Techniki sztucznej inteligencji w programach grających

Jakub PAWLEWICZ[*]

1. Wprowadzenie

Sztuczna inteligencja w grach jest bardzo atrakcyjną dziedziną badań, gdyż opracowane metody można łatwo sprawdzać w praktyce, obserwując siłę programów grających w popularne gry, niejednokrotnie wprowadzających 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 cały 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. Algorytmy przeszukiwania drzewa gry

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 hetmana, aby w kilku posunięciach zbić kilka innych figur przeciwnikowi, zyskując przewagę materialną.

Strategia oznacza z kolei 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 3. Natomiast programy w przypadku większości gier przeważają nad ludźmi w elementach taktycznych, gdyż stosowane są w nich algorytmy przeszukiwania drzewa gry.

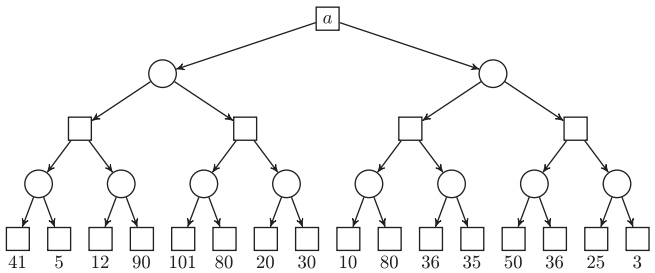
2.1. Drzewo gry

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 1. Przy liściach zaznaczyliśmy wartości funkcji oceny.



Rys. 1. Przykładowe drzewo gry.

2.2. Algorytm minimaks

Mając takie drzewo jak na rysunku 1, 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

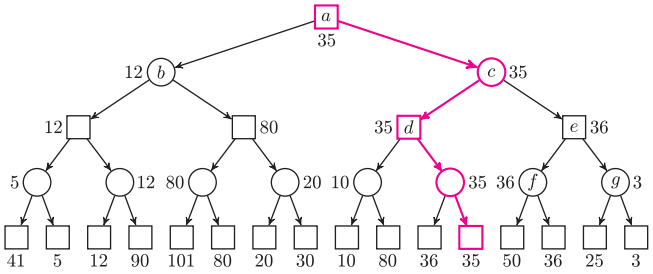
[*]Instytut Informatyki, Uniwersytet Warszawski

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 1 przedstawia tę metodę w wersji rekurencyjnej.

Algorytm 1 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 2. Na rysunku została wyróżniona optymalna rozgrywka obu graczy. Widzimy, że w stanie a optymalny dla nas jest ruch do stanu c.



Rys. 2. Przykładowe drzewo gry
z wartościami we wszystkich wierzchołkach.

2.3. 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 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:

$$(1)\qquad \begin{array}{ll} \text{jeśli } w \leqslant \alpha, & \text{to MINIMAKS}(v) \leqslant w, \\ \text{jeśli } \alpha < w < \beta, & \text{to MINIMAKS}(v) = w, \\ \text{jeśli } \beta \leqslant w, & \text{to MINIMAKS}(v) \geqslant w. \end{array}$$

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.

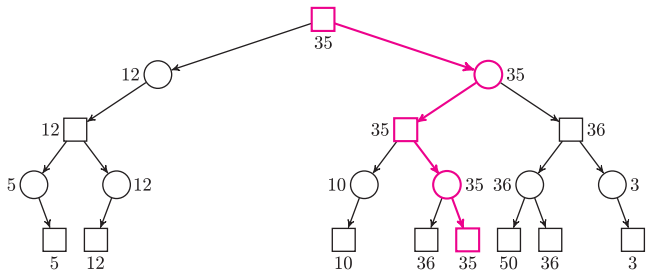
Dla korzenia chcemy uzyskać dokładną wartość, więc wystarczy przyjąć $\alpha = -\infty$ i $\beta = +\infty$. W wywołaniach dla synów parametry te możemy modyfikować. Przypuśćmy, że jesteśmy w wierzchołku max, i załóżmy, że znaleźliśmy już ruch dający wartość x. Wówczas dla wszystkich pozostałych ruchów 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 2. Pozostawiamy jako ćwiczenie uzasadnienie, że spełniony jest warunek (1).

Algorytm 2 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
```

2.4. Ulepszenia alfabety

Aby algorytm alfabeta był możliwie najbardziej skuteczny, najważniejsze jest, by w wywołaniach rekurencyjnych jako pierwszy wybierać ruch, który jest najlepszy z punktu widzenia gracza zagrywającego. W ten sposób można dokonać największej liczby cięć. Rysunek 3 pokazuje, jaka część drzewa zostałaby przejrzana przez algorytm alfabeta przy optymalnym wyborze ruchów. Wierzchołki z przykładowego drzewa, które nie zostałyby przejrzane, zostały ukryte.



Rys. 3. Działanie alfabety przy optymalnym wyborze ruchów.

Asymptotycznie, przy takim perfekcyjnym doborze ruchów, liczba przejrzanych wierzchołków drzewa jest pierwiastkiem kwadratowym z rozmiaru całego drzewa. Pozwoliłoby to nam przeglądać drzewo gry dwa razy głębiej niż przy zwykłym

minimaksie. Istnieje jednak cała gama różnych technik, które pomagają zwiększać liczbę cięć i w ogóle szybkość alfabety. W poniższym paragrafie wymieniamy w skrócie kilka z nich.

Dotychczas rozważaliśmy tylko drzewo gry, a w praktyce gry mogą być dowolnymi grafami skierowanymi, także z cyklami. W takim przypadku korzystne jest eliminowanie powtórnych obliczeń, w czym pomaga użycie tzw. tablicy transpozycji. Z kolei w zakresie lepszej kontroli nad czasem i trafniejszego doboru ruchów podstawową techniką jest iteracyjne pogłębianie przeszukiwania. Te dwie metody łącznie nieznacznie przybliżają nas do możliwie największej liczby cięć, ale jest to jeszcze dalekie od ideału. Istnieje szereg technik, które zmniejszają sztucznie okno wywołania, aby wymusić większą liczbę cięć kosztem niedokładnego wyniku, co może powodować konieczność wykonywania alfabety dla tego samego stanu więcej niż raz, z różnymi oknami wywołania. Najpowszechniej stosowana jest tu technika zwiadowcy. Czytelników zainteresowanych wspomnianymi usprawnieniami odsyłamy do rozszerzonej wersji artykułu na stronie internetowej *Delty*.

3. 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 wykonujemy go. 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 przypadku do niedużych głębokości ze względu na czasochłonność funkcji oceniającej, a siła otrzymanego programu i tak zależy bardziej od specyfiki gry.

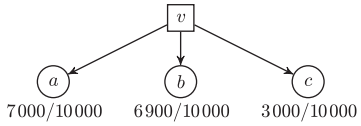
W związku z tym trzeba dobrze dobrać n. Intuicyjnie, im większe n weźmiemy, tym dokładniejsza będzie wartość oceny. Jednakże ocena Monte Carlo na ogół nie jest miarodajna, nawet gdybyśmy wzięli $n = +\infty$, w związku z tym n wcale nie musi być bardzo duże. Dobrze jest dobierać ten parametr drogą eksperymentów 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 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 stanowi znacznie bardziej kompleksową ocenę zagrania. 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 oceniającej. W ten sposób analizuje się dzisiaj trudne, strategiczne zagrania w grach arcymistrzów.

3.1. Granica ufności

Stosowanie zwykłego przeszukiwania drzewa wraz z oceną Monte Carlo nie jest zbyt efektywne. Rozważmy sytuację z rysunku 4. 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 1 000 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.



Rys. 4. 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 taki przedział, iż prawdopodobieństwo tego, że x należy do niego, jest dosyć duże (np. 95%). Rachunek prawdopodobieństwa mówi, że dla pewnego ustalonego ε zachodzi

$$\text{(2)} \qquad \mathrm{P}(x \in [E_v - c\sigma_v, E_v + c\sigma_v]) \geqslant 1 - \varepsilon \quad \text{dla } \sigma_v^2 = \frac{1}{n_v},$$

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 fakt, iż 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 (2), 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 3.

Algorytm 3 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)
```

3.2. 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. Wartość 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, po pierwsze dlatego, żeby 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 4. Algorytm ten w literaturze występuje pod skrótami UCT lub MCTS. W praktyce stosuje się dodatkowe modyfikacje, aby zwiększyć jego siłę. Na przykład, zamiast losowych rozgrywek zwiększa się ich jakość, stosując bardzo proste metody, jednocześnie wciąż dbając o losowość. Algorytm ten 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 dużo 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 Internetowym Turnieju Programów Walczących 2009 wszystkie najlepsze programy grające w grę karcianą *Planowanie* zostały napisane z użyciem technik Monte Carlo.

Algorytm 4 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
```