Techniki sztucznej inteligencji w programach grających

Jakub Pawlewicz

27 luty 2010

1 Wprowadzenie

Sztuczna inteligencja w grach jest bardzo atrakcyjną dziedziną badań, gdyż wymyślone metody można łatwo sprawdzać w praktyce, obserwując siłę programów grających w popularne gry, które niejednokrotnie wprowadzają w zdumienie autora programu. Zostało wymyślonych wiele technik ułatwiających pisanie programów pozwalających komputerom „myśleć".

W tym artykule przedstawiamy kilka takich technik. Dotyczą one głównie gier dwuosobowych z pełną informacją i o sumie zerowej. Pełna informacja oznacza, że pełny stan gry jest znany wszystkim graczom w dowolnym momencie rozgrywki (co nie ma miejsca np. w brydżu). Gra o sumie zerowej oznacza mniej więcej tyle, że wygrana jednego gracza to przegrana drugiego i na odwrót. Dodatkowo zakłada się jeszcze, że gracze wykonują ruchy na przemian. Do tej klasy wpada wiele znanych gier dwuosobowych, takich jak szachy, warcaby czy kółko i krzyżyk.

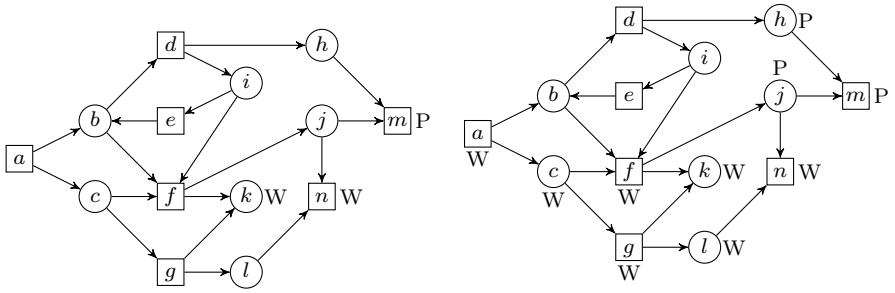
Mamy nadzieję, że ten przegląd technik ułatwi Czytelnikowi napisanie własnego „inteligentnego" programu grającego w jego ulubioną grę. Dobór techniki zależy od rodzaju gry. Można je łączyć lub sięgnąć po najróżniejsze udoskonalenia, a może wymyślić na ich podstawie własną technikę?

2 Strategia optymalna

O ile gra jest wystarczająco prosta, możemy próbować znaleźć dla niej strategię optymalną. O grze możemy myśleć jak o grafie skierowanym, w którym wierzchołki są pewnymi stanami w grze, a krawędzie dostępnymi ruchami. W grze dwuosobowej są dwa rodzaje wierzchołków: takie, w których my wykonujemy ruch, i takie, w których ruch wykonuje przeciwnik. Będziemy je oznaczać odpowiednio przez □ i ○.

Wierzchołki, z których nie wychodzą żadne krawędzie, reprezentują stany końcowe. W tych wierzchołkach znana jest wartość gry. Może to być wygrana, przegrana, a w niektórych grach remis.

Rozważmy przykładowy graf gry przedstawiony na rysunku 1(a). Stany oznaczone jako k, m i n są końcowe. P i W oznacza odpowiednio przegraną i wygraną z naszego punktu widzenia w danym stanie. Interesują nas wartości w pozostałych stanach. Do ich wyznaczania stosuje się technikę *tablicowania od końca*, którą prezentuje algorytm 1. Dla naszego przykładowego grafu, zakła-



(a) Wartości w stanach końcowych.

(b) Wyznaczone wartości.

Rysunek 1: Przykładowy graf gry.

Algorytm 1 Wyznaczanie wartości dla wszystkich stanów w grze.

```
Utwórz kolejkę q zawierającą wszystkie stany końcowe
while q jest niepusta do
    v ← wyjmij element z początku kolejki q
    for all w takich, że z w można przejść do v do
        g ← gracz wykonujący ruch w stanie w
        if v jest wygrany z punktu widzenia gracza g then
            Oznacz w jako wygrany z punktu widzenia gracza g
            Dodaj w na koniec kolejki q
        else if wszystkie ruchy z w prowadzą do przegranej then
            Oznacz w jako przegrany z punktu widzenia gracza g
            Dodaj w na koniec kolejki q
Wszystkie nieoznaczone stany oznacz jako remisowe
```

dając, że na początku w kolejce q umieszczamy n, m, k w takiej kolejności, w algorytmie obliczymy wartości kolejno w stanach l, h, j, f, g, c, a. Rysunek 1(b) zawiera graf z uzupełnionymi wartościami P i W (z naszego punktu widzenia). Dla przykładu, stan a, w którym my wykonujemy ruch, jest wygrywający, a jedyny ruch prowadzący do wygranej to ruch do stanu c. Natomiast stany b, d, i, e nieoznaczone w pętli while algorytmu uznajemy za remisowe, gdyż żadnemu z graczy nie opłaca się wyjść z tego cyklu. Faktycznie, każdy ruch wychodzący poza cykl prowadzi do przegranej z punktu widzenia gracza wykonującego ruch.

Ta prosta, siłowa metoda rozwiązywania gier została użyta do utworzenia tablicy końcówek sześciopionkowych w szachach i dziesięciopionkowych w warcabach. Całkowicie rozwiązano również takie gry jak *młynek* czy *awari*. Jako ćwiczenie można spróbować rozwiązać grę karcianą *pan* w wersji dla dwóch osób. Oczywiście, za pomocą opisanej metody można poznać strategię optymalną jedynie dla gier o małej liczbie stanów.

3 Algorytmy przeszukiwania drzewa gry

W przypadku, gdy nie mamy możliwości stablicować wartości dla wszystkich stanów, potrzebujemy technik, które będą „myślały". Umiejętność grania w gry składa się z dwóch aspektów: *taktyki* i *strategii*.

Taktyka dotyczy celów krótkoterminowych w grze. Jest to umiejętność wynajdywania takich kombinacji na kilka ruchów do przodu, które dają szybki zysk i widoczną przewagę według jakichś ustalonych kryteriów. Na ogół są to serie ruchów z groźbami, uniemożliwiające przeciwnikowi skuteczną odpowiedź. W szachach może to być seria szachów, po której bijemy cenną figurę. Innymi widowiskowymi zagraniami taktycznymi w szachach są poświęcenia. Czasami zdarza się, że gracz poświęca królową, aby w kilku posunięciach zbić kilka innych figur przeciwnikowi, zyskując przewagę materialną.

Strategia z kolei oznacza takie planowanie rozgrywki, że zysk uwidacznia się dopiero w dalszej perspektywie. Myślenie strategiczne wiąże się z głęboką wiedzą o grze. Przykładowo, w szachach w początkowej fazie warto zadbać między innymi o strukturę pionów. Bardzo często dopiero w samych końcówkach dobrze dobrane ustawienie pionów daje przewagę nad przeciwnikiem.

Komputer trudno nauczyć myślenia strategicznego bez wprowadzania dużej ilości heurystyk i wiedzy specyficznej dla danej gry. Niemniej istnieją uniwersalne techniki, które sprawdzają się w niektórych grach, np. bazujące na metodzie Monte Carlo; powiemy o nich trochę w punkcie 4. Natomiast programy w większości gier przeważają nad ludźmi w elementach taktycznych, gdyż stosowane są w nich algorytmy przeszukiwania drzewa gry.

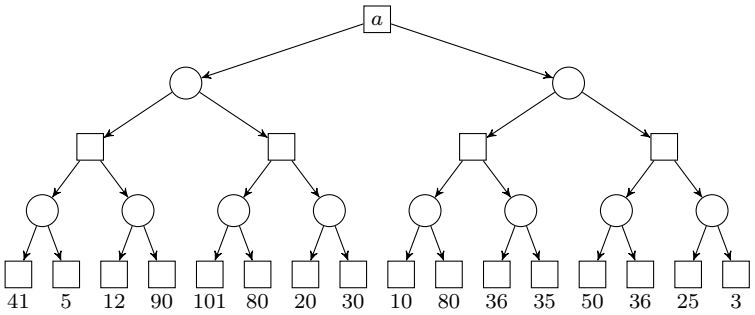
3.1 Drzewo gry

Załóżmy, że dla pewnej sytuacji chcemy znaleźć możliwie najlepszy ruch. W tym celu rozważmy wszystkie sytuacje, jakie otrzymamy po wykonaniu przez nas jednego ruchu. W otrzymanych sytuacjach ruch ma teraz przeciwnik. Dla każdej z nich rozważamy wszystkie możliwe ruchy, jakie może z kolei on wykonać. Wtedy dochodzimy do sytuacji, w których my mamy ruch, itd. W ten sposób budujemy drzewo, w którym wierzchołkami są sytuacje w grze, a krawędziami możliwe posunięcia. Proces ten wykonujemy tak długo, aż dojdziemy do sytuacji końcowych.

Powstałe drzewo w większości znanych gier jest jednak zbyt duże, by móc je całe odtworzyć. W tym celu drzewo obcinamy na pewnej głębokości, w ten sposób, że liście niekoniecznie są stanami końcowymi. Dla liści drzewa musimy jakoś określić ich jakość, oznaczającą jak bardzo w tych stanach jesteśmy blisko wygranej bądź przegranej. W tym celu należy utworzyć funkcję oceniającą, która dla danego stanu gry zwraca liczbę całkowitą. Dla sytuacji końcowych możemy zwracać jakieś duże wartości – dużą dodatnią liczbę całkowitą dla wygranej i dużą ujemną liczbę całkowitą dla przegranej.

Problem tworzenia funkcji oceny jest trudny i na ogół wymaga pomysłów specyficznych dla danej gry. Często używa się tu zaawansowanych metatechnik, takich jak programowanie genetyczne, sieci neuronowe czy regresja liniowa. Są to tematy zbyt obszerne, żeby je tutaj omówić. Przykładowo, w szachach funkcję oceny dzieli się na trzy aspekty: materiał, mobilność i strukturę pionów. Materiał to posiadane figury. Zwyczajowo piony mają wartość 100, skoczek i goniec 300, wieża 500, hetman 900, a król $+\infty$. W mobilność wchodzi liczba atakowanych pól, liczba możliwych ruchów każdej z figur, itp. W strukturze pionów ważne jest, żeby wzajemnie się chroniły, czyli na przykład należy unikać dziur (kolumn niezawierających pionów) i zdublowanych pionów (pionów w jednej kolumnie). We współczesnych programach szachowych ocena jest znacznie bardziej skomplikowana, niemniej jednak zawiera wymienione wyżej pomysły.

Przykładowe drzewo gry przedstawione jest na rysunku 2. Przy liściach zaznaczyliśmy wartości funkcji oceny.



Rysunek 2: Przykładowe drzewo gry.

3.2 Algorytm minimaks

Mając takie drzewo jak na rysunku 2, chcemy znaleźć najlepszy ruch z korzenia drzewa (a). Podczas rozgrywki na tym drzewie naszym celem jest zejście do liścia o największej wartości. Przeciwnik stara się nam w tym przeszkodzić, więc jego celem jest dojście do liścia o najmniejszej wartości. Wartość, do jakiej dojdziemy z danego wierzchołka, możemy obliczyć dynamicznie od dołu. W wierzchołku □ my mamy ruch, więc wybieramy syna z większą wartością, a w wierzchołku ○ ruch wykonuje przeciwnik, więc wybieramy syna z mniejszą wartością. Innymi słowy, w □ maksymalizujemy wartość (taki wierzchołek nazywamy więc wierzchołkiem *max*), a w ○ minimalizujemy wartość (wierzchołek *min*). Algorytm 2 przedstawia tę metodę w wersji rekurencyjnej.

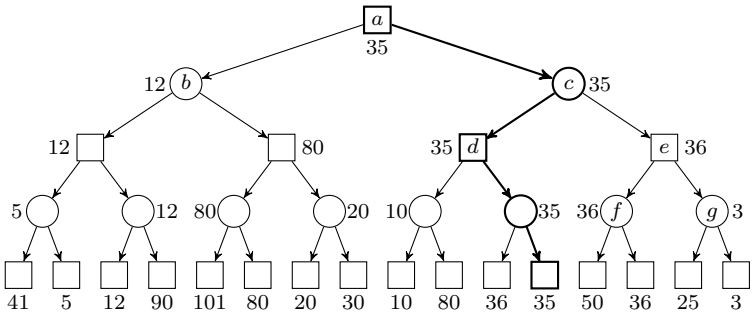
Algorytm 2 Minimaks.

```
1: function MINIMAKS(v)
2:     if v jest liściem then return OCENA(v)
3:     if v jest max then
4:         return max{MINIMAKS(s) | s jest synem v}
5:     else ▷ v jest min
6:         return min{MINIMAKS(s) | s jest synem v}
```

Wartości, jakie otrzymamy dla przykładowego drzewa gry, przedstawione są na rysunku 3. Na rysunku została wyróżniona optymalna rozgrywka obu graczy. Widzimy, że w stanie a optymalny dla nas jest ruch do stanu c.

3.3 Algorytm alfabeta

Można zauważyć, że przy obliczaniu wartości minimaks w celu wyznaczenia optymalnego ruchu ze stanu a nie potrzebujemy przeglądać niektórych wierzchołków drzewa. Przypuśćmy, że podczas obliczania wartości stanu c obliczyliśmy już, że wartość stanu d to 35. Przetwarzając stan e, obliczamy, że wartość stanu f wynosi 36. Teraz możemy zauważyć, że wartość stanu g nie jest nam potrzebna. Otóż znajomość wartości g pozwoli nam ewentualnie stwierdzić, że



Rysunek 3: Przykładowe drzewo gry z wartościami we wszystkich wierzchołkach.

wartość e jest jednak większa niż 36. W każdym razie wiemy, że wartość stanu e będzie co najmniej 36, a co za tym idzie, będzie większa niż wartość stanu d. W związku z tym przeciwnik w stanie c wybierze ruch prowadzący do d niezależnie od tego, jaka jest wartość stanu g.

Istnieje ogólna metoda pozwalająca stwierdzać, których wierzchołków nie musimy już przeglądać. W tym celu modyfikujemy algorytm MINIMAKS tak, aby niekoniecznie zwracał dokładne wartości. Modyfikację tę nazwiemy ALFABETA. Funkcji ALFABETA, oprócz wierzchołka, przekazujemy dwa parametry α i β. Niech w oznacza wartość zwracaną przez wywołanie ALFABETA(v, α, β). Będziemy żądać następującego warunku:

$$\begin{aligned} &\text{jeśli } w \le \alpha, && \text{to } \textsc{Minimaks}(v) \le w, \\ &\text{jeśli } \alpha < w < \beta, && \text{to } \textsc{Minimaks}(v) = w, \\ &\text{jeśli } \beta \le w, && \text{to } \textsc{Minimaks}(v) \ge w. \end{aligned} \tag{1}$$

Innymi słowy, ALFABETA(v, α, β) zwróci dokładną wartość, jeśli będzie ona w przedziale (α, β), ograniczenie górne na wartość minimaks, jeśli zwrócona wartość będzie nie większa niż α, oraz ograniczenie dolne, jeśli wynik będzie nie mniejszy niż β. Przedział (α, β) nazywamy często *oknem wywołania* funkcji ALFABETA.

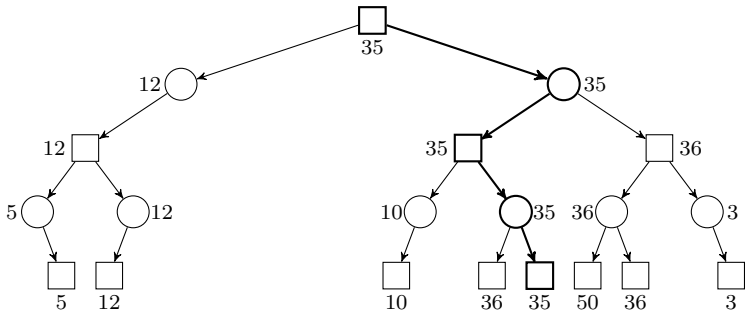
Dla korzenia chcemy uzyskać dokładną wartość, więc wystarczy przyjąć $\alpha = -\infty$ i $\beta = +\infty$. W wywołaniach dla synów parametry te możemy modyfikować. Przypuśćmy, że jesteśmy w wierzchołku max, i załóżmy, że znaleźliśmy już ruch dający wartość x. Wówczas dla wszystkich pozostałych ruchów z tej sytuacji, jeżeli któryś z nich będzie miał wartość nie większą niż x, to nie będzie nas już interesowała dokładna wartość. Zatem możemy parametr α powiększyć do x przy wywoływaniach dla kolejnych synów. No dobrze, a kiedy będziemy mogli stwierdzić, że jakiegoś syna nie trzeba już odwiedzać? Przypuśćmy, że jesteśmy w wierzchołku max. Jeżeli znajdziemy ruch, którego wartość jest większa niż β, to możemy w tym momencie zakończyć przeszukiwanie i zwrócić znalezioną wartość, zachowując warunki dla funkcji ALFABETA. W ten sposób wykonujemy tzw. *β-cięcie* i nie przeszukujemy części drzewa. Analogicznie wprowadzamy *α-cięcie* w wierzchołku min. Całe rozumowanie podsumowane jest algorytmem 3. Pozostawiamy jako ćwiczenie uzasadnienie, że spełniony jest niezmiennik (1).

Algorytm 3 Alfabeta.

```
1:  function ALFABETA(v, α, β)
2:      if v jest liściem then return OCENA(v)
3:      if v jest max then
4:          r ← −∞
5:          for all s jest synem v do
6:              x ← ALFABETA(s, max(r, α), β)
7:              if x ≥ β then return x                    ▷ β-cięcie
8:              r ← max(r, x)
9:      else ▷ v jest min
10:         r ← +∞
11:         for all s jest synem v do
12:             x ← ALFABETA(s, α, min(r, β))
13:             if x ≤ α then return x                    ▷ α-cięcie
14:             r ← min(r, x)
15:     return r
```

3.4 Ulepszenia alfabety

Aby algorytm alfabeta był możliwie najbardziej skuteczny, najważniejsze jest, by w wywołaniach rekurencyjnych wybierać jako pierwszy ruch, który jest najlepszy z punktu widzenia gracza zagrywającego. W ten sposób można dokonać największej liczby cięć. Rysunek 4 pokazuje, jaka część drzewa zostałaby przejrzana przez algorytm alfabeta przy optymalnym wyborze ruchów. Wierzchołki z przykładowego drzewa, które nie zostałyby przejrzane, zostały ukryte.



Rysunek 4: Działanie alfabety przy optymalnym wyborze ruchów.

Asymptotycznie, przy takim perfekcyjnym doborze ruchów, liczba przejrzanych wierzchołków drzewa wynosi tyle, co pierwiastek kwadratowy z rozmiaru całego drzewa. Pozwoliłoby to nam przeglądać drzewo gry dwa razy głębiej niż przy zwykłym minimaksie. Istnieje cała gama różnych technik, które pomagają zwiększać liczbę cięć i w ogóle szybkość alfabety. Przedstawimy kilka najważniejszych.

3.4.1 Tablica transpozycji

W algorytmach minimaks i alfabeta zakładaliśmy, że operujemy na drzewie gry. W rzeczywistości niekoniecznie jest to drzewo, gdyż mogą istnieć różne sekwencje ruchów, które prowadzą do tych samych sytuacji. Pary takich sekwencji nazywa się *transpozycjami*.

W związku z tym, że większość gier można reprezentować dowolnymi grafami, i to do tego często z cyklami, w algorytmie alfabeta wielokrotnie będziemy powtarzać obliczenia dla tego samego stanu. Aby uniknąć tego typu sytuacji, należy zapamiętywać wyniki w celu późniejszego ich wykorzystania. Można utworzyć słownik, którego kluczem będzie sytuacja, a wartością wynik minimaksa. Niestety w praktyce okazuje się, że w ułamku sekundy zapełnimy całą pamięć operacyjną, więc nie możemy pamiętać wszystkich obliczonych wartości, lecz tylko niektóre. Ponadto, w takim słowniku do zapamiętania klucza potrzebujemy dużej ilości danych, aby opisać sytuację w grze.

Dodatkowy problem jest taki, że obliczenia powtarzane dla jednej sytuacji v mogą być różnej jakości. Wynika to z faktu, że do v możemy dojść za pomocą różnej liczby ruchów, a co za tym idzie, alfabeta odpalona dla v będzie wykonywała przeszukiwanie w głąb dla różnych głębokości d. Przez *głębokość* rozumiemy tutaj odległość v od liści drzewa, czyli od wierzchołków, w których dokonujemy oceny.

Powyższe problemy rozwiązujemy w następujący sposób. Załóżmy, że mamy pewną funkcję h (nazwiemy ją funkcją haszującą), która dla dowolnej sytuacji zwraca liczbę całkowitą z przedziału $[0, R)$, gdzie R jest pewną ustaloną dużą liczbą (na ogół $R = 2^{64}$, czyli $h(\cdot)$ zwraca liczbę 64-bitową). Funkcja h powinna być deterministyczna i mieć tę własność, że prawdopodobieństwo zwrócenia tej samej liczby dla dwóch różnych sytuacji jest małe, a najlepiej żeby wynosiło $1/R$. Każdą sytuację będziemy reprezentować liczbą całkowitą z przedziału $[0, R)$ zwracaną przez funkcję h.

Tworzymy dużą tablicę mem rozmiaru N. Każdy element tablicy będzie przechowywał informację o jednym stanie gry. N powinno być tak dobrane, aby tablica mem mieściła się w pamięci. Aby zwiększyć efektywność operacji, N przyjmuje się jako potęgę dwójki. Każdy element tablicy zawiera następujące pola:

- liczbę całkowitą $r \in [0, R)$ reprezentującą sytuację,
- liczby całkowite x_d, x_g, które oznaczają dolne i górne ograniczenie na wartość minimaks danego stanu,
- głębokość d.

Funkcja ALFABETA zwraca dokładną wartość, ograniczenie górne lub ograniczenie dolne. Używając dwóch liczb całkowitych x_d i x_g, możemy reprezentować dowolny rodzaj tej wartości. Parametr d oznacza głębokość, z jaką została wywołana alfabeta dla danego stanu.

Tablicę mem z początku inicjujemy pustymi wartościami jakoś symbolicznie, np. przyjmując $r = 0$, bądź $d = -1$. Funkcję ALFABETA(v, α, β) modyfikujemy następująco. Bierzemy reprezentanta stanu $r = h(v)$. Obliczamy indeks stanu w tablicy mem jako $i = r \bmod N$. Sprawdzamy zawartość $mem[i]$. Jeżeli jest tam użyteczna wartość, to ją zwracamy i nie wykonujemy właściwej treści funkcji. Użyteczna wartość to taka, która została obliczona z wystarczającą głębokością

d i w której ograniczenia x_d i x_g umożliwiają zwrócenie wartości spełniającej niezmiennik (1). W przeciwnym przypadku musimy obliczyć wartość od zera. Przed zwróceniem jej zapamiętujemy wynik w polu $mem[i]$, nadpisując jego zawartość.

W powyższej technice kryje się jedna drobna luka. Otóż może się zdarzyć, że dwa stany będą miały tę samą wartość funkcji h i odczytując z tablicy mem wartość dla jednego z nich, pobierzemy wartość obliczoną dla innego. Wtedy ta wartość jest fałszywa. Jednak takie sytuacje są na tyle rzadkie, że nie wpływają na ostateczny wynik i milcząco ignorujemy ten problem.

Dodatkowo chcemy, aby nasza tablica wypełniała się w miarę równomiernie, zatem kolejna właściwość, jaką powinna mieć funkcja h, jest taka, aby wartości zwracane przez tę funkcję modulo N rozkładały się równomiernie w przedziale $[0, N)$.

Haszowanie Zobrista. Pozostaje pytanie, jak wybrać funkcję haszującą h? Istnieje bardzo skuteczna technika, którą stosuje się przede wszystkim w grach planszowych, ale nie tylko. Pokażemy, jak zbudować taką funkcję na przykładzie warcabów.

Plansza składa się z 64 pól, z czego wszystkie akcje odbywają się na polach czarnych, czyli gra toczy się na 32 polach. Każde pole może być puste lub może na nim stać pionek bądź damka jednego bądź drugiego gracza. Zatem to, co znajduje się na danym polu, możemy reprezentować liczbą całkowitą od 0 do 4. Tworzymy dwuwymiarową tablicę $magic[0..31][0..4]$, którą wypełniamy losowymi liczbami z zakresu $[0, R)$. Z jej użyciem możemy skonstruować funkcję h. Dla zadanej sytuacji v niech $p[i]$ dla $i = 0, \ldots, 31$ oznacza zawartość i-tego pola – liczbę od 0 do 4. Przyjmujemy:

$$h(w) = \sum_{i=0}^{31} magic[i][p[i]] \bmod R. \tag{2}$$

Tak zbudowana funkcja haszująca ma wszystkie wymagane własności. Co więcej, daje się ją szybko obliczać w sposób inkrementalny. Otóż dowolny ruch w warcabach powoduje zmianę zawartości dwóch pól. Aby uzyskać wartość funkcji haszującej dla nowo otrzymanego stanu, wystarczy zatem wykonać dwa odejmowania i dwa dodawania modulo R. Zanotujmy, że we wzorze (2) możemy zamiast dodawania użyć operacji xor, jeżeli R jest potęgą dwójki, co stosuje się w praktyce.

3.4.2 Iteracyjne pogłębianie

Głębokość, na jaką odsuwamy ocenę pozycji, czyli głębokość drzewa gry, w najprostszym podejściu ustala się na sztywno tak, aby czas działania był praktycznie akceptowalny, czyli rzędu kilku sekund na ruch. To podejście ma tę wadę, że zależnie od sytuacji w grze, czas działania alfabety może znacznie się wydłużać. Chcielibyśmy mieć lepszą kontrolę nad tym czasem.

Zauważmy, że wykonanie alfabety do głębokości $d - 1$ trwa o rząd wielkości krócej niż wykonanie alfabety do głębokości o jeden większej, d. Możemy zatem wykonywać alfabetę dla kolejnych możliwych głębokości: wpierw jedno posunięcie w głąb, następnie dwa, itd. W ten sposób możemy ustalić czas na ruch i

zakończyć przeszukiwanie, jak nam się skończy czas. Najlepszy ruch wybieramy z wyniku ostatniego kompletnego wywołania alfabety.

To podejście okazuje się bardzo dobrze współgrać z techniką tablicy transpozycji. Otóż przy przejściu do głębokości o jeden większej możemy nie czyścić tej tablicy! Wyniki z poprzedniego wywołania alfabety mogą być użyteczne, ale główny użytek, jaki możemy zrobić, to wykorzystywać tablicę transpozycji do segregowania ruchów przy wywołaniach rekurencyjnych. W ten sposób wykorzystujemy informacje z poprzednich wywołań.

W tym celu, w tablicy transpozycji pamiętamy dodatkowe pole, reprezentujące najlepszy ruch w danym stanie. Powiedzmy, że w stanie v wykonujemy algorytm alfabeta do głębokości d. Szukamy wpisu o tym stanie w tablicy transpozycji. Jeśli taki wpis istnieje, to jako pierwszy rozważamy ruch, który jest tam zapisany. Jeżeli nie znajdziemy takiego wpisu, to musimy próbować innych metod do wybrania obiecującego ruchu. Jedną z takich technik jest wołanie „płytkiej" alfabety, na przykład do głębokości $d-2$. Takie wywołanie trwa zaniedbywalnie krótko w porównaniu z wywołaniem do głębokości d, a daje duże szanse na wczesne znalezienie cięć.

3.4.3 Zmniejszanie okna wywołania

Zamiast dla aktualnego stanu gry v wywoływać $\textsc{Alfabeta}(v, -\infty, +\infty)$, można spróbować wstępnie oszacować wartość minimaksową x stanu v i wywołać alfabetę z mniejszym oknem, na przykład $(x-d, x+d)$. Przy tym d ma być na tyle duże, że spodziewamy się, że wartość minimaksowa stanu v z dużym prawdopodobieństwem znajdzie się w tym przedziale. Przy użyciu iteracyjnego pogłębiania, za wartość x możemy wziąć wynik z poprzedniego wywołania alfabeta dla głębokości o jeden mniejszej.

Niech $y = \textsc{Alfabeta}(v, x-d, x+d)$. Jeżeli $y \in (x-d, x+d)$, to mamy dokładną wartość stanu v. Jeżeli jednak y będzie poza przedziałem, to w celu znalezienia dokładnej wartości musimy ponowić nasze wywołanie z innym oknem. Jeśli na przykład $y \geq x+d$, to za nowe okno możemy wziąć przedział $(y, +\infty)$.

3.4.4 Zwiadowca

Można też zmniejszać okno wywołania w dowolnych wierzchołkach, a nie tylko w korzeniu drzewa. Otóż, załóżmy, że jesteśmy w stanie typu max i obliczyliśmy wartość minimaksową r dla pierwszego syna. Dla pozostałych synów jedynie chcemy pokazać, że ich wartość jest nie większa niż r. W tym celu okno wywołania możemy zmniejszyć do $(r, r+1)$. Takie okno nazywamy *pustym oknem*, gdyż do przedziału $(r, r+1)$ nie wpada żadna liczba całkowita. Wołanie alfabety dla syna z pustym oknem nazywamy potocznie *puszczaniem zwiadowcy*. Jeżeli wartość minimaksowa, po puszczeniu zwiadowcy, dla któregoś z synów okaże się jednak większa od r, to wtedy nie jest ona dokładna i niestety potrzebujemy drugiego wywołania.

U podstaw tej metody jest założenie, że ruchy są dobrze posegregowane – najlepiej jeśli pierwszy rozważany ruch jest optymalny. Wtedy ponowne przebiegi dla kolejnych synów powinny zdarzać się dosyć rzadko. Aby to założenie było spełnione, w praktyce tę technikę należy stosować w połączeniu z iteracyjnym pogłębianiem i tablicą transpozycji. Przy takim użyciu daje ona istotne przyspieszenie.

W implementacji technika zwiadowcy polega na zastąpieniu w algorytmie 3 linii

6: $x \leftarrow \textsc{Alfabeta}(s, \max(r, \alpha), \beta)$

następującymi:

$r' \leftarrow \max(r, \alpha)$
$x \leftarrow \textsc{Alfabeta}(s, r', r'+1)$ ▷ puszczanie zwiadowcy
if $x > r' \wedge x < \beta$ **then** ▷ jednak potrzebujemy obliczyć dokładną wartość
$x \leftarrow \textsc{Alfabeta}(s, x, \beta)$

Analogiczną modyfikację należy wykonać dla wierzchołków typu min. Można też zauważyć, że drugie wywołanie dla syna może być zbędne, jeżeli jest on ostatnim rozważanym synem i nie potrzebujemy jego dokładnej wartości.

4 Metody Monte Carlo

Rozważmy następującą metodę oceniania pozycji, która nie wymaga żadnej wiedzy o grze poza implementacją zasad. Dla analizowanej pozycji wykonujemy losową rozgrywkę w następujący sposób. Pierwszy ruch wybieramy losowo z równym prawdopodobieństwem ze zbioru dostępnych ruchów, po czym go wykonujemy. Następnie dla otrzymanej pozycji ponownie losujemy ruch w analogiczny sposób i go wykonujemy. Kontynuujemy takie losowe wykonywanie ruchów, aż dojdziemy do sytuacji końcowej. Wtedy na podstawie zasad gry potrafimy podać wynik tej rozgrywki. Załóżmy, że możliwe wyniki to wygrana i przegrana. Aby ocenić daną pozycję, wykonujemy dużą liczbę losowych rozgrywek n, wśród których w rozgrywek zakończyło się naszą wygraną. Pozycję oceniamy wartością w/n. Taką ocenę nazywamy *oceną Monte Carlo*.

Ocenianie pozycji w powyższy sposób, zależnie od typu gry, daje mniej lub bardziej sensowne wyniki. Taką ocenę z powodzeniem stosuje się w grach takich jak *go* czy *hex*, biorąc na przykład $n = 10\,000$. Przeszukiwanie drzewa gry możemy stosować w tym wypadku do niedużych głębokości ze względu na czasochłonność funkcji oceniającej, a siła otrzymanego programu i tak zależy bardziej od specyfiki gry.

W związku z tym trzeba dobrze dobrać n. Intuicyjnie, im większe n weźmiemy, tym dokładniejsza będzie wartość oceny. Jednakże ocena Monte Carlo na ogół nie jest miarodajna, nawet gdybyśmy wzięli $n = +\infty$, w związku z tym n wcale nie musi być bardzo duże i dobrze dobierać ten parametr drogą eksperymentów dla różnych gier.

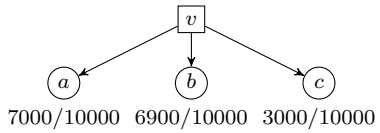
Ocena Monte Carlo daje jakąś liczbę; pytanie: jaką? Otóż jest to jakaś ocena wartości sytuacji ze strategicznego punktu widzenia, tzn. wartość danego zagrania może mieć znaczenie dopiero w końcowej fazie rozgrywki. Na pierwszy rzut oka wydaje się to niezbyt sensowne, jednak nieoczekiwanie dobrze sprawdza się w niektórych grach.

Zamiast losowych rozgrywek można przeprowadzać bardziej inteligentne partie. Na przykład program szachowy *Rybka* ma taką funkcję, która z zadanej sytuacji przeprowadza setki wysokiej jakości nieco losowych partii. W ten sposób dla każdego zagrania jesteśmy w stanie wyznaczyć procent wygranych partii, który ocenia dane zagranie znacznie bardziej kompleksowo. Może się wszakże zdarzyć, że dane zagranie daje zysk dopiero w dalszej fazie rozgrywki, co jest daleko poza zasięgiem zwykłego przeszukiwania drzewa i normalnej funkcji oce-

niającej. W ten sposób analizuje się dzisiaj trudne, strategiczne zagrania w grach arcymistrzów.

4.1 Granica ufności

Stosowanie zwykłego przeszukiwania drzewa wraz z oceną Monte Carlo nie jest zbyt efektywne. Rozważmy sytuację na rysunku 5. Ze stanu v mamy trzy możliwe ruchy. Dla każdego z nich przeprowadziliśmy 10 000 losowych rozgrywek. Wyraźnie ruch c jest gorszy niż ruchy a i b. Z kolei różnica między ruchami a i b jest niewielka i do końca nie wiemy, który z nich jest lepszy. Otóż może by wystarczyło dla ruchu c przeprowadzić tylko 1000 rozgrywek, aby z dużym prawdopodobieństwem stwierdzić, że ten ruch jest jednak znacznie gorszy, a pozostałe 9 000 przeznaczyć na rozstrzygnięcie, który z ruchów a i b jest lepszy. W ten sposób, z użyciem tej samej łącznej liczby losowych rozgrywek, bylibyśmy w stanie z większą pewnością wybrać lepszy ruch.



Rysunek 5: Przykładowe oceny synów metodą Monte Carlo.

Stajemy przed problemem, jak rozdzielać rozgrywki pomiędzy synów, aby z jak największym prawdopodobieństwem wybrać optymalny ruch. Nie wchodząc w szczegóły, opiszemy powszechnie stosowaną metodę.

Załóżmy, że dla pewnej sytuacji v przeprowadziliśmy n_v losowych rozgrywek, wśród których było w_v wygranych. Wartość oczekiwana oceny sytuacji wynosi

$$E_v = \frac{w_v}{n_v}.$$

Załóżmy, że rzeczywista ocena stanu wynosi x (tzn. jakbyśmy przeprowadzili nieskończenie wiele rozgrywek). Interesuje nas przedział taki, że prawdopodobieństwo, że x należy do tego przedziału, jest dosyć duże (np. 95%). Rachunek prawdopodobieństwa mówi, że dla pewnego ustalonego ε zachodzi

$$\Pr(x \in [E_v - c\sigma_v, E_v + c\sigma_v]) \geq 1 - \varepsilon \text{ dla } \sigma_v^2 = \frac{1}{n_v}, \tag{3}$$

gdzie c jest pewną stałą zależną od ε. Przedział $[E_v - c\sigma_v, E_v + c\sigma_v]$ nazywamy *przedziałem ufności*.

Stosując przedziały ufności, można lepiej zaplanować rozdzielanie losowych rozgrywek pomiędzy synów. Rozważmy stan v typu max. Dla każdego z jego synów s wykonaliśmy n_s losowych rozgrywek, wśród których w_s było wygranych. Zakładamy, że wartość syna s wpada do przedziału $[E_s - c\sigma_s, E_s + c\sigma_s]$, po cichu ignorując, że istnieje małe prawdopodobieństwo tego, że może ona być poza przedziałem ufności. Którego syna teraz wybrać do przeprowadzenia kolejnej losowej rozgrywki? Ponieważ szukamy ruchu dającego największą wartość, więc wybieramy taki stan, który oferuje największą możliwą dostępną wartość, a mianowicie górną granicę przedziału ufności: $E_s + c\sigma_s$.

Należy rozwiązać jeszcze dwa problemy. Po pierwsze, synowie s, którzy nie mieli jeszcze przydzielonych żadnych losowych rozgrywek, powinni być wybierani jako pierwsi. Symbolicznie można to zrobić, przypisując im $\sigma_s = +\infty$. Po drugie, może zdarzyć się taka sytuacja, że jeden z synów (s) po małej liczbie losowych rozgrywek nieszczęśliwie będzie mieć małą średnią i małą górną granicę ufności. Istnieje niewielkie prawdopodobieństwo, że jednak ten ruch jest najlepszy. Problem ten obchodzi się przez przemnożenie wartości σ_s^2, zdefiniowanej we wzorze (3), przez pewną wolno rosnącą funkcję h od łącznej liczby losowych rozgrywek n_v przeprowadzonych dla wszystkich synów, czyli $n_v = \sum_{t \text{ syn } v} n_t$. Najczęściej przyjmuje się $h(n_v) = \log n_v$, ale może być to dowolna inna funkcja – czasami lepiej sprawdza się $h(n_v) = \sqrt{n_v}$. Dobór optymalnej funkcji zależy między innymi od konkretnej gry i dokonuje się go na podstawie eksperymentów. Dodajmy, że w przypadku użycia funkcji h dobór stałej c jest nieistotny, więc odtąd zakładamy, że $c = 1$.

W przypadku wierzchołka typu min, wybieramy syna z najmniejszą dolną granicą ufności. Funkcja wyboru syna przedstawiona jest w algorytmie 4.

Algorytm 4 Wybór syna metodą granicy ufności.

```
1: function GranicaUfności(v, s)
2:     E_s ← w_s / n_s, σ_s ← sqrt(log n_v / n_s)
3:     if v jest max then
4:         return E_s + σ_s
5:     else ▷ v jest min
6:         return E_s − σ_s
7: function WybierzSyna(v)
8:     return syn s stanu v optymalizujący wartość GranicaUfności(v, s)
```

4.2 Przeszukiwanie drzewa Monte Carlo

Metoda granicy ufności stosuje się do jednego stanu, gdy chcemy wybrać najlepszy ruch na podstawie wartości synów. Odpowiada to przeszukiwaniu drzewa gry do głębokości 1. Jak przydzielać losowe rozgrywki w drzewie gry w przypadku, gdy chcemy ocenę pozycji obliczyć znacznie głębiej?

Istnieje prosty i niezwykle skuteczny algorytm, który potrafi budować drzewo gry przyrostowo, przydzielając losowe rozgrywki najbardziej obiecującym gałęziom drzewa. W danym momencie algorytm przechowuje pewne drzewo gry. Na początku jest to korzeń i jego synowie. W każdym wierzchołku v trzymane są dwie wartości, n_v i w_v. n_v reprezentuje liczbę przeprowadzonych rozgrywek, w których pierwsze ruchy pokrywają się z ruchami, jakie trzeba wykonać z korzenia drzewa do wierzchołka v. Natomiast w_v oznacza liczbę zwycięstw wśród tych rozgrywek.

Kolejną rozgrywkę tworzymy w następujący sposób. Pierwsze ruchy rozgrywki wybieramy, stosując metodę granicy ufności, wędrując od korzenia drzewa do liścia. Po osiągnięciu liścia dalszą część rozgrywki przeprowadzamy zupełnie losowo. Po utworzeniu rozgrywki aktualizujemy statystyki n_v i w_v we wszystkich wierzchołkach na ścieżce od korzenia do liścia, gdyż ta rozgrywka dotyczy właśnie tych wierzchołków. Ewentualnie wcześniej rozwijamy osiągnięty

liść drzewa, dodając do drzewa wszystkich synów reprezentujących stany, które można osiągnąć przez wykonanie jednego ruchu.

Są różne heurystyki mówiące, kiedy rozwijać liść. Nie można tego robić za często, żeby po pierwsze, drzewo mieściło się cały czas w pamięci, a po drugie, żeby zbyt szybko wybory nie były determinowane granicą ufności. Z drugiej strony ma być to na tyle często, aby rzeczywiście drzewo przypominało drzewo minimaksowe. Standardowo liść v rozwijamy wtedy, gdy liczba rozgrywek z tego wierzchołka n_v przekroczy pewien ustalony próg N_0. Dobór wartości N_0 zależy od gry i na ogół najlepiej się sprawdza, gdy ustawi się ją na średnią liczbę możliwych ruchów, jakie można wykonać w losowej sytuacji (tzw. stopień rozgałęzienia gry). Dla szachów wartość N_0 wynosi mniej więcej 30 w początkowej fazie gry.

Całe powyższe rozumowanie przedstawione jest w postaci pseudokodu algorytmu 5. Algorytm ten w literaturze występuję pod skrótami UCT lub MCTS.

Algorytm 5 Przeszukiwanie drzewa gry Monte Carlo.

```
1:  function Rozgrywka(v)
2:      if v jest liściem then
3:          if n_v ≥ N_0 then
4:              rozwiń v
5:              W ← Rozgrywka(WybierzSyna(v))
6:          else
7:              Przeprowadź rozgrywkę do końca, losowo wybierając ruchy
8:              return 1 dla wygranej, bądź 0 dla przegranej
9:      else
10:         W ← Rozgrywka(WybierzSyna(v))
11:     n_v ← n_v + 1, w_v ← w_v + W
12:     return W
13: function ZnajdźNajlepszyRuch(v)
14:     utwórz korzeń drzewa ze stanem v
15:     rozwiń v                          ▷ korzeń wyjątkowo rozwijamy od razu
16:     while nie skończył się nam czas na ruch do
17:         Rozgrywka(v)
18:     if v jest max then
19:         return syn s stanu v maksymalizujący E_s = w_s / n_s
20:     else ▷ v jest min
21:         return syn s stanu v minimalizujący E_s = w_s / n_s
```

W praktyce stosuje się dodatkowe modyfikacje, aby zwiększyć jego siłę. Na przykład, zamiast losowych rozgrywek, zwiększa się ich jakość, stosując bardzo prostą inteligencję, jednocześnie wciąż dbając o losowość. Powyższy algorytm został stosunkowo niedawno odkryty (w tym wieku) i okazuje się niezwykle skuteczny. Sprawdził się on w wielu grach, takich jak wspomniane już *go* i *hex*, wypierając wcześniej używane metody.

Jego siła wynika z tego, że łączy on w sobie elementy taktyczne ze strategicznymi. Teoria mówi, że drzewo tworzone przez ten algorytm zbiega do pełnego drzewa minimaksowego. Oznacza to, że przy bardzo dużej liczbie losowych rozgrywek metoda ta powinna taktycznie zachowywać się tak, jak zwykłe

algorytmy przeglądania drzewa.

Zastosowania przeszukiwania drzewa Monte Carlo są duże szersze. Dobrze się sprawdza w grach wieloosobowych z elementami losowości (np. w *Osadnikach z Catanu*!), a nawet w grach z niepełną informacją czy też grach jednoosobowych (czyli łamigłówkach). W ITPW (Internetowy Turniej Programów Walczących) 2009 wszystkie najlepsze programy grające w grę karcianą „Planowanie" zostały napisane z użyciem technik Monte Carlo.