, ustaw wartość wypłaty na NIEZNANĄ.

2. Dla każdego rekordu opisującego stan R z F o NIEZNANEJ wartości wypłaty:

   Jeśli wartości wypłat dla każdego stanu z $N = nast(R)$ (zbioru następników R) są już ustalone (nie są NIEZNANE), to znajdź w N taki stan B, którego wartość wypłaty w_B jest najmniejsza (B jest najlepszym następnikiem R). Zapamiętaj dla R wartość wypłaty $-w_B$ oraz ilość ruchów do końca, równą ilości ruchów do końca dla stanu B powiększoną o 1.

   Jeśli w N znajduje się kilka stanów z wartością wypłaty w_B, to jako B można wybrać ten z nich, dla którego ilość ruchów do końca gry była minimalna (jeśli $-w_B > 0$) lub maksymalna (jeśli $-w_B \leqslant 0$), tzn. postępować zgodnie z zasadą, że do wyniku korzystnego chcemy doprowadzić jak najszybciej, zaś remis lub porażkę chcemy jak najbardziej odwlec.

3. Jeśli wartości wypłaty dla któregokolwiek rekordu została ustalona w bieżącej iteracji (w pkt. 2), to przejdź do punktu 2.

4. Oznaczmy zbiór rekordów o NIEZNANEJ (w tej chwili) wartości wypłaty[13] przez LOOPS.

   W każdym rekordzie z LOOPS ustaw wartość wypłaty na wynikającą z powtórzenia ruchów (najczęściej odpowiadające remisowi 0).

[13] takie rekordy mogą się zdarzyć ze względu na cykle w grafie gry

Ilości ruchów do końca w tych rekordach można nadać specjalną wartość $+\infty$.

5. Dla każdego rekordu z LOOPS, opisującego stan R:

   Znajdź w $nast(R)$ taki stan B którego, zapisana w bazie wartość wypłaty w_B jest najmniejsza.

   Jeśli $-w_B$ jest różna od wartość wypłaty zapisanej dla R, to ustaw dla R nową wartość wypłaty, równą $-w_B$ i nową ilość ruchów do końca (obliczoną w sposób analogiczny do opisanego w pkt. 2)

6. Jeśli wartości wypłaty dla któregokolwiek rekordu została poprawiona w bieżącej iteracji (w pkt. 5), to przejdź do punktu 5.

Dla każdego stanu opisanego w bazie (po jej wypełnieniu), można opcjonalnie wyznaczyć i zapamiętać najlepsze posunięcie (algorytmem opisanym na początku rozdziału), rezygnując równocześnie z przechowywania informacji o ilości ruchów do końca.

Prostrzy algorytm generowania bazy końcówek, ale jedynie dla gier o 3 możliwych wartościach wypłaty (dla wygranej, remisu i przegranej), opisany jest w [6].

4.9 Uwaga na niestabilności przeszukiwania

Użyciem niektórych technik (np. tablicy transpozycji, czy heurystyki odcięć w oparciu o pusty ruch), narażamy proces przeszukiwania grafu gry (np. algorytmem α-β) na wystąpienie niestabilności, np.

- dwa kolejne wywołania α-β, dla tego samego węzła, okna i głębokości poszukiwań dadzą inne wyniki
- dolne ograniczenie wartości pozycji wynikające z jednego wywołania α-β (por. tw.2) będzie większe od górnego ograniczenia pochodzącego z innego wywołania α-β

Powody niestabilności mogą być różne. Stosując tablicę transpozycji, godzimy się na to, iż znaleziona wartość węzła, nie zależy jedynie od parametrów poszukiwania, ale także od aktualnej zawartości tablicy. Przykładowo, czasami α-β odczyta z tablicy przejść i zwróci wartość węzła dokładniejszą od tej określonej wymaganą głębokością poszukiwań. Sporadycznie zdarzą się też pomyłki wynikłe stosowaniem skróconych kluczy, powodujące odczytanie i użycie rekordu opisującego inny stan niż aktualnie badany.

Wykrywanie powtórzeń sytuacji także może spowodować inną ocenę węzła, w zależności od tego, kiedy go badamy (występuje *interakcja z historią grafu* - ang. GHI, *Graph History Interaction*).

Rozdział 5

Funkcja oceniająca

5.1 Podstawy konstrukcji funkcji oceniającej

Jak już wcześniej wspomniałem, funkcja oceniająca $ocena : \mathbb{S} \to \Re$, nazywana także funkcją heurystyczną, przyporządkowuje każdemu stanowi $S \in \mathbb{S}$ przewidywaną wartość wypłaty, jaka nastąpi po tym, gdy gra znajdzie się w stanie S (i dalej zawodnicy będą grali w sposób optymalny). Przy czym, wartość ta jest obliczana w sposób statyczny (bez sprawdzania następników S), na podstawie wybranych (najczęściej przez eksperta) cech stanu S.

Niech C będzie zbiorem cech. Załóżmy ponadto, że są one wyrażone liczbowo i oznaczmy ich wartości dla stanu S przez $C_0(S), C_1(S), \ldots, C_{n-1}(S) \in \Re$, gdzie $n = |C|$.

Najczęściej używa się funkcji oceniających, będących ważoną sumą cech:

$$ocena(S) = \sum_{i=0}^{|C|-1} w_i C_i(S) \tag{5.1}$$

gdzie $w_i \in \Re$ dla $i = 0, 1, \ldots, |C| - 1$ to waga i-tej cechy.

Wagi poszczególnych cech mogą być dobrane przez eksperta, ale istnieją też automatyczne metody ich doboru (jest to problem optymalizacji, por. np. roz. 5.2).

Czasami stosuje się bardziej złożone funkcje oceniające lub sztuczne sieci neuronowe, które dobrze aproksymują nieznane zależności (na których wejście podaje się wartości cech, na wyjściu otrzymuje się zaś ocenę, por. np. roz. 5.4).

Nierzadko, w programy wbudowuje się kilka funkcji, z których do oceny, w zależności od sytuacji, wybierana jest jedna. Przykładowo można używać innej funkcji na początku partii, innej w fazie środkowej, a jeszcze innej na jej końcu (zauważmy, że np. w szachach król w centrum jest wiele wart pod koniec partii, podczas gdy na jej początku, oznacza niemalże porażkę). Przy czym, należy unikać stosowania

różnych funkcji oceniających w trakcie przeszukiwania pojedynczego węzła (chyba, że umiemy przeskalować ich wartości tak, by były ze sobą rozsądnie porównywalne).

Nie należy też zapominać, że stosując wprost algorytmy w notacji negamaksowej (por. roz. 3.2) zakładamy zerową sumę wypłat. Fakt ten należy uwzględnić dobierając funkcję oceny, w szczególności wartości, jakie ona zwraca. Wartości te powinny się zawierać pomiędzy największą i najmniejszą możliwą wypłatą.

W wielu grach są tylko trzy możliwe wypłaty, jakie może otrzymać gracz: $+W$ za zwycięstwo, 0 za remis i $-W$ za przegraną (gdzie $W \in \Re_+$). W takim wypadku wartości funkcji oceniającej powinna być z przedziału $[-W, W]$ i mogą być interpretowane następująco (dla stanu S w którym ruch należy do gracza G):

- $ocena(S) = W$ - na pewno wygra G
- $ocena(S) \in (0, W)$ - zapewne wygra G (wartość bezwzględna wyraża stopień pewności)
- $ocena(S) = 0$ - zapewne będzie remis
- $ocena(S) \in (-W, 0)$ - zapewne przegra G (wartość bezwzględna wyraża stopień pewności)
- $ocena(S) = -W$ - na pewno przegra G

Dalsza część tego rozdziału charakteryzuje wybrane metody automatycznego konstruowania funkcji oceniających. Temat ten, bardziej szczegółowo przedstawiony jest w książce [14].

5.2 Metoda Samuela doboru współczynników funkcji oceniającej

W latach 50., w laboratoriach IBM, pod kierownictwem Arthura L. Samuela powstał program grający w warcaby, który jest uznawany za pierwszy program uczący się.

Program ten używał algorytmu minimaksowego[1] oraz funkcji oceniającej postaci 5.1. Dodatkowo, komputerowy gracz Samuela był w stanie uczyć się na własnych doświadczeniach poprzez modyfikowanie wag używanej funkcji oceniającej[2]. Zmieniał je tak, by dla sytuacji w której przed chwilą wykonywał ruch, podawała ona wartość bliższą tej (dokładniejszej) właśnie obliczonej algorytmem minimaksowym.

[1] niezbyt szybkie wtedy komputery uniemożliwiały przeszukiwanie grafu gry zbyt głęboko. Dlatego Samuel stosował liczne ulepszenia, min. pewną odmianę Quiescence Search (por. roz. 4.3) i coś na kształt tablicy transpozycji (por. roz. 4.2), do której zapisywał (ze względu na wyraźnie ograniczoną ilość pamięci) wybrane wyniki dla korzenia drzewa poszukiwań

[2] zbiór używanych cech też był zmieniany na podstawie ich wyliczanego skorelowania

Waga jednej z cech wyraźnie skorelowanej z wygraną (konkretnie przewagi materialnej, tj. w pionach i damkach), miała jednak stałą, dużą wartość. Było to niezbędne by ukierunkować zmiany pozostałych wag na poprawę funkcji oceniającej[3].

W eksperymencie Samuela program rozgrywał pojedynki sam ze sobą (w roli graczy A i B) w celu wypracowania dobrych wag (współczynników) funkcji oceniającej. Przy czym, jedynie funkcja gracza A była modyfikowana po każdym ruchu. Jeśli zwyciężył on partię, to współczynniki jego funkcji oceniającej były kopiowane do B, w celu wyrównanie poziomu obu zawodników. Jeśli zaś przegrał trzy razy z rzędu, to najwyższy współczynnik jego funkcji oceniającej był zerowany, co umożliwiało wydostanie się z lokalnego minimum.

Bardziej szczegółowy opis metody Samuela znajduje się w [24].

5.3 GLEM - ogólny liniowy model oceniania

GLEM (ang. Generalized Linear Evaluation Model) to (opisany w [2]) pomysł Michaela Buro który z powodzeniem zastosował go do gry w Othello. Jego program, LOGISTELLO, wygrał w 1997 roku pojedynek z mistrzem świata.

W GLEM, funkcja oceniająca jest kombinacją liniową pewnej liczby cech stanu gry. GLEM umożliwia łatwy (automatyczny) dobór tych cech i ich wag na podstawie zbioru uczącego, składającego się z ocenionych pozycji[4].

Niech A będzie zbiorem atomowych cech stanów gry, z których każda przyjmuje wartość całkowitą (np. w warcabach taką atomową cechą może być wartość bierki stojącej na konkretnym polu, przykładowo: 2 dla naszej damki, 1 dla naszego kamienia, 0 oznacza pole puste, -1 - kamień przeciwnika, zaś -2 - damkę przeciwnika).

Cechy atomowe są bloczkami, z których budowane są bardziej złożone cechy (konfiguracje).

Konfiguracja opisana jest przez zbiór atomowych cech $a_1, a_2, \ldots, a_l \in A$ wraz z określonymi dla nich konkretnymi wartościami $v_1, v_2, \ldots, v_l \in \mathbb{Z}$ (np. pole 1 ma cechę 2, a pole 4 cechę -1).

Mówimy, że konfiguracja c jest aktywna w stanie gry S (lub równoważnie, że zdanie $c(S)$ jest prawdziwe), jeśli ma on wszystkie wyspecyfikowane przez nią cechy równe podanym wartością, tzn.

$$c(S) = [(a_1(S) = v_1) \wedge (a_2(S) = v_2) \wedge \ldots \wedge (a_l(S) = v_l)]$$

.

[3] np. fatalna funkcja oceniająca o wszystkich wagach równych 0 dawała by niemalże całkowitą „niezależność" od głębokości przeszukiwań

[4] Wiedza w tej formie jest w praktyce dużo łatwiejsza to zdobycia niż wiedza sformalizowana. Istnieją zapisy różnych gier, czasami wraz z analizami ekspertów. Jeśli nie dysponujemy takimi analizami, za ocenę wszystkich stanów gry (pod warunkiem że była ona rozgrywana pomiędzy dwoma niezłymi graczami) możemy przyjąć wartość ostatniego jej stanu, tj. wypłaty jaka nastąpiła po zakończeniu gry.

Dla danej pozycji S i konfiguracji c, definiujemy:

$$val(c(S)) = \begin{cases} 1 & \text{gdy konfiguracja c jest aktywna w S} \\ 0 & \text{w przeciwnym wypadku} \end{cases}$$

Funkcja oceny w GLEM jest postaci:

$$ocena(S) = g\left(\sum_{i=0}^{N-1} w_i val(c_i(S))\right)$$

gdzie:

c_i i-ta konfiguracja

w_i waga przypisana i-tej konfiguracji

$g : \Re \to \Re$ dobierana do konkretnego problemu funkcja przejścia, rosnąca i różniczkowalna, np. $g(x) = 1/(1 + e^{-x})$ której pochodną jest $g(x)(1 - g(x))$

Zbiór konfiguracji $C = \{c_0, c_1, \ldots, c_{N-1}\}$ wykorzystywanych przez funkcję oceny może być dobrany za pomocą algorytmu 5.1. Wybiera on konfiguracje, które są aktywne dostateczną ilość razy (konkretnie n lub więcej) w zbiorze treningowym E. Jego działanie opiera się na iteracyjnym dodawaniu coraz bardziej złożonych konfiguracji do zbioru wynikowego C, aż do momentu gdy kolejne konfiguracje będą zbyt szczegółowe aby występować w zbiorze uczącym przynajmniej n razy.

Algorytm na wydruku 5.1 ma zbyt dużą złożoność by stosować go bezpośrednio. Buro, w [2] zaproponował wiele praktycznych ulepszeń.

Po wybraniu zbioru cech pozostaje jeszcze problem doboru ich wag $w = [w_0, w_1, \ldots, w_{N-1}]$. W GLEM, czyni się to tak, aby zminimalizować średni błąd kwadratowy, który wyraża się wzorem:

$$E(w) = \frac{1}{n}\sum_{i=1}^{n}(r_i - ocena(s_i))^2$$

gdzie r_i (dla $i = 1, 2, \ldots, n$) jest wzorcową oceną i-tej pozycji (s_i) ze zbioru treningowego (wielkości n).

Można wykorzystać do tego (sugerowaną przez Buro) iteracyjną metodę najszybszego spadku, która w każdym kroku poprawia wektor wag w kierunku ujemnego gradientu funkcji błędu, tj. dodając do niego:

$$\Delta w = -\alpha w \nabla_w E$$

gdzie $\alpha > 0$ to długość kroku, $\nabla_w E$ oznacza wektor pochodnych cząstkowych po wagach $(\frac{\delta E}{\delta w_i})$ zaś w - aktualny wektor wag. Początkowe wartości wag można wylosować.

Listing 5.1: algorytm generowania zbioru konfiguracji

```
1  //Zwraca konfiguracje na zbiorze A, które są aktywne przynajmniej n razy
2  //w zbiorze przykładów treningowych E
3  //match(e, E) oznacza ilość sytuacji z E w których konfiguracja e jest aktywna
4  set<Conf> GenConf(set<Stan> E, int n) {
5      //R to zbiór konfiguracji składających się z pojedynczych cech atomowych
6      //i wartości, aktywnych przynajmniej n razy w E
7      const set<Conf> R =
8          {{f(·) = k}; f ∈ A, k ∈ range(f), match({f(·) = k}, E) ⩾ n};
9      set<Conf> C = R; //zbiera wszystkie poprawne konfiguracje
10     set<Conf> N = R; //konfiguracje stworzone w poprzedniej iteracji
11     while (N ≠ ∅) {
12         M = ∅;
13         foreach (c ∈ N, d ∈ R) {
14             e = c ∪ d;
15             if (match(e, E) ⩾ n)
16                 M = M ∪ {e};
17         }
18         N = M;
19         C = C ∪ N;
20     }
21     return C;
22 }
```

5.4 TD-Gammon - neuronowy mistrz backgammona

Rolę funkcji oceniających mogą pełnić sztuczne sieci neuronowe, które, jak wiadomo, dobrze aproksymują nieznane zależności. Na wejście takiej sieci podawany jest stan gry lub pewne jego cechy, zaś na wyjściu otrzymywana jest ocena tego stanu. Dużym sukcesem w tego typu postępowaniu może poszczycić się Gerald Tessauro [25], którego program grający w backgammona, TD-Gammon osiągnął poziom arcymistrzowski (nawiązał wyrównaną walkę min. z byłym mistrzem świata, Billem Robertim).

Na wejście używanego przez TD-Gammona trójwarstwowego perceptronu, w początkowych wersjach tej gry, podawana była wprost sytuacja na planszy (nie wykorzystano żadnej dodatkowej wiedzy dziedzinowej).

Warstwa wejściowa składała się ze 194 neuronów: po 8 dla każdego pola planszy (po 4 dla określenia liczby pionów każdego z graczy na tym polu), 2 (po 1 dla gracza) dla pionków na polu „więzienie", 2 (po 1 dla gracza) dla pionków które opuściły już planszę, oraz 2 określające do którego gracza należy ruch.

Warstwa wyjściowa składała się z 4 neuronów. Każdy pokazywał oczekiwaną maksymalną wypłatę, rozumianą jako szansę wygranej, kolejno: pierwszego gracza („czerwonego"), drugiego gracza, wysokiej wygranej pierwszego gracza (w której drugi został „gammoned") i wysokiej wygranej drugiego gracza. Nie uwzględniono osobno szansy na wygraną z 3 punktów, jako iż zdarza się ona niezwykle rzadko.

W warstwach ukrytej i wyjściowej użyto sigmoidalnej funkcji[5] przejścia, która zwraca wartości z przedziału $[0, 1]$.

W celu wytrenowania sieci neuronowej program rozgrywał partie sam ze sobą, przy każdej partii modyfikując (początkowo losowe) wagi w sieci, zgodnie z algorytmami wstecznej propagacji błędu, dla której błąd był wyznaczany na podstawie metody różnic w czasie TD(λ).

W trakcie każdej partii, której przebieg wyznaczały kolejne pozycje $x_1, x_2, \ldots, x_n$, o ocenach (podanych przez sieć), odpowiednio $Y_1, Y_2, \ldots, Y_n$, wagi sieci były modyfikowane tak, by zmniejszyć różnice pomiędzy ocenami kolejnych stanów gry[6] $(Y(t+1) - Y(t))$, wg. następującej zależności:

$$w(t+1) = w(t) + \alpha(Y(t+1) - Y(t)) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w Y_k$$

gdzie:

t numer sytuacji, $t \in \{1, 2, \ldots, n-1\}$

$w(t)$ wagi neuronu

$Y(t)$ odpowiedź sieci w chwili t

λ $(0 \leqslant \lambda \leqslant 1)$ (ang. *elligiblity trace*) współczynnik określający jak „daleko" w przeszłość aktualny błąd wpłynie na zmianę wag (równy 1 oznacza równy wpływ na wszystkie poprzednie oceny, zaś 0 wpływ jedynie na ostatnią ocenę)

$\nabla_w Y_k$ wektor pochodnych wektora wyjściowego po poszczególnych wagach neuronu

α niewielka stała zwana współczynnikiem uczenia

Po zakończeniu partii, wagi były modyfikowane z uwzględnieniem jej wyniku. Zastosowano do tego ten sam wzór (dla $t = n$), w którym jednak różnicę $Y(t+1) - Y(t)$, zastąpiono różnicą $Z - Y(n)$ (można przyjąć oznaczenie $Y(n+1) = Z$), gdzie Z oznacza zakodowany zgodnie z specyfikacją wyjścia sieci wektor określający wynik partii.

Mimo, iż początkowe partie miały dość losowy przebieg (a co za tym idzie trwały dość długo), to po rozegraniu pewnej ich ilości[7] program uzyskał poziom mistrzowski.

[5] $\sigma(x) = 1/(1 + e^{-x})$

[6] co jest istotą nauki na podstawie różnic w czasie - ang. Temporal Difference Learning

[7] w zależności od wersji TD-Gammona trening trwał od 300000 do 1500000 partii

Na podkreślenie zasługuje fakt, iż udało się tego dokonać bez „ręcznego" wprowadzenia do programu wiedzy eksperckiej dotyczącej gry (przynajmniej w początkowych wersjach TD-Gammona, w których sytuacja na planszy była podawana niemalże bezpośrednio na wejście sieci). Trzeba jednak pamiętać, iż do tego sukcesu przyczyniły się specyficzne cechy samego Backgammona[8], w tym głównie:

- element losowy, dzięki któremu uzyskano dużą rozmaitość przebiegów i sytuacji podczas procesu nauczania, oraz uniknięto wpadania w lokalne optima. Brak determinizmu sprawia też, że funkcja wypłaty dla kolejnych stanów partii jest w zasadzie ciągła, zaś pomyłka przy wyborze ruchu nie powoduje znacznego pogorszenia sytuacji na planszy
- możliwość wykonywania jedynie ruchów „do przodu", co zapewnia rozstrzygnięcie partii w rozsądnym czasie, nawet przy losowym wybieraniu posunięć.

[8] podobne podejście nie sprawdziło się aż tak dobrze w innych grach, takich jak np. szachy czy Go

Rozdział 6

Program grający w warcaby Little Polish... krok po kroku

6.1 Podstawowe informacje o warcabach brazylijskich

Warcaby to gra planszowa posiadająca wiele odmian, spośród których najpopularniejsze obecnie są w świecie warcaby polskie, nazywane też międzynarodowymi (100 polowe).

My jednak zajmiemy się inną odmianą warcabów, odmianą klasyczną. Warcaby klasyczne, grane na 64 polowej warcabnicy z użyciem międzynarodowych zasad gry w warcaby[1], podobnie jak warcaby polskie, są od 1991 r. w Polsce dyscypliną sportową. Od kilkunastu lat ta odmiana warcabów nazywana jest w świecie warcabami brazylijskimi. Mało kto jednak wie, że na początku XX wieku w angielskojęzycznej literaturze nazywano te warcaby „minor polish draughts" lub „little polish draughts" (skąd pochodzi nazwa programu) czyli „małymi warcabami polskimi".

Z punktu widzenia teorii gier, warcaby klasyczne to gra dwuosobowa, deterministyczna, z pełną informacją. Średnia ilość posunięć jaką ma do wyboru gracz wykonujący ruch, wynosi nieco ponad $5,3$ (szczegóły w roz. 6.5.2).

To wszystko sprawia, iż warcaby brazylijskie są jedną z prostszych gier logicznych. Można by się nawet pokusić nie tylko o napisanie programu grającego na mistrzowskim poziomie, ale nawet o rozwiązanie tej gry (inaczej mówiąc, o obliczenie dokładnej wartości minimaksowej dla stanu początkowego gry), podobnie jak warcabów amerykańskich (konsekwentnie jest to czynione w ramach projektu *Chinook*).

Mimo to, ze względu na niewielką popularność poza Brazylią i Polską, powstało dość niewiele programów dobrze grających w tą odmianę warcabów.

[1] można je znaleźć w dodatku A

6.2 Założenia projektowe

Pisany program, działając na przeciętnym komputerze osobistym, powinien zapewnić przyjemną rozgrywkę (komputer powinien w miarę szybko wykonywać ruchy), na możliwie wysokim[2] poziomie (najlepiej regulowanym przez użytkownika). Pożądane jest także, by nie zabierał przy tym całych zasobów systemu (zużycie pamięci nie powinno przekroczyć kilkudziesięciu megabajtów).

6.3 Kilka uwag natury projektowej

Pisanie własnego programu grającego jest zajęciem wciągającym, ale i dość trudnym. Miło jest obserwować postępy w sile gry własnego programu, po wprowadzeniu do kodu kolejnych poprawek i rozszerzeń, które, jak się wydaje, można by wprowadzać bez końca. Jednak, trzeba pamiętać, iż ewentualnie popełnione przy tym błędy mogą objawiać się tylko w specyficznych sytuacjach w grze i minie dużo czasu zanim je w ogóle zauważymy, nie wspominając już o domyśleniu się, w którym momencie je popełniliśmy.

Dlatego, pisanie gry logicznej, dobrze jest zacząć od zaprojektowania, zaimplementowania i przetestowania niezbędnego do jej działania minimum[3], czyli:

- struktury reprezentującej stan gry (np. sytuację na planszy, zbiory posiadanych przez jej uczestników kart, itp.)
- generatora ruchów (wyznaczającego zbiór decyzji możliwych do podjęcia przez gracza w każdej sytuacji)
- funkcji oceniającej (początkowo może być bardzo prosta)

Posiadając już składniki niezbędne do implementacji algorytmu α-β, możemy wzbogacić o nią nasz program i rozpocząć rywalizację[4]. W jej trakcie, powinniśmy postarać się dostrzec wady programu (np. w jakich fazach gry radzi sobie najsłabiej, czy nie „myśli" zbyt długo nad wyborem posunięcia, itd.) i następnie spróbować je usunąć, wprowadzając do niego odpowiednie algorytmy lub heurystyki. Nie wolno jednak zapomnieć, o dokładnym jego przetestowaniu po wprowadzaniu każdej poprawki. Specjalnie przygotowane, automatycznie wykonywane testy mogą okazać się w tym pomocne (takie podejście, zwane jest *programowaniem sterowanym testami* - ang. *Test Driven Development*).

Uwaga: W powyższym opisie pominięto niektóre, istotne z punktu widzenia projektowania gier zagadnienia, nie związane bezpośrednio z podejmowaniem przez

[2] nie musi być arcymistrzem, ale powinien stawiać opór nawet dobrym warcabistom
[3] jego realizacja na potrzeby programu Little Polish, opisana jest w rozdziale 6.4
[4] lub jakiś inny sposób testowania

	28		29		30		31
24		25		26		27	
	20		21		22		23
16		17		18		19	
	12		13		14		15
8		9		10		11	
	4		5		6		7
0		1		2		3	

Rysunek 6.1: Numeracja pól warcabnicy. Na początku partii białe piony zajmują pola o numerach od 0 do 11 włącznie, zaś czarne od 20 do 31.

sztucznego gracza decyzji[5], jak np.: oprawa graficzna, sposób komunikacji z użytkownikiem, itd. W przypadku niektórych gier (np. szachów), autor nie musi zresztą sam się nimi zajmować, gdyż istnieją ich wolno dostępne realizacje. Wtedy, jego rola sprowadza się do wyposażenia modułu sztucznej inteligencji w interfejs programowy (najczęściej odpowiednio skonstruowanej biblioteki dynamicznej) przewidziany przez twórców używanego interfejsu użytkownika.

6.4 Implementacja podstawowych elementów gry

6.4.1 Zorientowana bitowo reprezentacja sytuacji na warcabnicy

W rozważanej wersji klasycznej warcabów, partię rozgrywa się na 64-polowej warcabnicy. Przy czym, jedynie połowa jej pól, konkretnie czarne 32 pola, są wykorzystywane (aktywne) podczas gry. Do zapamiętania sytuacji na planszy, wystarczy więc zapisać, co znajduje się na tych polach. Rysunek 6.1 przedstawia ich numerację.

Na 32-bitowym słowie maszynowym możemy zapisać binarną cechę dla każdego z 32 używanych pól warcabnicy, ustawiając i-ty ($i \in \{0, 1, \ldots, 31\}$) bit tego słowa na 1, wtedy i tylko wtedy gdy i-te pole posiada daną cechę.

Sytuacja na warcabnicy jest jednoznacznie opisana przez trójkę binarnych cech określonych dla każdego z jej aktywnych pól:

- Czy na danym polu znajduje się biała bierka?
- Czy na danym polu znajduje się czarna bierka?
- Czy na danym polu znajduje się damka?

Można więc ją zapisać używając trzech 32-bitowych masek (w sumie na 96 bitach), na których zaznaczone są (odpowiednio):

[5] czyli główną tematyką poruszaną w tej pracy

- pola na których znajdują się białe bierki (`white`)
- pola na których znajdują się czarne bierki (`black`)
- pola na których znajdują się damki (`kings`)

Taka, *zorientowana bitowo* reprezentacja (ang. *bitboard*) daje nie tylko oszczędność pamięci, ale umożliwia także szybki dostęp (przy użyciu prostych operacji bitowych) do wielu informacji, np.

- `white&~kings` - położenie białych pionków
- `black&~kings` - położenie czarnych pionków
- `white&kings` - położenie białych damek
- `black&kings` - położenie czarnych damek
- `white|black` - pola zajęte przez warcaby
- `~(white|black)` - pola wolne
- `white&white_last_line` - białe piony na polu promocji (`white_last_line=0xF0000000`) które powinny stać się damkami (`kings|=white&white_last_line`) (analogicznie dla czarnych: `black&black_last_line`, `black_last_line=0x0000000F`)

6.4.2 Generator ruchów

Generator ruchów jest jedną z najczęściej wywoływanych funkcji, dlatego jego staranna, efektywna implementacja może się przyczynić do szybszego podejmowania decyzji przez naszego komputerowego gracza.

W Little Polish praca generatora ruchów opiera się min. na spostrzeżeniach, że dla danego kierunku poruszania (np. w lewo-górę) oraz parzystości linii z której się poruszamy[6] pole docelowe ma stałe przesunięcie w stosunku do źródłowego, i tak np. następująca funkcja zwraca pole (lub pola) po ruchu w górę lewo:

```
1 int32u upleft(const int32u p) {
2     return ((p & upleft3_from)<<3) | ((p & upleft4_from)<<4);
3 }
```

gdzie:

p jest maską z zaznaczonym polem źródłowym (lub polami źródłowymi)

[6] Dobrym pomysłem byłoby też takie ponumerowanie pól, by wykluczyć ten warunek. W przypadku warcabów wymagałoby to jednak zrezygnowanie z ciągłości numeracji, a co za tym idzie (w przypadku używania zorientowanych bitowo reprezentacji) użycie dłuższych słów maszynowych do reprezentowania cech planszy.

Listing 6.1: funkcja zliczająca ilość bitów ustawionych w 32 bitowym słowie

```
1 //Zwraca ilość bitów równych 1 w u32.
2 //Używa tablicy: unsigned char count16[2^16];
3 //count16[x] to ilość ustawionych bitów w 16-bitowym słowie x
4 int count32(const int32u u32) {
5     return count16[u32 & max16] + count16[u32 >> 16];
6 }
```

upleft3_from (=0x0E0E0E0E) jest maską z zaznaczonymi polami, z których przy ruchu w górę-lewo przesunięcie wynosi 3 (zawiera wybrane pola z parzystych linii, tj. pola 1..3, 9..11, itd.)

upleft4_from (=0x00F0F0F0) jest maską z zaznaczonymi polami, z których przy ruchu w górę-lewo przesunięcie wynosi 4 (zawiera nieparzyste linie, tj. pola 4..7, 12..15, itd.)

Używane są też tablice (po jednej dla każdego kierunku) następników pól, np.:

```
1 char upleft_nr[32] = {
2     -1, 4,  5,  6,  8,  9,  10, 11,
3     -1, 12, 13, 14, 16, 17, 18, 19,
4     -1, 20, 21, 22, 24, 25, 26, 27,
5     -1, 28, 29, 30, -1, -1, -1, -1
6 };
```

`upleft_nr[x]=y` oznacza iż przy ruchu w górę-lewo następnikiem pola nr. x jest pole nr. y (gdy $0 \leqslant y \leqslant 31$) lub z pola nr. x nie da się iść w górę-lewo (gdy $y = -1$).

6.4.3 Funkcja oceniająca

Funkcja oceniająca jest wywoływana jeszcze częściej niż generator ruchów. Na szczęście, używana w Little Polish reprezentacja planszy, umożliwia efektywne obliczenie wielu jej cech użytecznych przy konstruowaniu funkcji oceniającej. Przykładowo, funkcja z listingu 6.1 umożliwia bardzo szybką odpowiedź na pytanie (na podstawie maski, na której zaznaczone są pola posiadające pewną cechę): ile pól posiada daną cechę?

Wydruk 6.2 przedstawia funkcję oceniającą (aproksymującą wartość wypłaty[7] dla białych) wykorzystywaną w Little Polish. Liczy ona ich przewagę materialną, tj. ilości pionów (z wagą 1) i damek (z wagą 3).

Główną zaletą zastosowania tak prostej funkcji oceniającej jest możliwość szybkiego obliczenia jej wartości. To zaś daje nadzieję na zajrzenie, w rozsądnym czasie, dalej, w głąb grafu gry, i tym samym zwiększa szansę na znalezienie ruchu, po którym (w przyszłości) odniesiemy wyraźną korzyść (w tym przypadku materialną). Jeśli

[7] przyjęto wartości wypłat: $+40$ (za wygraną), 0 (za remis), -40 (za przegraną)

Listing 6.2: funkcja oceniająca programu Little Polish

```
1 //Zwraca ocenę (liczbę całkowitą z zakresu [-36, +36]) stanu [white, black, kings]
2 eval_value_t eval(const int32u white, const int32u black,
3                    const int32u kings) {
4     return count32(white) - count32(black) +
5     ((count32(white & kings) - count32(black & kings)) << 1);
6 }
```

jednak, nie „sięgniemy" aż tak daleko, to w efekcie wszystkie ruchy (poza „oczywistymi", błędnymi posunięciami, jak np. podstawienie pod bicie bez możliwości odbicia) zostaną ocenione jednakowo. To zaś sprawi, iż program będzie wykonywał posunięcia pozornie poprawne, jednak niedające mu przewagi w ustawieniu kamieni na planszy.

6.5 Wyznaczanie (sub)optymalnego ruchu

6.5.1 Ocena złożoności warcabów brazylijskich

Średni czynnik rozgałęzienia dla małych warcabów polskich wynosi około $5,345$.

Liczbę tą obliczono na podstawie 10^8 losowo rozegranych[8] partii (w sumie prawie $5*10^9$ ruchów), poprzez uśrednienie ilości możliwych posunięć w kolejno uzyskiwanych pozycjach.

Zależność średniego czynnika rozgałęzienia od fazy gry (konkretniej, ilości wykonanych w sumie przez obie strony ruchów) przedstawia wykres 6.2. Należy jednak pamiętać, iż wielkość rozgałęzienia, szczególnie w dalszych fazach gry, jest mocno zależna od jej przebiegu. W szczególności, pojawienie się na warcabnicy damek, może znacznie zwiększyć ilość możliwych do wykonania ruchów.

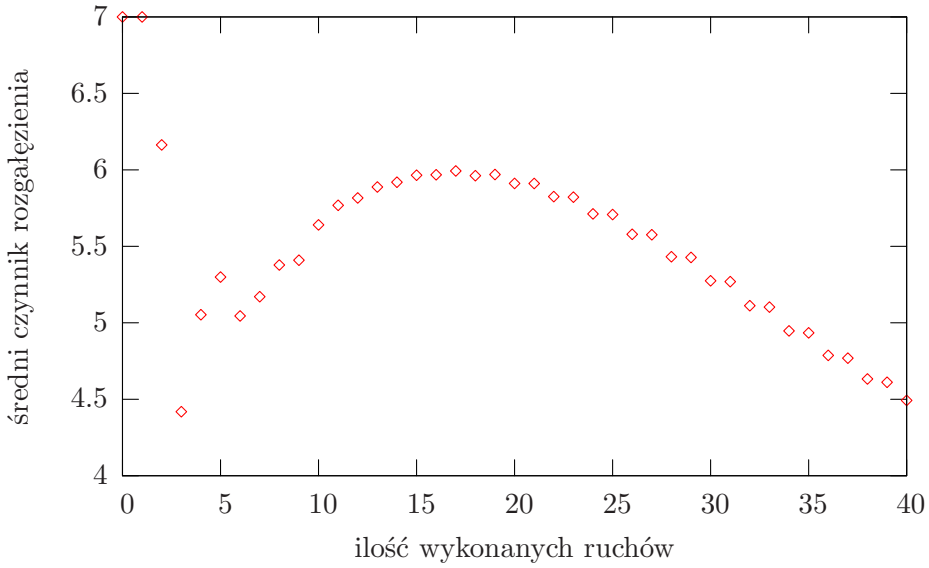
W rozegranych partiach, ponad $1/4$ ruchów była wymuszona (przymusowe bicia, itp.). Jeśli gracz miał jednak do podjęcia jakąś decyzję, to najczęściej wybierał spośród 7 posunięć. Szczegóły przedstawia wykres 6.3.

6.5.2 Algorytmy zastosowane w Little Polish

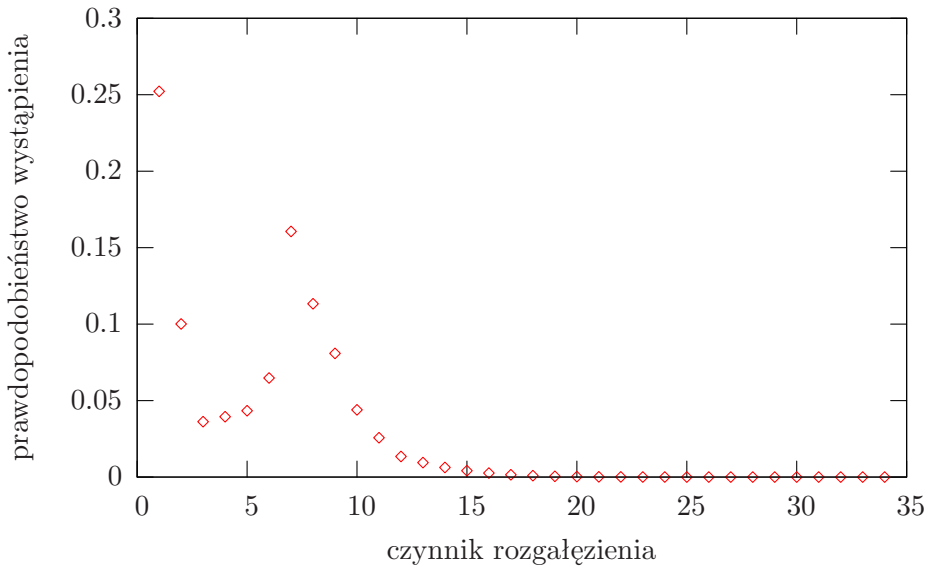
Pierwsze wersje programu korzystały z α-β w klasycznej postaci, bez żadnych dodatkowych rozszerzeń. Głębokość poszukiwań była stała, równa 13. Program nie popełniał bardzo wyraźnych błędów i na ogół grał dość szybko (około 1-2sek/ruch). Jednak długie „namysły" w jakie popadał, gdy na planszy pojawiły się damki, odbierały przyjemność grania przeciwko niemu. Dlatego, pierwszym jego rozszerzeniem był algorytm iteracyjnego pogłębiania. Teraz każdy ruch wykonywany był po mniej więcej stałym czasie, jednak niektóre sytuacje badane były zbyt mało dokładnie. Dlatego

[8] każdy z graczy, w każdej sytuacji, z równym prawdopodobieństwem wykonywał jeden z dozwolonych ruchów

Rysunek 6.2: Zależności czynnika rozgałęzienia od fazy gry (ilości wykonanych w sumie przez obie strony ruchów) w warcabach klasycznych



Rysunek 6.3: Prawdopodobieństwo wystąpienia sytuacji o danym stopniu rozgałęzienia (w warcabach klasycznych)



program wzbogacono o: Quiescence Search (wykonywanie ewentualnych bić przed statyczną oceną), algorytm aspirującego okna, tablicę transpozycji, PVS (opartym na Fail Soft α-β), oraz bazy końcówek i debiutów (którą, za zgodą J. B. Alemanni, skopiowano z jego programu Winbraz 3d).

Szczególnie cenne okazały się: tablica transpozycji, która pozwoliła, bez dodatkowego nakładu czasowego, nawet na dwa razy głębsze zbadanie niektórych pozycji oraz baza końcówek, umożliwiająca bezbłędną rozgrywkę w końcowej fazie partii.

W dalszej części tego rozdziału, podane są szczegóły implementacji poszczególnych rozszerzeń wraz z oceną ich efektywności w kontekście rozważanej gry.

6.5.3 Algorytmy przeszukiwania najwyższego poziomu

Na najwyższym poziomie drzewa poszukiwań zastosowano (przedstawione na wydruku 6.3) algorytmy iteracyjnego pogłębiania (por. roz. 3.6) i aspirującego okna (por. roz. 3.7). Oba w nieco zmodyfikowanej (bardziej „skrupulatnej") postaci.

Po pierwsze, gdy okno poszukiwań okaże się za małe, gdyż nie będzie się w nim mieściła ocena i-tego z kolei dziecka korzenia, może zostać odpowiednio powiększone bez konieczności ponownego przejrzenia poprzednich dzieci (do $i-1$-go włącznie).

Po drugie, przeszukiwanie na głębokość d, może zostać przerwane po zbadaniu pewnej ilości (niekoniecznie wszystkich) dzieci korzenia, gdy nagle skończy się czas. A ponieważ są one przeglądane od najlepszego (wg. ocen z przeszukiwania na głębokość $d-1$), to najlepiej oceniony z dotychczas zbadanych na głębokość d stanów, może zostać zwrócony jako wynik.

6.5.4 Tablica transpozycji

Tablica przejść w Little Polish służy do wykrywania powtórek sytuacji i szybkiej oceny stanu wcześniej badanego lub wyznaczeniu jego potencjalnie najlepszego następnika (szczegóły w roz. 4.2).

Została ona zaimplementowana za pomocą tablicy haszującej z adresowaniem otwartym (por. roz. 4.2.3). Ciąg kontrolny jest wyznaczany za pomocą *haszowania dwukrotnego*. Jego i-ty wyraz jest postaci:

$$h_i(k) = (f(k) + id(k)) \mod m$$

gdzie:

$\mathbf{k}$ klucz

$\mathbf{m}$ rozmiar tablicy

$\mathbf{i}$ numer wyrazu w ciągu kontrolnym, $i = 0, 1, \ldots, m-1$

f pomocnicza funkcja haszująca wyznaczająca pierwszą pozycję w ciągu kontrolnym: $h_0(k) = f(k) \mod m$

Listing 6.3: algorytm zastosowany w Little Polish na najwyższym poziomie drzewa poszukiwań

```
1  //Zwraca najlepszy ruch w sytuacji S w ograniczonym przez time_limit czasie
2  Stan bestmove(Stan S, float time_limit) {
3  movetime_reset();    //zaczynamy mierzyć czas
4  vector<Stan> N = nast(S);
5  if (N == ∅) throw Exception("brak_ruchów");
6  TT.clear(); //czyścimy tablicę transpozycji i
7  //w celu wykrycia remisu po napodkaniu S głębiej w drzewie poszukiwań
8  TT.load(S)->set_repeated(true); //dodajemy do niej S
9  Stan result;
10 int values[|N|], α = −∞, β = ∞, d = 1;
11 do {
12     for (int i = 0; i < |N|; i++) {
13         if (i == 0) {    //pierwszy ruch:
14             values[i] = - AlphaBeta(N[i], d, −β, −α);
15             //jeśli nie znaleziono prawdziwego ograniczenia dolnego:
16             if ((α ≠−∞) && (values[i] ⩽α)) {
17                 alpha = −∞; //zdejmuje ograniczenie dolne
18                 values[i] = - AlphaBeta(N[i], d, −β, ∞);
19             }
20         } else {     //pozostałe ruchy (PVS):
21             values[i] = - AlphaBeta(N[i], d, −α−1, −α);
22             if ((values[i] > α) && (values[i] < β))
23                 values[i] = - AlphaBeta(N[i], d, −β, −α);
24         }
25         //jeśli znaleziono lepszy ruch niż zakłada okno:
26         if (values[i] ⩾ β) {
27             β = ∞; //zdejmujemy górne ograniczenie na okno
28             values[i] = - AlphaBeta(N[i], d, −∞, −α);
29         }
30         if (values[i] > α) { //poprawa wyniku
31             α = values[i];
32             result = moves[i];
33         }
34         if ((i ≠ |N|−1) && (movetime() ⩾ time_limit))
35             return result;  //koniec czasu
36     }
37     stableSort(N, values);  //porządkujemy następniki po ich ocenach
38     α = values[0] − δ;  //nowe okno
39     β = values[0] + δ;
40     ++d;     //zwiększamy głębokość poszukiwań
41 } while (movetime() * (1.0 + 2.0 / |N|) < time_limit);
42 return result;
43 }
```

Listing 6.4: 64-bitowa funkcja mieszająca Boba Jenkinsa

```
1  uint64_t mix64(uint64_t a, uint64_t b, uint64_t c) {
2    a=a-b; a=a-c; a=a^(c>>43);
3    b=b-c; b=b-a; b=b^(a<<9);
4    c=c-a; c=c-b; c=c^(b>>8);
5    a=a-b; a=a-c; a=a^(c>>38);
6    b=b-c; b=b-a; b=b^(a<<23);
7    c=c-a; c=c-b; c=c^(b>>5);
8    a=a-b; a=a-c; a=a^(c>>35);
9    b=b-c; b=b-a; b=b^(a<<49);
10   c=c-a; c=c-b; c=c^(b>>11);
11   a=a-b; a=a-c; a=a^(c>>12);
12   b=b-c; b=b-a; b=b^(a<<18);
13   c=c-a; c=c-b; c=c^(b>>22);
14   return c;
15 }
```

d pomocnicza funkcja haszująca, mówiąca o oddaleniu kolejnych wyrazów ciągu:

$$h_j(k) = (h_{j-1}(k) + d(k)) \mod m \text{ dla każdego } j = 1, 2, \ldots, m-1$$

Aby ciąg $h_0(k), h_1(k), \ldots, h_{m-1}(k)$, stanowił permutację ciągu $0, 1, \ldots, m-1$ wszystkich indeksów tablicy, należy zapewnić, że wartość $d(k)$ jest względnie pierwsza z rozmiarem tablicy m. Jednym ze sposobów, by to uczynić (wykorzystanym w Little Polish), jest przyjęcie pewnej potęgi dwójki jako m przy równoczesnym zapewnieniu, że d daje tylko wartości nieparzyste.

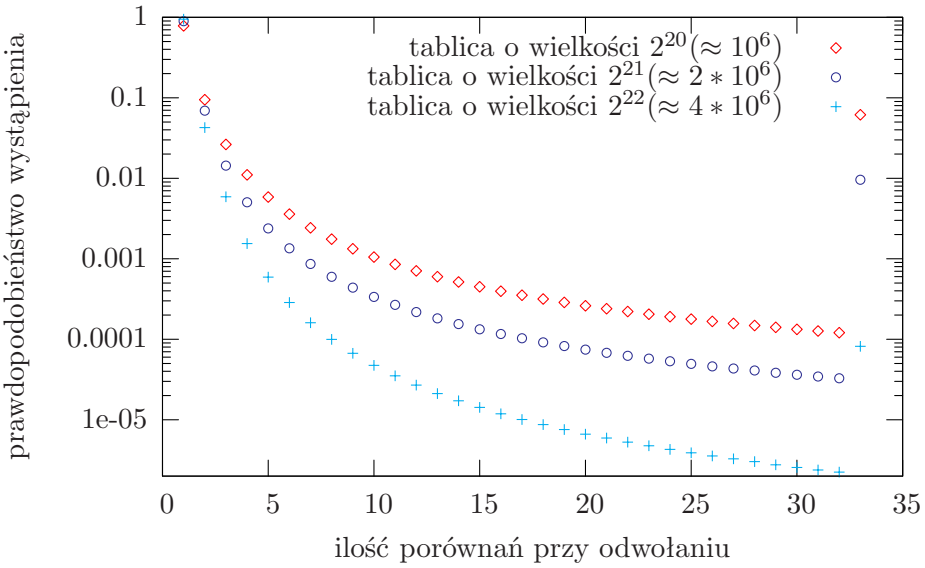
Na podstawie stanu gry (trzech 32-bitowych masek traktowanych jako liczby całkowite bez znaku) tworzony jest 64-bitowy klucz k. Używana jest do tego funkcja skrótu przedstawiona na wydruku 6.4. Jego 32 najstarsze bity wyznaczają $f(k)$ i $d(k)$, odpowiednio: `k >> (64 - p)` i `((k >> 32) << 1) + 1`. Młodsze 32 bity zapisywane są w tablicy (w celu uniknięcia pomyłek).

W programie, wykorzystywane są tylko 33 początkowe wyrazy ciągu kontrolnego, co ogranicza maksymalną ilość porównań kluczy wykonanych przy odszukiwaniu wpisu. Implikuje to niestety, iż niektóre z przeszukanych stanów nie są dodawane do tablicy[9].

W zależności od wielkości tablicy, średnia ilość porównań kluczy wykonana przy odwołaniu do niej, wynosi około: $1,67$ (przy 2^{20} miejsc w tablicy), $1,28$ (przy 2^{21}), $1,07$ (przy 2^{22}). Szczegóły, przedstawione są na wykresach 6.4, 6.5 i 6.6. Statystyczne dane potrzebne do ich sporządzenia, pozyskano przy okazji zbadania algorytmem PVS

[9] stan o kluczu k nie zostanie dodany do tablicy, gdy miejsca o indeksach $h_0(k), h_1(k), \ldots, h_{32}(k)$ zajmują wpisy o innych kluczach

Rysunek 6.4: Szacowane prawdopodobieństwo wykonania danej ilości porównań kluczy przy odwołaniu do intensywnie wykorzystywanej tablicy



(używającym tablicy transpozycji) 35000 wylosowanych[10] stanów gier. Każdy ze stanów, eksploatowany był na kolejne głębokości: $3, 5, 7, \ldots, 15$, bez czyszczenia tablicy między pogłębieniami. Wykorzystanie jej było więc intensywne, gdyż głębokość równa 15 jest dość duża. Podczas eksperymentów, tylko dla najmniejszej z testowanych tablic (2^{20} miejsc) zdarzyły się[11] błędy odczytania nieprawidłowego rekordu wynikające z użycia krótkich (32 bitowych) fragmentów kluczy jako identyfikatorów zapisanych stanów.

Każdy wpis w użytej w Little Polish tablicy przejść zawiera:

klucz (precyzyjniej jego 32-bitowy fragment) identyfikujący zapisany stan S

numer najlepszego znalezionego następnika (8 bitów) w wygenerowanym wektorze ruchów

wartość (8 bitów) stanu gry S którego dotyczy wpis

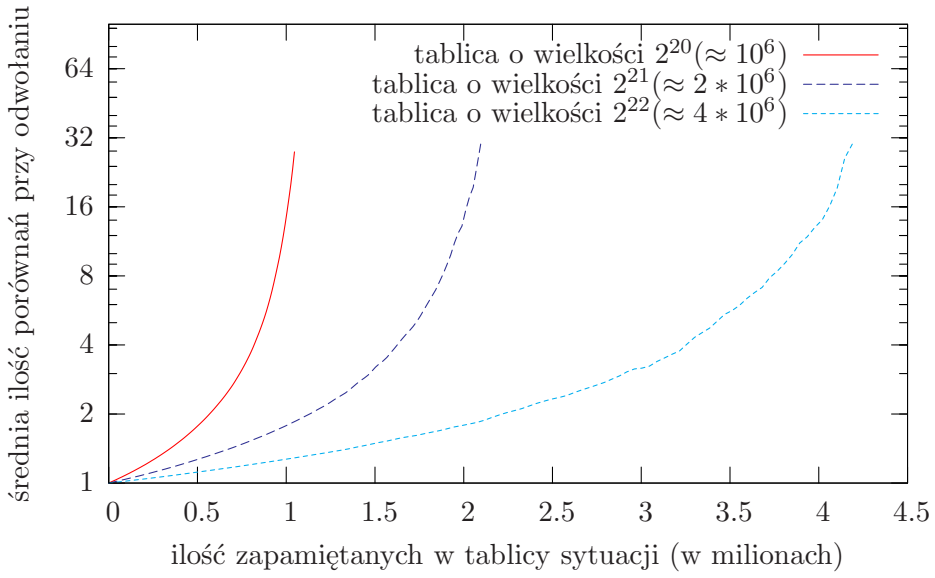
głębokość na jaką badany był stan S

flagi informujące, czy zapisana wartość jest dokładna, czy też jest górną lub dolną granicą (wynikłą z obcięć) tej dokładnej oraz czy dany węzeł drzewa poszukiwać leży na ścieżce od korzenia do węzła aktualnie sprawdzanego

[10] uzyskanych ze stanu początkowego, poprzez wykonanie od 6 do 40 losowych ruchów (po 1000 sytuacji dla każdej liczby posunięć)

[11] precyzyjniej, było ich 11 (przy ponad 10^{10} odwołaniach do tablicy)

Rysunek 6.5: Zależność średniej ilość porównań kluczy (wykonanych przy pojedynczym odwołaniu) od wypełnienia tablicy



Rysunek 6.6: Porównanie uzyskanej zależności średniej ilość porównań kluczy od wypełnienia tablicy, do szacowanej przez twierdzenie 3 o efektywności haszowania

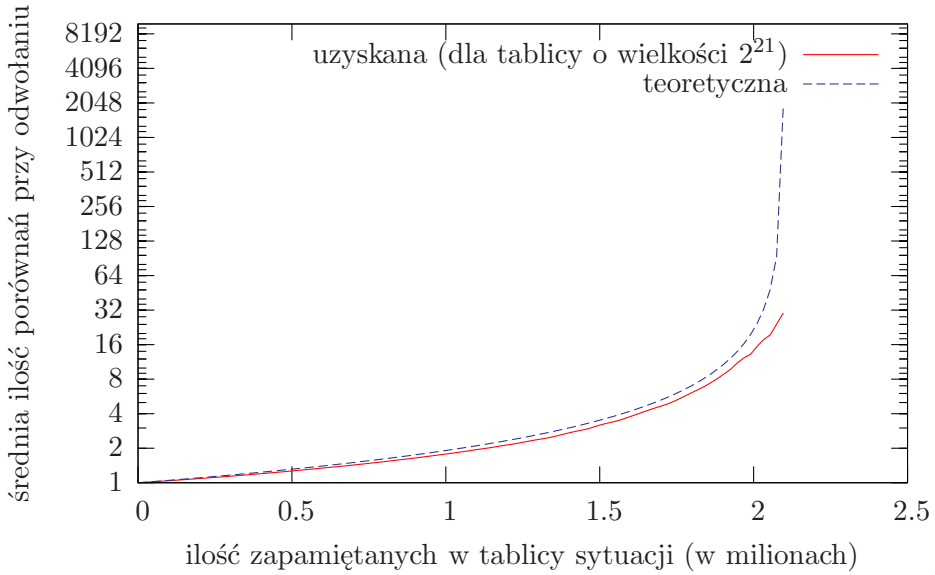


Tabela 6.1: Wielkość bazy końcówek w Little Polish

Numer części (ilość warcab) i	Ilość wpisów $2^{2i}\binom{32}{i}$	Sumaryczna wielkość $\sum_{n=1}^{i} 2^{2n}\binom{32}{n}$
1	128	128
2	7 936	8 064
3	317 440	325 504
4	9 205 760	9 531 264
5	206 209 024	215 740 288
6	3 711 762 432	3 927 502 720
7	55 146 184 704	59 073 687 424

Rozmiar tej struktury, na większości architektur, nie powinien przekroczyć 8 bajtów[12] (nawet po uwzględnieniu wyrównywania struktur w pamięci), zaś całkowity rozmiar użytej w programie tablicy: $2^{21} * 8b = 16Mb$

6.5.5 Baza końcówek

Little Polish używa bazy końcówek zarówno w trakcie przeszukiwania grafu gry (bardzo szybko i dokładnie oceniając wszystkie stany opisane w bazie) jak i do wykonywania ruchów. Dla wszystkich wygranych i wszystkich przegranych sytuacji opisanych w bazie, zapisany jest numer ruchu, który najszybciej doprowadzi do wygranej lub najbardziej odwlecze w czasie przegraną. Sytuacje oznaczone jako remisowe, przeszukiwane są normalnie, algorytmem minimaksowym (w tym przypadku, wszystkie węzły składające się na drzewo poszukiwań opisane są w bazie końcówek).

Jeśli podczas eksplorowania grafu gry, zostanie napotkany węzeł opisany w bazie jako remisowy, to głębokość jego dalszego (rekurencyjnego) badania jest znacznie zmniejszana. Natychmiastowe zwracanie dla takich węzłów 0, uważam za postępowanie zbyt „radykalne". Przeciwnik może nie wiedzieć przecież, że dany stan jest remisowy, zaś 0 nie wyraża dobrze szans na zwycięstwo (relatywnie do ocen uzyskanych przy użyciu funkcji oceniającej). Zauważyłem, że zmniejszenie głębokości poszukiwań, powoduje przeważnie zmniejszenie bezwzględnej wartości uzyskanej oceny (zawodnik posiadający przewagę, szczególnie materialną, ma zazwyczaj możliwość jej powiększenia w trakcie dalszej rozgrywki).

Cała baza końcówek jest podzielona na s części. Część i-ta ($i \in \{1, 2, \ldots, s\}$) zawiera wpisy dotyczące wszystkich sytuacji, w których na planszy stoi dokładnie i kamieni.

Ilość wpisów w i-tej części wynosi $2^{2i}\binom{32}{i}$, zaś sytuacją przyporządkowane są numery wg. następującego schematu:

$$Nr(S) = 2^{2i}P(S) + 2^{i}K(S) + W(S)$$

[12] albo 16 gdy zamiast 32 bitowego klucza zapiszemy całą sytuację

gdzie:

$0 \leqslant Nr(S) < 2^{2i}\binom{32}{i}$ numer przypisany sytuacji S (o i polach zajętych)

$0 \leqslant P(S) < \binom{32}{i}$ jest numerem zależnym od tego, na których i polach stoją (jakiekolwiek) warcaby w sytuacji S, przy czym $P(S) = \sum_{a_x(S) \geqslant x} \binom{a_x(S)}{x}$, gdzie $a_1(S), a_2(S), \ldots, a_i(S)$ to rosnący ciąg numerów pól[13] zajętych w stanie S (numerowanych zgodnie z rys. 6.1)

$0 \leqslant K(S) < 2^i$ to i-bitowa maska wskazująca na których spośród i zajętych pól stoją damki (na pozostałych zajętych polach stoją zwykłe kamienie)

$0 \leqslant W(S) < 2^i$ to i-bitowa maska wskazująca, na których spośród i zajętych pól stoją białe warcaby (na pozostałych zajętych polach stoją czarne). Jednocześnie zakłada się, że do białych należy ruch w stanie S. Sytuacje, w których ruch mają czarne, przekształca się przed obliczeniem jej numeru (wykorzystując symetrię warcabnicy) do odpowiadającej jej z ruchem należącym do białych.

W zastosowanym schemacie numerowania, występuje pewna (nieduża) nadmiarowość. Przykładowo, niedozwolone sytuacje, w których zwykły kamień stoi na polu promocji (powinien więc być damką), mają swoje numery.

Funkcja Nr jest oczywiście odwracalna. Wartość Nr^{-1} można policzyć wyznaczając wpierw składniki:

$$W(S) = Nr(S) \mod 2^i$$

$$K(S) = \lfloor Nr(S)/2^i \rfloor \mod 2^i$$

$$P(S) = \lfloor Nr(S)/2^{2i} \rfloor$$

a następnie odtwarzając zbiór zajętych pól na podstawie $P(S)$ za pomocą algorytmu przedstawionego na wydruku 6.5. Odtworzenie reszty jest trywialne.

Bazę końcówek zbudowano za pomocą lekko zmodyfikowanego algorytmu opisanego w roz. 4.8. Przy czym budowę podzielono na etapy wynikające ze spostrzeżeń, że:

- ilość bierek, jaką dysponuje każdy z graczy nie może wraz z przebiegiem gry wzrosnąć
- ilość królówek należących do gracza nie może wraz z przebiegiem gry zmaleć, jeśli nie zmieni się ilość należących do niego bierek

Tabela 6.5.5 przedstawia ilości wpisów, które znalazłyby się w bazie zawierającej końcówki gier do określonej ilości kamieni na planszy (przy zastosowaniu wyżej opisanego indeksowania). W Little Polish każdy wpis zajmuje w pamięci 1 bajt (na 2 bitach zapisany jest wypłata, zaś na pozostałych 6, numer najlepszego ruchu). Dodatkowy

[13] i elementowy podzbiór 32 elementowego zbioru numerów pól, takich podzbiorów jest $\binom{32}{i}$

Listing 6.5: funkcja odtwarzająca zbiór na podstawie jego numeru

```
1  //Zwraca 32 bitową maskę z zaznaczonymi i bitami
2  //wyznaczającą i elementowy podzbiór zb. 32 elementowego numer nr.
3  //Algorytm łatwo można uogólnić na zbioru o rozmiarze innym niż 32.
4  uint32_t nr2set(int nr, int i) {
5      uint32_t result = 0;
6      for (int x = 31; i > 0 /*lub x >= 0*/; --x)
7          if (nr >= (x i)) {
8              //el. nr. x należy do zbioru wynikowego
9              result |= 1 << x;
10             nr -= (x i);
11             i--;
12         }
13     return result;
14 }
```

1 bajt na wpis, potrzebny jest podczas konstruowania bazy (na zapamiętanie ilości ruchów do końca gry).

Widać, że baza z końcówkami do 5 bierek na planszy (maksymalna, dopuszczalna w programie) nie jest zbyt duża (około 216MB) i bez problemu można przechować ją całą w pamięci. Dla 6 bierek, przechowanie informacji o samych wypłatach wymagałoby prawie 1GB pamięci, zaś proces budowania byłby naprawdę trudny (użyty algorytm potrzebowałby niespełna 8GB pamięci). Pomocne okazać by się mogły różne metody kompresji danych, jednak ich zastosowanie oznaczałoby spowolnienie dostępu do bazy. Dla końcówek z 7 kamieniami na warcabnicy, sprawa wydaje się beznadziejna.

6.6 Analiza skuteczności poszczególnych metod

W celu sprawdzenia, jak skuteczne są[14] poszczególne algorytmy i heurystyki przebadano przy użyciu różnych z nich, 8750 losowych sytuacji[15]. Badanie odbywało się na głębokość od 5 do 14 włącznie, z dodatkowym wykonywaniem obowiązkowych bić przed dokonaniem statycznej oceny wartości węzła (Quiescence Search)[16].

Każdy ze stanów był eksplorowany algorytmem α-β lub PVS, wzbogaconym pewnym podzbiorem następujących metod (w nawiasach podano dalej używane oznaczenia):

[14] konkretniej, w jakim stopniu redukują drzewo poszukiwań

[15] każdą z nich uzyskano ze stanu początkowego, poprzez wykonanie od 6 do 40, dopuszczalnych, losowo wybranych ruchów (po 250 sytuacji dla każdej liczby posunięć)

[16] algorytm ten został jednak potraktowany jako część funkcji oceniającej i rozwinięte przez niego węzły nie zostały policzone jako część drzewa poszukiwań

Tabela 6.2: Porównanie efektywności algorytmów przeszukiwania

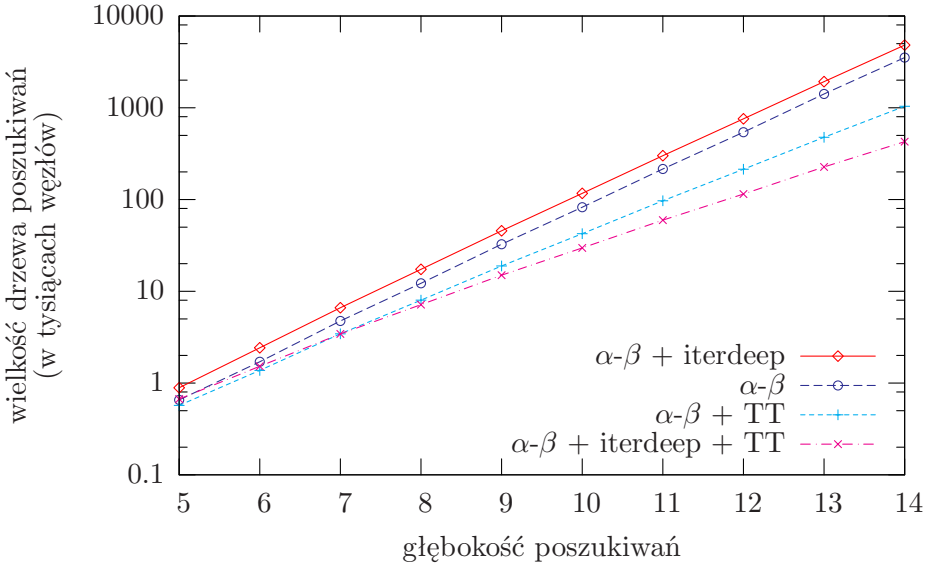
zastosowane algorytmy	średnia ilość węzłów odwiedzonych przy poszukiwaniu na głębokość		
	6	10	14
α-β + iterdeep	2 423	116 790	4 818 576
PVS	1 946	87 873	3 537 031
α-β	1 706	82 428	3 522 847
α-β + TT	1 374	42 448	1 037 559
α-β + iterdeep + TT	1 524	29 708	427 175
PVS + iterdeep + TT	1 512	29 082	417 349
PVS + AspWin + TT	1 510	29 043	416 945
PVS + AspWin + TT + ED5	1 501	28 683	412 156

- algorytm aspirującego okna (AspWin)
- algorytm iteracyjnego pogłębiania (iterdeep)
- tablica transpozycji (TT)
- baza końcówek (EDx - oznacza zastosowanie bazy zawierającej wszystkie końcówki w których na planszy znajduje się do x kamieni włącznie)

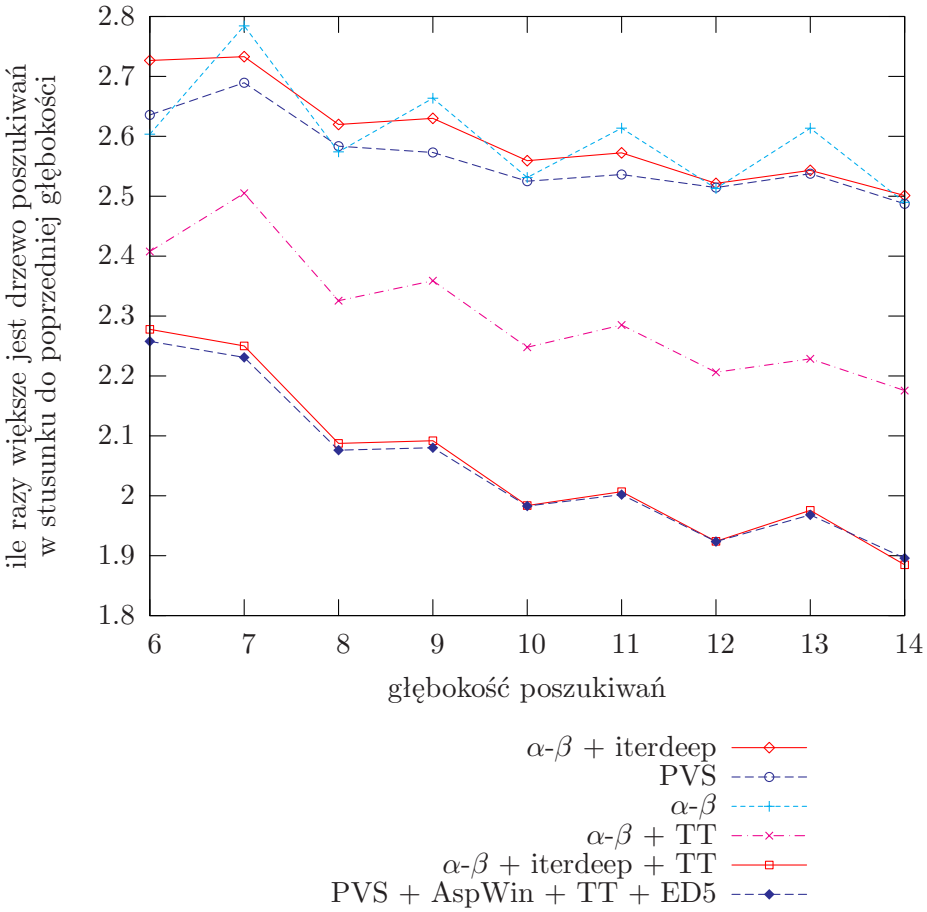
W tabeli 6.6 zestawiono zbadane algorytmy w kolejności od najwolniejszego (konkretniej, od tego który odwiedził najwięcej węzłów przy głębokości poszukiwania wynoszącej 14) do najszybszego. Najistotniejszym ulepszeniem algorytmu α-β okazała się być tablica transpozycji. Szczególnie w połączeniu z iteracyjnym pogłębianiem (razem redukują przeciętne drzewo poszukiwań o wysokości 14 ponad $8,2$ raza), tworzy zestaw, który trudno jest w znaczącym stopniu poprawić. Heurystyki zawężające okno poszukiwań (PVS i algorytm aspirującego okna) nie dają radykalnej poprawy. Najprawdopodobniej spowodowane jest to prostą funkcją oceniającą, która oceniając wiele węzłów tak samo, umożliwia dość szybkie, dostateczne zawężenia okna przez „czyste" α-β (równość ocen wystarcza do wykonania β-odcięcia). Wyraźnej redukcji wielkości drzewa poszukiwań nie daje też baza końcówek. Nic dziwnego, rezultaty jej działania można zaobserwować dopiero w dość dalekich fazach gry (por. wyk. 6.11). Nie należy jednak zapominać o innych jej zaletach (por. roz. 4.8).

Proszę zauważyć, jak cenne staje się realizowane przez tablicę przejść porządkowanie ruchów i odcinanie wcześniej odwiedzonych gałęzi w przypadku stosowania iteracyjnego pogłębiania. Bez niej, algorytm iteracyjnego pogłębiania zamiast znacznie przyspieszyć, spowalnia proces poszukiwania (dobrze jest to zilustrowane na wykresie 6.7).

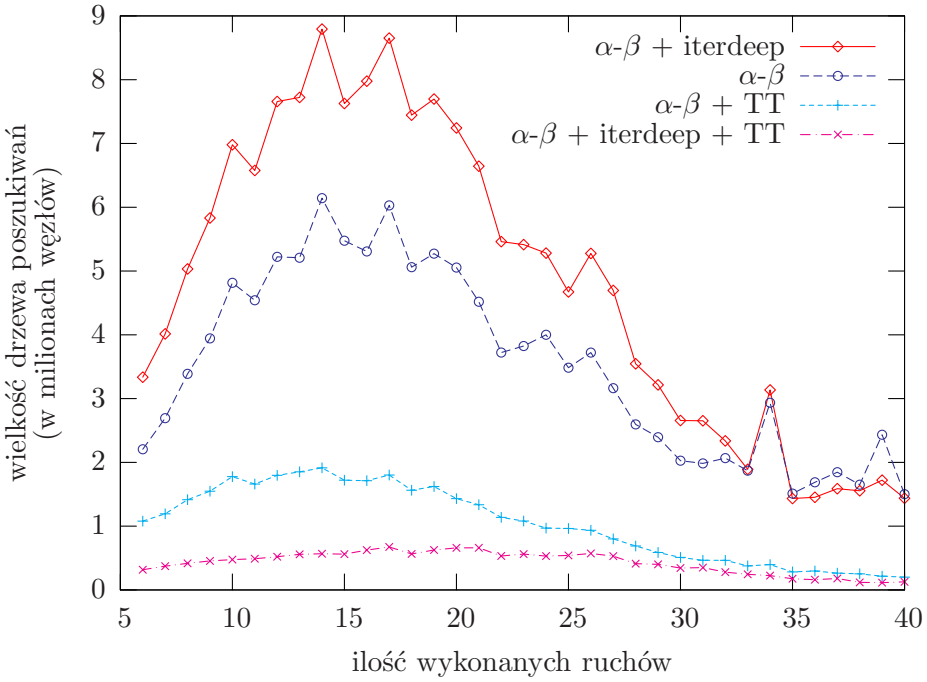
Rysunek 6.7: Wielkość drzewa poszukiwań w zależności od głębokości poszukiwań



Rysunek 6.8: Wielkość drzewa poszukiwań w zależności od głębokości poszukiwań



Rysunek 6.9: Wielkość drzewa poszukiwań w zależności od fazy gry (przy głębokości poszukiwań równej 14)



Wykresy 6.7 i 6.8 ukazują wzrost złożoności algorytmów przy rosnącej głębokości poszukiwań. Bardzo dobrze widać na nich, że mimo zastosowania wymyślnych heurystyk, algorytmy przeszukiwania zachowały wykładniczy[17] charakter.

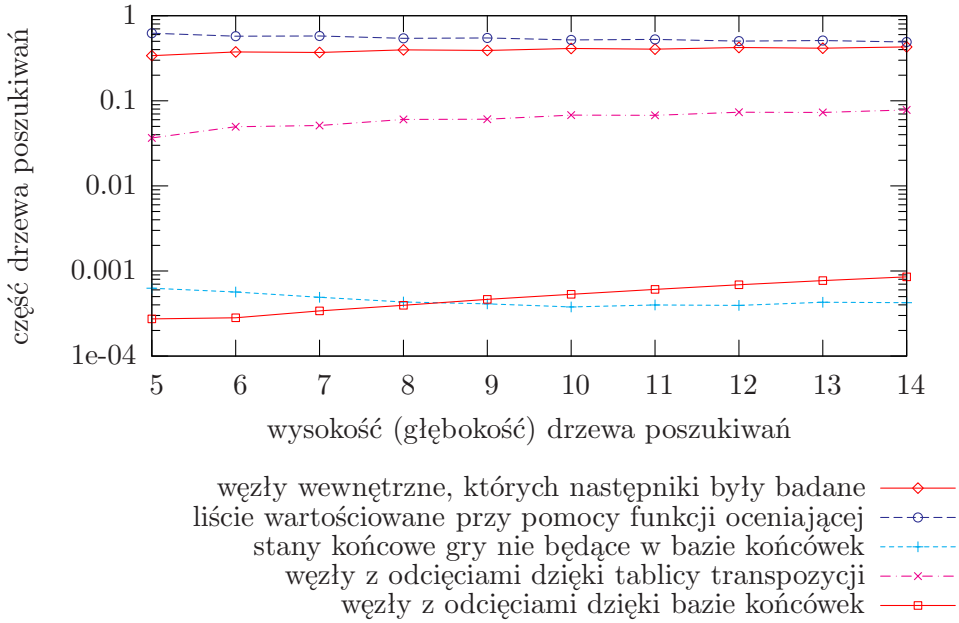
Na widoczny na wykresie 6.8 spadek wartości ilorazów, dla większych głębokości poszukiwań, wpływ mają min. stany bliskie końcowi gry[18], w których wiele gałęzi sięga do węzłów końcowych (o zerowym rozgałęzieniu). Oczywiście, szansa „dotarcia" do takich węzłów rośnie wraz ze wzrostem głębokości poszukiwań. W przypadku stosowania tablicy transpozycji, równocześnie rośnie też szansa na odcięcia wynikłe z wielokrotnego osiągnięcia tych samych węzłów, po różnych sekwencjach ruchów. Algorytmy iteracyjnego pogłębiania i aspirującego okna także większe korzyści dają w dalszych (głębszych) iteracjach.

Wykres 6.9, przedstawia złożoność badanych algorytmów w sytuacjach występujących w grze po różnej ilości ruchów od jej początku. Dla większości metod, ocena stanów pomiędzy 10 a 25 posunięciem okazała się „najtrudniejsza" (wymagająca największych nakładów obliczeniowych). Jest to konsekwencją tego, iż właśnie na tym etapie gry, średni czynnik rozgałęzienia jest największy (por. wyk. 6.2).

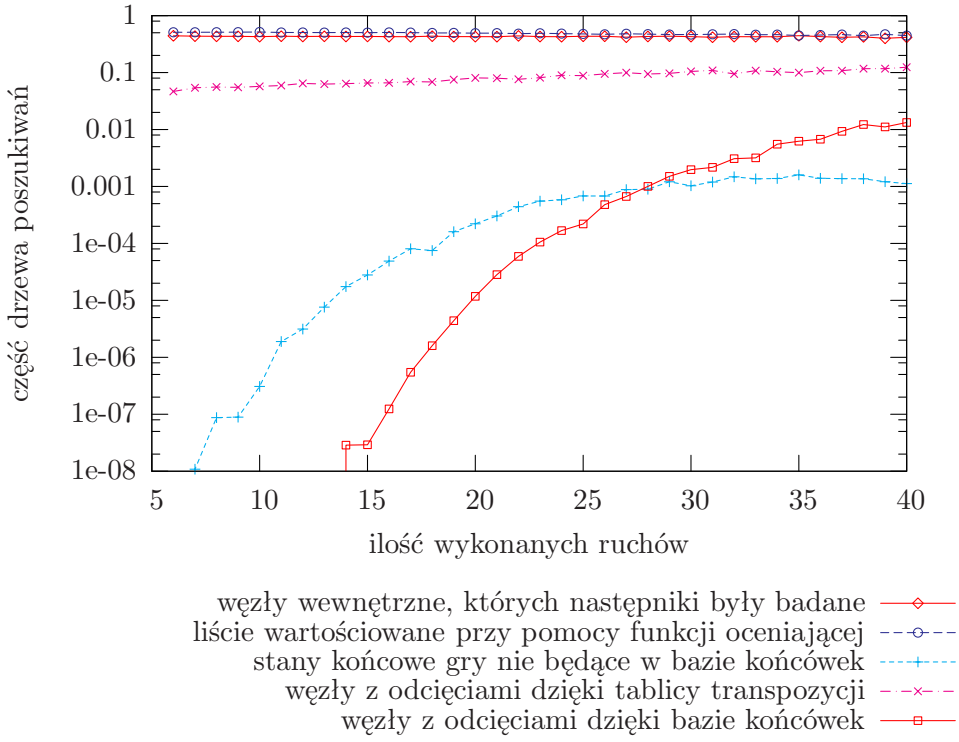
[17] na wykresie 6.7, wartości dla poszczególnych metod, naniesione na logarytmiczną skalę, układają się na prostych

[18] które znalazły się także w zbiorze testowym

Rysunek 6.10: Udział różnego typu węzłów w drzewach poszukiwań różnych wysokości (uzyskanych przy użyciu: PVS + AspWin + TT + ED5)



Rysunek 6.11: Udział różnego typu węzłów w drzewach poszukiwań (uzyskanych przy użyciu: PVS + AspWin + TT + ED5) o wysokości 14 w zależności od fazy gry.



Wykresy 6.10 i 6.11 ukazują udział różnego typu węzłów w drzewach poszukiwań. Zazwyczaj, większość drzewa stanowią liście wartościowane za pomocą funkcji oceniającej. Niemała jego część to węzły wewnętrzne. Odcięcia najczęściej spowodowane są zastosowaniem tablicy transpozycji, zaś ich udział rośnie, zarówno wraz z wysokością drzewa, jak i ilością wykonanych od początku gry ruchów. Dopiero w dalszych fazach gry, osiągnięte stany końcowe oraz takie, w których nastąpiło odcięcie dzięki bazie końcówek mają swój (niewielki) udział w drzewie poszukiwań.

6.7 Little Polish a programy innych autorów

Algorytm α-β i jego odmiany, stanowią obecnie podstawę większości komputerowych realizacji gier logicznych. Popularność zyskują też algorytmy z rodziny MTD (szczególnie MTD(f)). Zasadnicze różnice między poszczególnymi programami leżą jednak głównie w doborze dodatkowych heurystyk oraz w szczegółach implementacji.

Ze sprawdzonego schematów nie wyłamuje się też Little Polich. Oparty jest on o jeden z najczęściej stosowanych algorytmów przeszukiwania drzew: PVS. Dobór rozszerzeń też jest typowy: tablica transpozycji, Quiescence Search, algorytm aspirującego okna i baza debiutów. Cechą wyróżniającą jest całkiem niemała baza końcówek (zawierająca wszystkie końcówki w których na planszy znajduje się nie więcej niż 5 warcabów).

Podczas implementacji duży nacisk położono na wydajność, co objawia się min. częstym stosowaniem tablic LUT (ang. Look-Up Table), arytmetyki wskaźników i operacji bitowych, oraz unikaniem dzielenia i operacji zmiennoprzecinkowych. Sporadycznie, zdarzają się też wstawki asemblerowe.

Brak dostępu do opisu architektury lub kodów źródłowych jakiejkolwiek innej aplikacji grającej w warcaby klasyczne, uniemożliwia dokonanie szczegółowych porównań. Możliwe jest jedynie oszacowanie różnicy w sile gry Little Polish w stosunku do innych programów, w oparciu o wyniki bezpośrednich pojedynków (które przedstawia tabela 6.7).

Najlepszym i najpopularniejszym spośród programów w zestawieniu jest grający na mistrzowskim poziomie Winbraz 3d (autorstwa J. B. Alemanniego). Do niedawna był on aplikacją komercyjną, aktualnie jednak jest darmowy. Darmowe są także dwa polskie programy, napisane przez, odpowiednio Krzysztofa Giaro i Marcina Żółtkowskiego. PLUS800 to komercyjny produkt, którego autorem jest Serge Startsev. Aplikacja ta korzysta z ogromnej bazy zawierającej wszystkie końcówki z co najwyżej 6 bierkami na warcabnicy i ma możliwości samodouczania się (niestety autor nie podał żadnych szczegółów dotyczącuch tego aspektu). Zestawienie dotyczy najnowszych, dostępnych w chwili pisania pracy, wersji programów (wersji demo w przypadku PLUS800). Każdy z nich, co drugą partię grał białymi, co drugą zaś czarnymi.

Tabela 6.3: Wyniki uzyskiwane przez Little Polish w grze przeciwko innym programom.

Testowany program		Little Polish			
nazwa, autor,	Ilość	Wyniki / ilość partii			
konfiguracja	sek/ruch	Wygrane	Remisy	Przegrane	
Winbraz 3d 7.00					
J. B. Alemanni					
poziom: 3min/30ruchów	6	3	2 / 10	7 / 10	1 / 10
przy rozm. TT: 1,5MB[a]					
poziom: 3min/30ruchów	6	6	1 / 10	6 / 10	3 / 10
przy rozm. TT: 15MB					
PLUS 800					
Serge Startsev					
tryb analityczny					
poziom: 1sek/ruch	1[b]	1	2 / 10	7 / 10	1 / 10
poziom: 3sek/ruch	3[c]	3	4 / 10	6 / 10	0 / 10
Gra w warcaby					
Krzysztof Giaro					
poziom: 6 (najwyższy)	3[d]	1	3 / 10	6 / 10	1 / 10
Warcaby					
Marcin Żółtkowski					
poziom: 3 (najwyższy)	?[e]	0,05	10 / 10	0 / 10	0 / 10

[a] rozmiar domyślny
[b] dodatkowo, program „myślał na czasie przeciwnika"
[c] dodatkowo, program „myślał na czasie przeciwnika"
[d] czas zmierzony (przybliżony)
[e] czas bardzo krótki, trudny do zmierzenia

Wyniki uzyskane przez Little Polish wydają się być dość dobre. Wyraźnie pokonał on obu polskich rywali (jednego do zera) oraz komercyjnego PLUS 800. Nawiązywał też walkę z bardzo silnym Winbrazem 3d, od którego okazał się jednak nieco słabszy.

6.8 Możliwe dalsze ulepszenia

Ilość potencjalnych poprawek i heurystyk jaką można wprowadzić do programu grającego w dowolną, niebanalną grę logiczną, wydaje się być nieskończona. Widać to też na przykładzie Little Polish. Mimo ogromnej metamorfozy jaką przeszedł ten program od czasu powstania jego pierwszej wersji, lista wciąż możliwych do ulepszenia elementów jest długa.

Na samym początku tej listy, znajduje się, moim zdaniem, funkcja oceniająca. Szczególnie w początkowych fazach każdej partii, zbyt wiele ruchów ocenianych jest

przez nią tak samo. Jej wyraźne udoskonalenie jest jednak zadaniem niebanalnym i równocześnie doskonałym tematem na kolejne opracowanie.

Do poprawy umiejętności gry prezentowanej przez program, mogłoby się także przyczynić bardziej selektywne przeszukiwanie grafu gry.

Znaczna redukcja wielkości uzyskiwanych drzew poszukiwań wydaje się bardzo trudna. W pewnym, raczej niewielkim stopniu, mogłoby się do tego przyczynić lepsze porządkowanie ruchów np. przy użyciu heurystyki historycznej (por. roz. 4.4) lub heurystyki ruchów morderców (por. roz. 4.5). Heurystyki te, nabrałyby jednak większego znaczenia przy bardziej złożonej funkcji oceniającej.

Rozdział 7

Podsumowanie

7.1 Co udało się zrealizować?

Zgodnie z założeniami, udało się dość dokładnie opisać i zbadać metody, na których oparte jest działanie większości współczesnych programów grających w rozważaną klasę gier. Przedstawiono wszystkie powszechnie stosowane algorytmy przeszukiwania grafów gier (algorytmy minimaksowe) oraz najważniejsze heurystyki przyspieszające ten proces. Zaproponowano też efektywne rozwiązania wielu szczegółowych zagadnień implementacyjnych związanych z tematem, w tym sposobów:

- reprezentacji stanu gry w pamięci komputera (por. roz. 6.4.1)
- generowania ruchów (por. roz. 6.4.2)
- realizacji tablicy transpozycji (por. roz. 4.2.3 i 6.5.4)
- budowania bazy końcówek (por. roz. 4.8) i numerowania sytuacji na jej potrzeby (por. roz. 6.5.5)
- manipulowania oknem poszukiwań algorytmu α-β (por. roz. 6.5.3)

Czytelnik, opierając się na wskazówkach zawartych w tej pracy, nie powinien mieć problemu ze stworzeniem własnego, dobrze grającego komputerowego zawodnika.

Bazując na opisywanych metodach zrealizowano program Little Polish, grający w warcaby brazylijskie. Reprezentowany przez niego poziom okazał się całkiem wysoki względem innych, powszechnie dostępnych aplikacji (szczegóły zawarte są w pkt. 6.7). Potwierdza to efektywność zastosowanych rozwiązań.

Nie tylko osoby zainteresowane bezpośrednio tematem programowania gier logicznych, ale także amatorzy gry w warcaby klasyczne powinni być więc zadowoleni z wykonanej pracy. Wnioskując po rosnącej ilości odwiedzin i pobrań programu Little Polish ze strony autora, tak istotnie jest.

7.2 Co można by jeszcze dodać? Możliwe kierunki dalszych badań

Zarówno rozdział 4 jak i 5, poruszają tematy, w których zrodziło się co najmniej tyle pomysłów, ilu ludzi zajmowało się programowaniem gier logicznych. Nie sposób przeanalizować i opisać je wszystkie, dlatego w pracy zostały przedstawione tylko te, które wydawały się najważniejsze.

Szczególnie interesujący wydaje się temat automatycznego tworzenia funkcji oceniających. Jest on świetnym poligonem doświadczalnym dla różnych rozwiązań pokrewnych mu problemów maszynowego uczenia oraz optymalizacji. Nie dziwi więc ogromna ilość badań wykonywanych w tym zakresie, dotyczących min. stosowania: sztucznych sieci neuronowych (np. [25]), klasycznych metod optymalizacji (np. metody najszybszego spadku [2,19]), algorytmów ewolucyjnych (np. [1]) lub genetycznych (np. [5,17]). Dobrym kierunkiem dalszych badań jest analiza tych rozwiązań, szczególnie pod kątem ich względnej skuteczności w poszczególnych grach (np. poprzez bezpośrednie pojedynki programów z nich korzystających). Można też spróbować znaleźć ewentualne związki pomiędzy efektywnością poszczególnych algorytmów a cechami charakterystycznymi gry lub grupy gier w których zostały użyte. Ich znalezienie dałoby wskazówkę, jak dobrać metodę konstruowania funkcji oceniającej do danej gry.

W dziedzinie programowania gier logicznych, istnieje szereg nieomówionych w tym opracowaniu zagadnień, dotyczących pisania programów używających do obliczeń wielu komputerów lub procesorów równocześnie.[1] Dotyczą one np. równoległego przeszukiwanie grafu gry, czy współdzielonego dostępu do tablicy transpozycji. Analiza zastosowanych w tym obszarze rozwiązań oraz przetestowanie ewentualnych, własnych pomysłów także wydaje się być ciekawym kierunkiem dalszych badań.

Poszerzenie obszaru zainteresowań i przyjrzenie się grom w których występuje czynnik losowy lub brak pełnej informacji także jest interesującym kierunkiem kolejnych analiz. Szczególnie ciekawe są zagadnienia: prób sprowadzania gier bez pełnej informacji do gier z pełną informacją, modelowania występujących w takich grach aspektów psychologicznych czy też sposobów wnioskowania z zachowania przeciwnika na temat stanu w jakim znajduje się gra.

Możliwości dalszego rozwoju programu Little Polish, przedstawiono w roz. 6.8.

[1] np. Deep Blue, pogromca mistrza świata w szachy, Garrego Kasparowa, uruchomiony był na 32-węzłowym klastrze IBM RS/6000 SP, zawierającym po 8 wyspecjalizowanych procesorów szachowych w każdym węźle.

Bibliografia

[1] Borkowski M.: Analiza algorytmów dla gier dwuosobowych. Praca magisterska, Politechnika Warszawska, 2000.

[2] Buro M.: From simple features to sophisticated evaluation functions. 1999.

[3] Chang M.-S.: Building a fast double-dummy bridge solver. Raport instytutowy TR1996-725, New York University, Sier., 1996.

[4] Cormen T. H., Leiserson C. E., Rivest R. L.: *Wprowadzenie do algorytmów*. WNT, Warszawa, 1997, 1998, 2000.

[5] Dąbrowski T., Kwaśnicka H., Piasecki M.: Zastosowanie algorytmu genetycznego do konstrukcji funkcji oceniającej w grze othello.

[6] Fierz M.: Strategy game programming. WWW.

[7] http://www.gametheory.net/. WWW.

[8] Ginsberg M. L.: How computers will play bridge.

[9] Ginsberg M. L.: Partition search. *AAAI/IAAI, Vol. 1*, strony 228–233, 1996.

[10] Ginsberg M. L.: GIB: Steps toward an expert-level bridge-playing program. Thomas D., redaktor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99-Vol1)*, strony 584–593, S.F., Lip. 31–Sier. 6 1999. Morgan Kaufmann Publishers.

[11] Ginsberg M. L.: GIB: Imperfect information in a computationally challenging game. *J. Artif. Intell. Res. (JAIR)*, 14:303–358, 2001.

[12] Kozłowska-Pięcek J.: Porównanie różnych strategii heurystycznych dla gier. Praca magisterska, Politechnika Warszawska, 1999.

[13] Kujawski A.: Programowanie gry w szachy. Praca magisterska, Uniwersytet Warszawski, 1994.

[14] Kwaśnicka H., Spirydowicz A.: *Uczący się komputer. Programowanie gier logicznych.* Oficyna Wydawnicza Politechniki Wrocławskiej, Wrocław, 2004.

[15] Marsland T.: Game tree searching and prunning.

[16] Michalski A.: Współczesne techniki przeszukiwania grafów gier dwuosobowych. Slajdy do wykładu, 2004.

[17] Michniewski T.: Samouczenie programów szachowych. Praca magisterska, Uniwersytet Warszawski, 1995.

[18] Moreland B.: http://www.seanet.com/ brucemo/topics/topics.htm. WWW.

[19] Paterek A.: Modelowanie funkcji oceniającej w szachach. Praca magisterska, Uniwersytet Warszawski, 2004.

[20] Plaat A.: Mtd(f), a minimax algorithm faster than negascout (http://home.tiscali.nl/askeplaat/mtdf.html/). WWW.

[21] Plaat A.: *Research, Re: Search & RE-SEARCH*. Praca doktorska, Erasmus Univ., Rotterdam, 1996.

[22] Plaat A., Schaeffer J., Pijls W., de Bruin A.: A new paradigm for minimax search. Research Note EUR-CS-95-03, Erasmus University Rotterdam, Rotterdam, Netherlands, 1995.

[23] Strona internetowa polskiego towarzystwa warcabowego (http://www.ptw.weh.com.pl/). WWW.

[24] Samuel A. L.: Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 44(1):206–227, 2000.

[25] Tesauro G.: Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, Mar. 1995.

[26] Kervinck M. N. J. van: The design and implementation of the rookie 2.0 chess playing program. Praca magisterska, Technische Universiteit Eindhoven, Eindhoven, Sier. 2002.

Dodatki

Dodatek A

Międzynarodowe zasady gry w warcaby

Spisane w oparciu o Kodeks Warcabowy Polskiego Towarzystwa Warcabowego, międzynarodowe zasady gry w warcaby w 20 punktach, przytaczam za [23]:

1. Gra w warcaby toczy się pomiędzy dwoma osobami (przeciwnikami) nazywanymi warcabistami (warcabistkami w wypadku kobiet), które wykonują ruchy bierkami (warcabami) na kwadratowej planszy zwanej warcabnicą.

2. Plansza kwadratowa (warcabnica) służąca do gry jest podzielona w warcabach międzynarodowych na 100 równych pól (kwadratów), a warcabach klasycznych na 64 pola na przemian jasnych i ciemnych.

3. Gra w warcaby odbywa się na ciemnych polach będących jednocześnie 50 (warcaby międzynarodowe) lub 32 (warcaby klasyczne) aktywnymi polami.

4. Warcabnica musi być ustawiona między graczami w ten sposób, że każde pierwsze lewe pole dla każdego z graczy jest polem ciemnym.

5. Warcaby międzynarodowe rozgrywane są 20 warcabami (kamieniami) jasnymi i 20 warcabami (kamieniami) ciemnymi, zaś klasyczne 12 warcabami (kamieniami) jasnymi i 12 warcabami (kamieniami) ciemnymi. Przed rozpoczęciem partii kamienie są rozkładane na najbliższych, każdemu z graczy, rzędach (liniach) na ciemnych polach tak by dwa środkowe rzędy pozostały puste.

6. W czasie gry w warcaby rozróżniamy następujące warcaby:

   - warcaby zwykłe - kamienie
   - warcaby uprzywilejowane - damki

Kamienie i damki różnią się sposobem poruszania i bicia. Przesunięcie warcaby z jednego pola na inne nazywamy ruchem.

7. Gracze wykonują kolejno po jednym ruchu, na przemian, własnymi warcabami. Pierwszy ruch wykonuje zawsze gracz grający białymi warcabami. Wykonanie ruchu jest obowiązkowe - gracz nie może z przypadającego nań ruchu się zrzec.

8. Kamień może poruszać się po przekątnej tylko do przodu na wolne pole następnej linii.

9. W sytuacji gdy kamień dostanie się na pole przemiany (ostatnia przeciwległa linia w stosunku do linii wyjściowych - tzw. podstawa przeciwnika), to staje się damką. W celu odróżnienia damki od kamienia na kamień ulegający przemianie nakłada się warcabę tego samego koloru czyli koronę damki.

10. Nowo utworzona damka może wykonać ruch w następnym posunięciu po wykonaniu ruchu przez przeciwnika.

11. Damka porusza się po przekątnych (ciemnych polach) we wszystkich kierunkach (do przodu i do tyłu) na dowolne wolne pole.

12. Jeżeli kamień znajduje się w sąsiedztwie po przekątnej warcaby przeciwnika, za którą jest wolne pole, to może on przeskoczyć przez tę warcabę i zająć to wolne pole. Warcaba przeciwnika po wykonaniu ruchu jest usuwana z warcabnicy. Całość operacji nazywa się zbiciem. Zdjętą z warcabnicy warcabę uważa się za zbitą. Zbicie może być wykonane do przodu lub do tyłu.

13. Jeżeli damka po przekątnej znajduje się bezpośrednio lub w dalszym sąsiedztwie warcaby przeciwnika, za którą jest jedno lub więcej wolnych pól, to może przeskoczyć przez tę warcabę i zająć dowolne wolne pole na tej przekątnej. Warcaba przeskoczona jest zdejmowana z warcabnicy. Całość operacji nazywana jest zbiciem przez damkę. Zbicie może być wykonane do przodu lub do tyłu.

14. Zarówno kamień jak i damka mogą wykonywać w jednym ruchu zbić wielokrotnych o ile zachowane są warunki wymienione w pkt. 12 i 13, przy czym w czasie wielokrotnego bicia wolno przechodzić wielokrotnie przez to samo pole, ale nie przez tę samą warcabę przeciwnika, zaś przeskoki przez własne warcaby są niedozwolone. Zbite warcaby przeciwnika wolno usunąć z warcabnicy dopiero po zakończeniu bicia (zasada tureckiego bicia).

15. Bicie jest obowiązkowe i ma pierwszeństwo przed wykonaniem innego ruchu (zasada przymusu bicia).

16. W wypadku gdy istnieje wybór pomiędzy zbiciem różnych ilości warcab przeciwnika, to obowiązkowym jest bicie większej ilości warcab (zasada bicia większości). Przy czym damka nie ma przywileju pierwszeństwa wykonania bicia przed biciem kamieniem. Nie ma również znaczenia jakie warcaby są bite.

17. Jeżeli gracz ma dwie lub więcej możliwości bicia takiej samej ilości warcab przeciwnika, to może wybrać jedną z nich.

18. Jeżeli kamień (zwykła warcaba) przechodzi w czasie bicia przez jedno (z czterech) pól przemiany (w podstawie przeciwnika) i może kontynuować bicie to nie ulega przemianie w damkę i nadal pozostaje kamieniem (zwykłą warcabą).

19. Partia warcabowa może się zakończyć wygraną jednego z graczy lub remisem (wynik nierozstrzygnięty).

20. Gracza uznaje się za zwycięzcę w partii jeśli jego przeciwnik nie może wykonać ruchu z powodu:

    - braku warcab (wszystkie zostały zbite),
    - braku możliwości wykonania ruchu ponieważ jego warcaby zostały zablokowane (nie mają wolnych pól do ruchu),

    w pozostałych przypadkach partię uznaje się za zakończoną wynikiem remisowym.

Spis symboli i skrótów

Symbol	Znaczenie	Definicja
$\mathbb{S}$	przestrzeń stanowa gry	strona 4
$\mathbb{B}$	średni czynniki rozgałęzienia drzewa gry	strona 9

Spis rysunków

Spis listingów

Skorowidz