GRY PLANSZOWE

Najwcześniejsze zastosowanie sztucznej inteligencji w grach komputerowych to przeciwnicy w symulowanych wersjach popularnych gier planszowych. Na Zachodzie szachy są archetypową grą planszową, a ostatnie 40 lat przyniosło dramatyczny wzrost możliwości komputerów grających w szachy.

W tym samym czasie badano inne gry, takie jak Kółko i krzyżyk, Connect Four, Reversi (Othello) i Go, w każdym przypadku zakończone sztuczną inteligencją, która może pokonać najlepszych ludzkich przeciwników. Techniki sztucznej inteligencji potrzebne do tego, aby komputer grał w gry planszowe, są zupełnie inne niż pozostałe w tej książce. W przypadku gier czasu rzeczywistego, które dominują na listach przebojów, ten rodzaj sztucznej inteligencji ma tylko ograniczone zastosowanie. Czasami jest używana jako warstwa strategiczna, podejmująca długoterminowe decyzje w grach wojennych, ale nawet wtedy tylko wtedy, gdy gra została zaprojektowana jako coś w stylu planszówki. Najlepsi przeciwnicy AI do Go, Szachy, Warcaby, Backgammona i Reversi - ci, którzy są w stanie pokonać najlepszych graczy - używają dedykowanego sprzętu, algorytmów lub optymalizacji opracowanych specjalnie dla niuansów ich strategii. To może się zmieniać. Wraz z pojawieniem się sztucznej inteligencji do gier planszowych do głębokiego uczenia się, szczególnie tych stworzonych przez firmę Deep Mind, istnieją pewne oznaki, że podejścia na poziomie elitarnym można zastosować do wielu gier. W chwili pisania tego tekstu okaże się, czy sukces może zostać powtórzony przez innych, czy też można osiągnąć jeszcze lepsze wyniki poprzez dalszą specjalizację. Jednak we wszystkich tych grach podstawowe algorytmy są wspólne i mogą znaleźć zastosowanie w każdej grze planszowej. W tym rozdziale przyjrzymy się rodzinie algorytmów minimax, najpopularniejszym technikom AI w grach planszowych. Omówię również inną rodzinę algorytmów, która okazała się lepsza w wielu zastosowaniach: algorytmy sterowników testowych ze zwiększoną pamięcią (MTD). Zarówno minimax, jak i MTD to algorytmy przeszukiwania drzew: wymagają one specjalnej reprezentacji drzewa gry.

Algorytmy te są idealne do implementacji sztucznej inteligencji w grach planszowych, ale oba opierają się na pewnej wiedzy o grze. Algorytmy są zaprojektowane do wyszukiwania najlepszego ruchu do wykonania, ale komputer nie może wyczuć, co oznacza „najlepszy", należy mu to powiedzieć. W najprostszym przypadku może to być po prostu „najlepszy ruch wygrywa grę", co jest wszystkim, czego potrzebujemy w grze takiej jak kółko i krzyżyk, gdzie minimax lub MTD mogą przeszukiwać każdą możliwą sekwencję ruchów. Jednak w przypadku większości gier komputer nie ma wystarczającej mocy, aby przeszukiwać do końca gry, więc potrzebna jest pewna wiedza na temat stanów pośrednich: najlepsze ruchy prowadzą do najlepszych pozycji, więc jak określić, jak dobry jest stanowisko jest? Używamy „statycznej funkcji oceny". Jest to ten sam kompromis, który wprowadziłem jako „złotą zasadę sztucznej inteligencji", omawiając symboliczną sztuczną inteligencję w rozdziale 1, podrozdział 1.1: im więcej możemy przeprowadzić wyszukiwania, tym mniej potrzebujemy wiedzy i na odwrót. W przypadku dość skomplikowanych gier planszowych ilość wyszukiwań, które możemy wykonać, jest ograniczona przez dostępną moc komputera, lepsze algorytmy wyszukiwania niewiele nam dają. Tak więc jakość sztucznej inteligencji będzie bardziej zależeć od jakości funkcji oceny statycznej. Zostało to wprowadzone w części poświęconej algorytmowi minimax, ale opisane szczegółowo w dalszej części rozdziału, ze szczególnym uwzględnieniem metod uczenia głębokiego w zdobywaniu wiedzy o grach, które w ostatnim czasie zmieniły stan techniki. W ostatniej części tego rozdziału przyjrzymy się, dlaczego komercyjne turowe gry strategiczne są często zbyt złożone, by wykorzystać tę sztuczną inteligencję; wymagają innych technik z reszty tej książki. Jeśli nie interesuje Cię sztuczna inteligencja w grach planszowych, możesz spokojnie pominąć ten rozdział.

TEORIA GRY

Teoria gier to dyscyplina matematyczna zajmująca się badaniem abstrakcyjnych, wyidealizowanych gier. Ma tylko bardzo słabe zastosowanie do gier komputerowych czasu rzeczywistego, ale wywodzi się z niego terminologia używana w grach turowych. Ta sekcja wprowadzi wystarczająco dużo teorii gier, aby umożliwić Ci zrozumienie i wdrożenie turowej sztucznej inteligencji bez zagłębiania się w matematykę.

RODZAJE GIER

Teoria gier klasyfikuje gry według liczby graczy, rodzaju celu, jaki mają oni, oraz informacji, jakie każdy gracz posiada na temat gry.

Liczba graczy

Gry planszowe, które zainspirowały turowe algorytmy sztucznej inteligencji, prawie wszystkie mają dwóch graczy. Większość popularnych algorytmów jest więc ograniczona do dwóch graczy w swojej najbardziej podstawowej formie. Można je dostosować do użycia z większymi liczbami, ale rzadko można znaleźć opisy algorytmów dla czegokolwiek innego niż dwóch graczy. Ponadto większość optymalizacji dla tych algorytmów zakłada, że jest tylko dwóch graczy. Chociaż podstawowe algorytmy można dostosować, większość optymalizacji nie może być stosowana tak łatwo.

Warstwy, ruchy i zwroty

W teorii gier często nazywa się turę jednego gracza „tarczą" gry. Jedna runda wszystkich tur graczy nazywana jest „ruchem".

Wywodzi się to z szachów, gdzie jeden ruch polega na tym, że każdy gracz wykonuje jedną turę. Ponieważ większość turowej sztucznej inteligencji opiera się na programach do gry w szachy, słowo „ruch" jest często używane w tym kontekście.

Istnieje jednak wiele innych gier, w których turę każdego gracza traktuje się jako osobny ruch i jest to terminologia używana zwykle w turowych grach strategicznych. W tym rozdziale zamiennie używa się słów „obróć" i „ruch" i w ogóle nie używa słowa „sklejka".

Cel gry

W większości gier strategicznych celem jest wygrana. Jako gracz wygrywasz, jeśli wszyscy twoi przeciwnicy przegrają. Jest to znane jako gra o sumie zerowej: Twoja wygrana to przegrana przeciwnika. Jeśli zdobędziesz 1 punkt za wygraną, będzie to równoznaczne z uzyskaniem -1 za przegraną. Tak by nie było, bo na przykład w grze kasynowej, kiedy wszyscy moglibyście zakończyć noc gorszym.

W grze o sumie zerowej nie ma znaczenia, czy próbujesz wygrać, czy próbujesz sprawić, by przeciwnik przegrał; wynik jest taki sam. W przypadku gry o sumie niezerowej, w której wszyscy moglibyście wygrać lub wszyscy przegrać, wolałbyś skupić się na własnej wygranej, a nie na przegranej rywalu (chyba że jesteś bardzo samolubny). W przypadku gier z więcej niż dwoma graczami sytuacja jest bardziej skomplikowana. Nawet w grze o sumie zerowej najlepszą strategią nie zawsze jest spowodowanie, by każdy przeciwnik przegrał. Może lepiej zjednoczyć się z najsilniejszym przeciwnikiem, przynosząc korzyści słabszym przeciwnikom i mając nadzieję, że później ich wykończysz.

Informacja

W grach takich jak Szachy, Warcaby, Go i Reversi obaj gracze wiedzą wszystko o stanie gry. Wiedzą, jaki będzie wynik każdego ruchu i jakie będą opcje dla następnego ruchu. Wiedzą to wszystko od

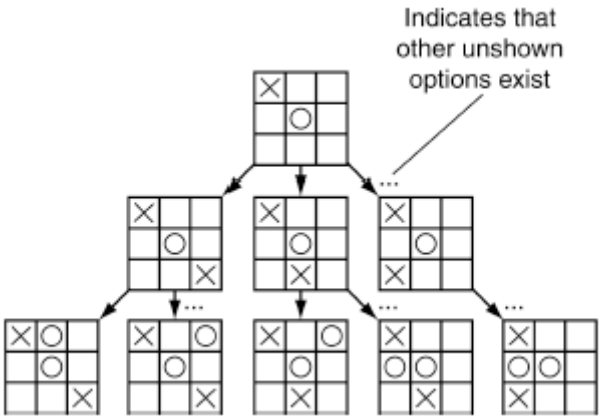
początku gry. Ten rodzaj gry nazywa się „doskonałą informacją”. Chociaż nie wiesz, jaki ruch zdecyduje się wykonać przeciwnik, masz pełną wiedzę na temat każdego ruchu, który może wykonać przeciwnik i jakie miałby on skutki. W grze takiej jak Backgammon Wydajnośćuje element losowy. Nie wiesz przed rzutem kostką, jakie ruchy będziesz mógł wykonać. Podobnie, nie możesz wiedzieć, jakie ruchy może wykonać twój przeciwnik, ponieważ nie możesz przewidzieć rzutu kośćmi przeciwnika. Ten rodzaj gry nazywa się „niedoskonałą informacją”. Większość turowych gier strategicznych zawiera niedoskonałe informacje; jest jakiś element losowy do wykonywania akcji (na przykład test umiejętności lub losowość w walce). Jednak doskonałe gry informacyjne są często łatwiejsze do przeanalizowania. Wiele algorytmów i technik turowej sztucznej inteligencji zakłada, że istnieją doskonałe informacje. Mogą być przystosowane do innych rodzajów gier, ale często w rezultacie działają gorzej.

Stosowanie algorytmów

Najbardziej znane i najbardziej zaawansowane algorytmy do gier turowych są przeznaczone do współpracy z dwuosobowymi, perfekcyjnymi grami informacyjnymi o sumie zerowej. Jeśli piszesz sztuczną inteligencję do grania w szachy, to jest to dokładnie taka implementacja, jakiej potrzebujesz. Jednak wiele turowych gier komputerowych jest bardziej skomplikowanych, angażujących więcej graczy i niedoskonałych informacji. Ten rozdział przedstawia algorytmy w ich najczęstszej postaci: dla dwóch graczy, doskonałych gier informacyjnych. Jak zobaczymy, będą musiały być przystosowane do innych rodzajów gier.

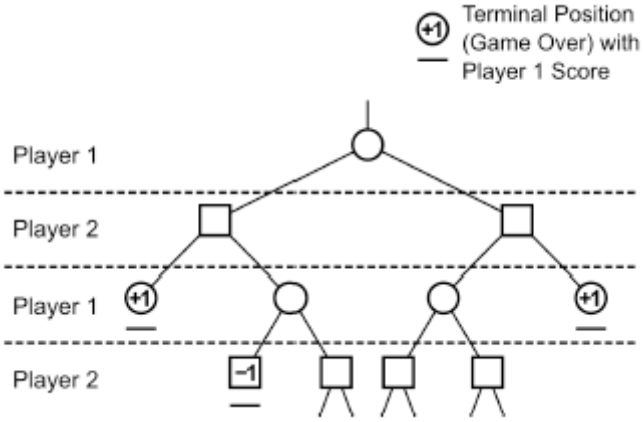
DRZEWO GRY

Każda gra turowa może być reprezentowana jako drzewo gry. Rysunek przedstawia część drzewa gry w kółko i krzyżyk.



Każdy węzeł w drzewie reprezentuje pozycję planszy, a każda gałąź reprezentuje jeden możliwy ruch. Przesuwa prowadzenie z jednej pozycji na planszy do drugiej. Każdy gracz może poruszać się po różnych poziomach drzewa. Ponieważ gra jest turowa, plansza zmienia się tylko wtedy, gdy jeden z graczy wykona ruch. Liczba gałęzi z każdej planszy jest równa liczbie możliwych ruchów, które gracz może wykonać. W Kółko i krzyżyk ta liczba to dziewięć w turze pierwszego gracza, potem osiem i tak dalej. W wielu grach mogą istnieć setki, a nawet tysiące możliwych ruchów, które każdy gracz może wykonać. Niektóre pozycje na planszy nie mają żadnych możliwych ruchów. Są to tak zwane pozycje końcowe i reprezentują koniec gry. Za każdą pozycję końcową każdy gracz otrzymuje wynik końcowy. Może to być tak proste, jak +1 za wygraną i -1 za przegraną lub może odzwierciedlać wielkość wygranej. Dozwolone są również remisy z wynikiem 0. W grze o sumie zerowej końcowe wyniki każdego gracza sumują się do zera. W grze o sumie niezerowej wyniki będą odzwierciedlać wielkość osobistej wygranej lub przegranej każdego gracza. W tym miejscu podaję tylko wyniki końcowe, dla pozycji po zakończeniu

gry. Do pomysłu przypisywania wyników do pośrednich pozycji na planszy powrócę później. Najczęściej drzewo gry jest przedstawiane w formie abstrakcyjnej bez diagramów planszowych, zamiast tego pokazuje końcowe wyniki. Rysunek zakłada, że gra ma sumę zerową, więc pokazuje tylko wyniki dla pierwszego gracza.

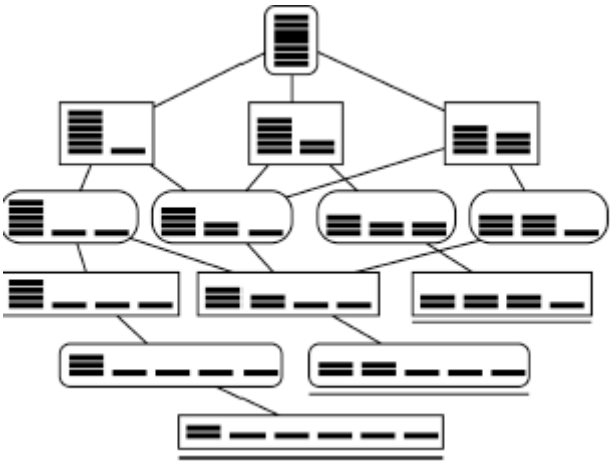


Współczynnik rozgałęzienia i głębokość

Liczba rozgałęzień w każdym punkcie rozgałęzienia drzewa nazywana jest współczynnikiem rozgałęzienia i jest dobrym wskaźnikiem tego, jak trudno komputerowi będzie grać w grę. Różne gry mają również różną głębokość drzewa: różną maksymalną liczbę tur. W grze w kółko i krzyżyk każdy gracz na zmianę dodaje swój symbol do planszy. Na planszy jest dziewięć pól, więc tur jest maksymalnie dziewięć. To samo dzieje się w Reversi, rozgrywanym na planszy osiem na osiem. W Reversi na początku gry na planszy znajdują się cztery pionki, więc maksymalnie może być 60 tur (z wyłączeniem podań, które w Reversi nigdy nie są dobrowolne). Gry takie jak Szachy mogą mieć prawie nieskończoną liczbę tur (ogranicza to zasada 50 ruchów w szachach turniejowych). Drzewo gry dla takiej gry byłoby niezwykle głębokie, nawet jeśli współczynnik rozgałęzienia byłby stosunkowo niewielki. Graj w szachy krasnoludów, z ponad 1048 możliwymi ruchami w grze i maksymalnym współczynnikiem rozgałęzienia 19 x 19 = 361. Komputerom łatwiej jest grać w gry z małym współczynnikiem rozgałęzienia i głębokim drzewem niż w gry z płytkim drzewem, ale ogromnym współczynnik rozgałęzienia. Idealna gra z możliwością wyszukiwania miałaby mały współczynnik rozgałęzienia i płytką głębokość (taką jak kółko i krzyżyk), chociaż takie gry są często nudne dla ludzkich graczy. Najciekawsze gry często mają bardzo duże wartości dla obu.

Transpozycja

W wielu grach możliwe jest kilkakrotne zajęcie tej samej pozycji na planszy w trakcie gry. W wielu innych grach możliwe jest osiągnięcie tej samej pozycji przez różne kombinacje ruchów. Posiadanie tej samej pozycji na planszy z różnych sekwencji ruchów nazywa się transpozycją. Oznacza to, że w większości gier drzewo gry wcale nie jest drzewem; gałęzie mogą się łączyć lub dzielić. Split-Nim, odmiana chińskiej gry Nim, zaczyna się od jednego stosu monet. W każdej turze, zmieniający się gracze muszą podzielić jeden stos monet na dwa nierówne stosy. Wygrywa ostatni gracz, który będzie mógł wykonać ruch. Rysunek przedstawia pełne drzewo gry dla gry 7-Split-Nim (zaczynając od 7 monet w stosie).



Widać, że istnieje duża liczba różnych gałęzi łączących się. Algorytmy oparte na Minimax (te, które przyjrzymy się w następnej sekcji) są zaprojektowane do pracy z czystymi drzewami. Mogą pracować ze scalonymi gałęziami, ale powielają swoją pracę dla każdej scalanej gałęzi. Muszą być rozszerzone o tabele przejść, aby uniknąć powielania pracy podczas łączenia gałęzi. Drugi zestaw kluczowych algorytmów w tym rozdziale, MTD, został zaprojektowany z myślą o transpozycji.

MINIMAKSOWANIE

Komputer gra w grę turową, patrząc na akcje dostępne w tym ruchu i wybierając jedną z nich. Aby wybrać jeden z ruchów, musi wiedzieć, które ruchy są lepsze od innych. Ta wiedza jest przekazywana komputerowi przez programistę za pomocą heurystyki zwanej funkcją oceny statycznej.

FUNKCJA OCENY STATYCZNEJ

W grze turowej zadaniem funkcji oceny statycznej jest spojrzenie na aktualny stan planszy i punktowanie go z punktu widzenia jednego gracza. Jeśli plansza jest pozycją końcową w drzewie, ten wynik będzie ostatecznym wynikiem gry. Więc jeśli na szachownicy widać mata do czarnego, to jego wynik będzie +1 do czarnego (lub jakakolwiek zwycięski wynik ma być), podczas gdy wynik białego będzie -1. Łatwo jest zdobyć zwycięską pozycję: jedna strona będzie miała najwyższą możliwą liczbę punktów, a druga najniższą możliwą liczbę punktów. W środku gry znacznie trudniej jest zdobyć punkty. Wynik powinien odzwierciedlać prawdopodobieństwo, że gracz wygra grę z tej pozycji na planszy. Jeśli więc plansza pokazuje przytłaczającą przewagę jednemu graczowi, to ten gracz powinien otrzymać wynik bardzo zbliżony do zwycięskiego wyniku. W większości przypadków bilans wygranej lub przegranej może nie być jasny.

Wartość punktowa

W zasadzie funkcja oceny może zwrócić dowolną liczbę o dowolnym rozmiarze. W większości implementacji zwraca jednak liczbę całkowitą ze znakiem. Kilka najpopularniejszych algorytmów w tym rozdziale opiera się na funkcji oceny będącej liczbą całkowitą. Ponadto arytmetyka na liczbach całkowitych jest szybsza niż arytmetyka zmiennoprzecinkowa na większości maszyn.

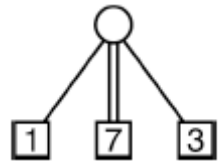
Zakres możliwych wartości nie jest zbyt ważny. Zakres zwróconych punktów powinien być mniejszy niż punktów za wygraną lub przegraną. Jeśli funkcja oceny statycznej zwraca +1000 dla pozycji, która jest bardzo bliska wygranej, ale tylko +100 dla wygranej, sztuczna inteligencja spróbuje nie wygrać gry, ponieważ bycie blisko wydaje się o wiele bardziej atrakcyjne.

Prosty wybór ruchu

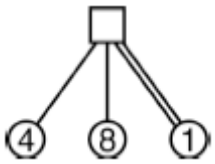
Dzięki dobrej funkcji oceny statycznej komputer może wybrać ruch, oceniając pozycje, które wynikną po wykonaniu każdego możliwego ruchu, a następnie wybierając ruch, który prowadzi do najwyższego wyniku. Rysunek 9.4 pokazuje możliwe ruchy gracza, oceniane za pomocą funkcji oceny. Oczywiste jest, że wykonanie drugiego ruchu da najlepszą pozycję na planszy, więc jest to ruch, który należy wybrać. Biorąc pod uwagę doskonałą funkcję oceny, to wszystko, co musiałaby zrobić sztuczna inteligencja: spojrzeć na wynik każdego możliwego ruchu i wybrać najwyższy wynik. Niestety idealna funkcja oceny to fantazja. Funkcje oceny, które zobaczymy na końcu tego rozdziału, działają rozsądnie, gdy są używane w ten sposób, ale nadal łatwo je pokonać w wyszukiwaniu. Komputer przesuwa się w dół drzewa: patrząc na możliwe odpowiedzi innego gracza, odpowiedzi na te odpowiedzi i tak dalej. Jest to ten sam proces, który wykonują ludzcy gracze, gdy patrzą w przód na jeden lub więcej ruchów. W przeciwieństwie do ludzkich graczy, którzy polegają na intuicyjnym wyczuciu, kto wygrywa, heurystyka komputerowa była zazwyczaj dość wąska, ograniczona i słaba. W takich przypadkach komputer musi wyprzedzać o wiele więcej ruchów, niż może to zrobić człowiek. Najbardziej znanym algorytmem wyszukiwania gier jest minimax. W różnych formach dominował w turowej sztucznej inteligencji do połowy lat 90., a dla niektórych gier nadal jest najlepszym wyborem.

MINIMAKSOWANIE

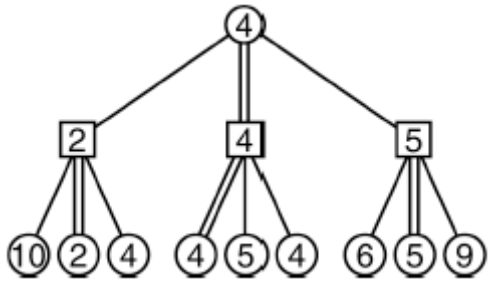
Jeśli wybierzemy ruch, prawdopodobnie wybierzemy ruch, który zapewni dobrą pozycję. Możemy założyć, że wybierzemy ruch, który prowadzi do najlepszej dostępnej nam pozycji. Innymi słowy, w naszych ruchach staramy się zmaksymalizować nasz wynik.



Kiedy jednak nasz przeciwnik się porusza, zakładamy, że wybierze ruch, który pozostawi nas w najgorszej dostępnej pozycji. Nasz przeciwnik stara się zminimalizować nasz wynik.



Kiedy szukamy odpowiedzi przeciwnika na nasze odpowiedzi, musimy pamiętać, że maksymalizujemy nasz wynik, podczas gdy nasz przeciwnik go minimalizuje. Ta zmiana między maksymalizacją a minimalizacją, gdy przeszukujemy drzewo gry, nazywa się minimaksowaniem. Drzewa gry na rysunkach mają głębokość tylko jednego ruchu. Aby ustalić, jaki jest nasz najlepszy możliwy ruch, musimy również wziąć pod uwagę reakcje przeciwnika. Na rysunku wyniki dla każdej pozycji na planszy są pokazane po dwóch ruchach.



Jeśli wykonamy pierwszy ruch, jesteśmy w sytuacji, w której możemy skończyć z wynikiem 10 punktów na stole, ale musimy założyć, że nasz przeciwnik nam na to nie pozwoli i wykona ruch, który pozostawia nam 2. Tak więc wynik pierwszego ruchu dla nas to 2; to wszystko, czego możemy się spodziewać, jeśli wykonamy ten ruch. Z drugiej strony, gdybyśmy wykonali ruch drugi, nie mielibyśmy nadziei na zdobycie 10. Ale niezależnie od tego, co zrobi nasz przeciwnik, skończylibyśmy z co najmniej 4. Tak więc możemy spodziewać się 4 z ruchu drugiego. Dlatego ruch drugi jest lepszy niż ruch pierwszy, ale nadal nie jest tak dobry jak ruch trzeci, który jest naszą najlepszą opcją. Zaczynając od dołu, wyniki są bąbelkowane zgodnie z zasadą minimaksów: w naszej turze bąbelkujemy najwyższy wynik; w turze naszego przeciwnika zdobywamy najniższy wynik. W końcu mamy dokładne wyniki dla wyników każdego dostępnego ruchu i po prostu wybieramy najlepszy z nich. Ten proces propagacji w górę drzewa jest tym, co robi algorytm minimaksowania. Aby określić, jak dobry jest ruch, szuka odpowiedzi i odpowiedzi na te odpowiedzi, aż nie może dalej szukać. W tym momencie opiera się na statycznej funkcji oceny. Następnie zbiera te wyniki z powrotem, aby uzyskać wynik za każdy z dostępnych ruchów. Nawet w przypadku wyszukiwań, które przewidują tylko kilka ruchów, minimaksowanie zapewnia znacznie lepsze wyniki niż po prostu opierając się wyłącznie na heurystyce.

ALGORYTM MINIMAKSOWANIA

Algorytm minimax jest rekurencyjny. Przy każdej rekursji próbuje obliczyć poprawną pośrednią wartość punktową aktualnej pozycji na planszy. Robi to, patrząc na każdy możliwy ruch z aktualnej pozycji na planszy. Dla każdego ruchu oblicza wynikową pozycję na planszy i powtarza się, aby znaleźć wartość tej pozycji. Aby wyszukiwanie nie trwało w nieskończoność (w przypadku, gdy drzewo jest bardzo głębokie), algorytm ma maksymalną głębokość wyszukiwania. Jeśli bieżąca pozycja planszy jest na maksymalnej głębokości, wywołuje funkcję oceny statycznej i zwraca wynik. Jeśli algorytm bierze pod uwagę pozycję, na której ma się poruszyć aktualny gracz, zwraca najwyższą wartość, jaką widział; w przeciwnym razie zwraca najniższy. Powoduje to zmianę pomiędzy krokami minimalizacji i maksymalizacji, nadając algorytmowi jego nazwę. Jeśli głębokość wyszukiwania wynosi zero, przechowuje również najlepszy znaleziony ruch. To będzie krok do zrobienia.

Pseudo kod

Algorytm minimax możemy zaimplementować w następujący sposób:

```
1 function minimax(board: Board,

2 player: id,

3 maxDepth: int,

4 currentDepth: int) -> (float, Move):

5 # Check if we're done recursing.

6 if board.isGameOver() or currentDepth == maxDepth:
```

```
7 return board.evaluate(player), null
8
9 # Otherwise bubble up values from below.
10 bestMove: Move = null
11 if board.currentPlayer() == player:
12 bestScore: float = -INFINITY
13 else:
14 bestScore: float = INFINITY
15
16 # Go through each move.
17 for move in board.getMoves():
18 newBoard: Board = board.makeMove(move)
19
20 # Recurse.
21 currentScore, currentMove = minimax(
22 newBoard, player, maxDepth, currentDepth+1)
23
24 # Update the best score.
25 if board.currentPlayer() == player:
26 if currentScore > bestScore:
27 bestScore = currentScore
28 bestMove = move
29 else:
30 if currentScore < bestScore:
31 bestScore = currentScore
32 bestMove = move
33
34 # Return the score and the best move.
35 return bestScore, bestMove
```

W tym kodzie założyłem, że funkcja minimax może zwrócić dwie rzeczy: najlepszy ruch i jego wynik. W przypadku języków, które mogą zwrócić tylko jeden element, przeniesienie można przekazać z powrotem za pomocą wskaźnika lub zwracając strukturę. Stała INFINITY powinna być większa niż

wszystko, co zwraca funkcja board.evaluate. Służy do upewnienia się, że zawsze znajdzie się najlepszy ruch, bez względu na to, jak słaby może być.

Funkcja minimax może być wyprowadzona z prostszej funkcji, która po prostu zwraca najlepszy ruch:

```
1 function getBestMove(board: Board, player: id, maxDepth: int) -> Move:

2 # Get the result of a minimax run and return the move.

3 score, move = minimax(board, player, maxDepth, 0)

4 return move
```

Struktury danych i interfejsy

Powyższy kod powoduje, że tablica wykonuje pracę polegającą na obliczaniu dopuszczalnych ruchów i ich stosowaniu. Instancja klasy Board reprezentuje jedną pozycję w grze. Klasa powinna mieć następującą formę:

```
1 class Board:

2 function getMoves() -> Move[]

3 function makeMove(move: Move) -> Board

4 function evaluate(player: id) -> float

5 function currentPlayer() -> id

6 function isGameOver() -> bool
```

gdzie getMoves zwraca listę obiektów ruchu (może mieć dowolny format, nie jest to istotne dla algorytmu), które odpowiadają ruchom, które można wykonać z pozycji planszy. Metoda makeMove pobiera jedną instancję ruchu i zwraca całkowicie nowy obiekt tablicy, który reprezentuje pozycję po wykonaniu ruchu. oceniać jest funkcją oceny statycznej. Zwraca wynik dla aktualnej pozycji z punktu widzenia danego gracza. Obecny Gracz zwraca gracza, którego tura ma zagrać na bieżącej planszy. Może się to różnić od gracza, którego najlepszy ruch staramy się wypracować. Wreszcie, isGameOver zwraca wartość true, jeśli pozycja planszy jest końcowa. Ta struktura dotyczy wszystkich doskonałych gier informacyjnych dla dwóch graczy, od gry w kółko i krzyżyk po szachy.

Więcej niż dwóch graczy

Możemy rozszerzyć ten sam algorytm, aby obsłużyć trzech lub więcej graczy. Zamiast naprzemiennej minimalizacji i maksymalizacji wykonujemy minimalizację w dowolnym ruchu, gdy nie jesteśmy graczem, i maksymalizację w naszym ruchu. Powyższy kod obsługuje to normalnie. Jeśli jest trzech graczy, to:

```
board.currentPlayer() == gracz
```

będzie prawdziwe jeden krok na trzy, więc otrzymamy jeden krok maksymalizacji, po którym następują dwa kroki minimalizacji.

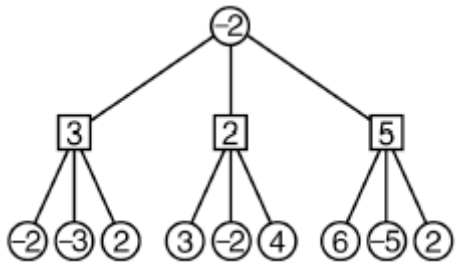
Wydajność

Algorytmem jest O(d) w pamięci, gdzie d jest maksymalną głębokością wyszukiwania (lub maksymalną głębokością drzewa, jeśli jest mniejsza). Jest O(nd) w czasie, gdzie n to liczba możliwych ruchów na

każdej pozycji planszy. Przy szerokim i głębokim drzewie może to być niewiarygodnie nieefektywne. W dalszej części tej sekcji przyjrzymy się sposobom optymalizacji jego wydajności.

NEGAMAXING

Procedura minimax konsekwentnie ocenia ruchy w oparciu o punkt widzenia jednego gracza. Obejmuje specjalny kod do śledzenia, czyj to ruch i czy w związku z tym wyniki powinny być maksymalizowane, czy minimalizowane, aby wzrosnąć. W przypadku niektórych gier ta elastyczność jest potrzebna, ale w niektórych przypadkach możemy to poprawić. W przypadku gier dla dwóch graczy o sumie zerowej wiemy, że zysk jednego gracza jest stratą drugiego gracza. Jeśli jeden gracz zdobędzie punkt za planszę na poziomie -1, przeciwnik powinien otrzymać punktację na +1. Możemy wykorzystać ten fakt do uproszczenia algorytmu minimax. Na każdym etapie bulgotania, zamiast wybierania najmniejszego lub największego, wszystkie wyniki z poprzedniego poziomu mają zmienione znaki. Wyniki są wtedy poprawne dla gracza w tym ruchu (tj. nie reprezentują już prawidłowych wyników dla gracza wykonującego wyszukiwanie). Ponieważ każdy gracz będzie starał się zmaksymalizować swój wynik, za każdym razem można wybrać największą z tych wartości. Ponieważ przy każdym bąblowaniu odwracamy wyniki i wybieramy maksimum, algorytm jest znany jako „negamax”. Daje takie same wyniki jak algorytm minimax, ale każdy poziom bąbelkowania jest identyczny. Nie ma potrzeby, aby śledzić czyj to ruch i zachowywać się inaczej. Rysunek pokazuje bulgotanie na każdym poziomie w drzewie gry. Zauważ, że na każdym etapie wartość odwróconych wyników jest największa na kolejnym niższym poziomie.



Negamax i funkcja oceny

Funkcja oceny statycznej ocenia planszę zgodnie z punktem widzenia jednego gracza. Na każdym poziomie podstawowego algorytmu minimaksowego do obliczania wyników używany jest ten sam punkt widzenia. Aby to zrealizować, funkcja punktacji musi zaakceptować gracza, którego punkt widzenia należy wziąć pod uwagę. Ponieważ negamax zmienia punkty widzenia między graczami w każdej turze, funkcja oceny zawsze musi punktować z punktu widzenia gracza, którego ruch jest na tej planszy. Tak więc punkt widzenia zmienia się pomiędzy graczami przy każdym ruchu. Aby to zrealizować, funkcja oceny nie musi już akceptować punktu widzenia jako danych wejściowych. Może po prostu spojrzeć na czyja kolej zagrać.

Pseudo kod

Zmodyfikowany algorytm negamaxingu wygląda następująco

```
1 function negamax(board: Board,
2 maxDepth: int,
3 currentDepth: int) -> (float, Move):
```

```
4 # Check if we’re done recursing.
5 if board.isGameOver() or currentDepth == maxDepth:
6 return board.evaluate(), null
7
8 # Otherwise bubble up values from below.
9 bestMove: Move = null
10 bestScore: float = -INFINITY
11
12 # Go through each move.
13 for move in board.getMoves():
14 newBoard: Board = board.makeMove(move)
15
16 # Recurse.
17 recursedScore, currentMove = negamax(
18 newBoard, maxDepth, currentDepth + 1)
19 currentScore = -recursedScore
20
21 # Update the best score.
22 if currentScore > bestScore:
23 bestScore = currentScore
24 bestMove = move
25
26 # Return the score and the best move.
27 return bestScore, bestMove
```

Zauważ, że ponieważ nie musimy już przekazywać go do metody oceniania, w ogóle nie potrzebujemy parametru odtwarzacza.

Struktury danych i interfejsy

Ponieważ nie musimy przekazywać gracza do metody Board.evaluate, interfejs Board wygląda teraz następująco

Wydajność

Algorytm negamax jest identyczny z algorytmem minimax dla charakterystyki wydajności. Jest to również O(d) w pamięci, gdzie d to maksymalna głębokość poszukiwań, a O(nd) w czasie, gdzie n to

liczba ruchów na każdej pozycji planszy. Pomimo tego, że jest prostszy do wdrożenia i szybszy do wykonania, skaluje się w ten sam sposób z dużymi drzewami gier.

Uwagi dotyczące implementacji

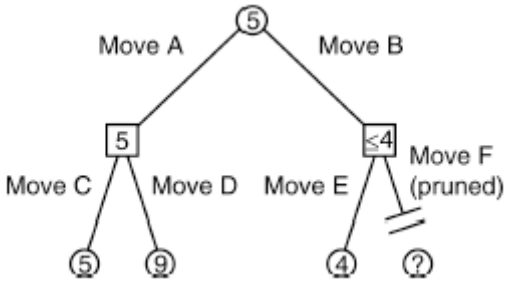
Większość optymalizacji, które można zastosować do negamaxingu, może działać przy ścisłym podejściu do minimaksowania. Optymalizacje w tym rozdziale zostaną wprowadzone pod kątem negamaxu, ponieważ jest on znacznie szerzej stosowany w praktyce. Kiedy programiści mówią o minimaxingu, często używają w praktyce algorytmu opartego na negamaxie. Minimax jest często używany jako ogólny termin obejmujący całą gamę optymalizacji. W szczególności, jeśli przeczytasz „minimaks" w książce opisującej sztuczną inteligencję w grach, najprawdopodobniej będzie to odnosić się do optymalizacji negamaksowej zwanej „alfa-beta (AB) negamax". Przyjrzymy się następnie optymalizacji AB.

PRZYCINANIE AB

Algorytm negamaxingu jest wydajny, ale sprawdza więcej pozycji na planszy niż to konieczne. Przycinanie AB pozwala algorytmowi zignorować sekcje drzewa, które nie mogą zawierać najlepszego ruchu. Składa się z dwóch rodzajów przycinania: alfa i beta.

Przycinanie alfa

Rysunek pokazuje drzewo gry, zanim jakiekolwiek bulgotanie zostanie zakończone. Aby łatwiej zobaczyć, w jaki sposób wyniki są przetwarzane, na tej ilustracji użyję algorytmu minimax.



Proces bulgotania rozpoczynamy w taki sam sposób, jak poprzednio. Jeśli gracz pierwszy wykona ruch A, to jego przeciwnik odpowie ruchem C, dając graczowi wynik 5. W ten sposób zdobędziemy 5. Teraz algorytm patrzy na ruch B. Widzi, że pierwszą odpowiedzią na B jest E, co punkty 4. Nie ma znaczenia, jaka jest teraz wartość F, ponieważ przeciwnik zawsze może wymusić wartość 4. Nawet nie biorąc pod uwagę F, gracz wie, że wykonanie ruchu B jest złe; mogą otrzymać 5 z ruchu A i maksymalnie 4 z ruchu B, być może nawet mniej. Aby przycinać w ten sposób, musimy śledzić najlepszy wynik, jaki możemy osiągnąć. W rzeczywistości ta wartość stanowi dolny limit wyniku, jaki możemy osiągnąć. W dalszej części wyszukiwania możemy znaleźć lepszą sekwencję ruchów, ale nigdy nie zaakceptujemy sekwencji ruchów, która daje nam niższy wynik. Ta dolna granica nazywana jest wartością alfa (czasami, ale rzadko, pisana jako grecka litera α), a przycinanie nazywa się przycinaniem alfa. Śledząc wartość alfa, możemy uniknąć rozważania jakiegokolwiek ruchu, w którym przeciwnik ma możliwość pogorszenia go. Nie musimy się martwić o to, o ile gorzej może to pogorszyć przeciwnik; już wiemy, że nie damy im takiej możliwości.

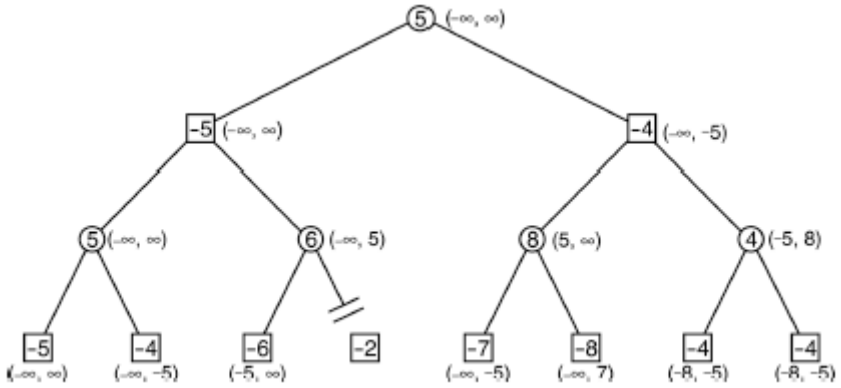
Przycinanie beta

Przycinanie beta działa w ten sam sposób. Wartość beta (ponownie, rzadko pisana β) śledzi górny limit tego, co możemy liczyć. Aktualizujemy wartość beta, gdy znajdziemy sekwencję ruchów, do których przeciwnik może nas zmusić. W tym momencie wiemy, że nie ma sposobu, aby zdobyć więcej punktów

niż wartość beta, ale może być jeszcze więcej sekwencji do odkrycia, które przeciwnik może wykorzystać do jeszcze większego ograniczenia nas. Jeśli znajdziemy sekwencję ruchów, która ma wyższą punktację niż wartość beta, możemy ją zignorować, ponieważ wiemy, że nigdy nie będziemy mieli możliwości ich wykonania. Razem wartości alfa i beta zapewniają okno możliwych wyników. Nigdy nie zdecydujemy się na ruchy, które dają wynik niższy niż alfa, a nasz przeciwnik nigdy nie pozwoli nam wykonywać ruchów, które mają wynik większy niż beta. Wynik, jaki w końcu osiągamy, musi leżeć między nimi. Podczas przeszukiwania drzewa wartości alfa i beta są aktualizowane. Jeśli zostanie znaleziona gałąź drzewa, która znajduje się poza tymi wartościami, można ją przyciąć. Z powodu naprzemiennego minimalizowania i maksymalizacji dla każdego gracza, tylko jedna wartość musi być sprawdzona na każdej pozycji na planszy. Na pozycji na planszy, w której jest kolej przeciwnika na grę, minimalizujemy wyniki, więc tylko minimalny wynik może się zmienić i musimy tylko sprawdzić przed alfą. Jeśli jest nasza kolej na grę, maksymalizujemy wyniki, dlatego wymagany jest tylko test beta.

AB Negamax

Chociaż łatwiej jest dostrzec różnicę między śliwkami alfa i beta w algorytmie minimax, są one najczęściej używane w przypadku negamax. Zamiast naprzemiennego sprawdzania alfa i beta w każdym kolejnym ruchu, AB negamax zamienia i odwraca wartości alfa i beta (w ten sam sposób, w jaki odwraca wyniki z następnego poziomu). Sprawdza i przycina tylko wartość beta. Używając przycinania AB z negamaxing, mamy najprostszy praktyczny algorytm AI gier planszowych. Stanowi on podstawę wszystkich dalszych optymalizacji w tej sekcji. Rysunek przedstawia parametry alfa i beta przekazywane do algorytmu negamax w każdym węźle drzewa gry oraz wynik, jaki generuje algorytm.



Widać, że gdy algorytm przeszukuje drzewo od lewej do prawej, wartości alfa i beta zbliżają się do siebie, ograniczając wyszukiwanie. Możesz także zobaczyć, w jaki sposób wartości alfa i beta zmieniają znaki i zamieniają się miejscami na każdym poziomie drzewa.

Pseudo kod

Algorytm AB negamax ma następującą strukturę:

```
1 function abNegamax(board: Board,
2 maxDepth: int,
3 currentDepth: int,
4 alpha: float,
5 beta: float) -> (float, Move):
6 # Check if we're done recursing.
```

```
7 if board.isGameOver() or currentDepth == maxDepth:
8 return board.evaluate(player), null
9
10 # Otherwise bubble up values from below.
11 bestMove: Move = null
12 bestScore: float = -INFINITY
13
14 # Go through each move.
15 for move in board.getMoves():
16 newBoard: Board = board.makeMove(move)
17
18 # Recurse.
19 recursedScore, currentMove = abNegamax(
20 newBoard, maxDepth, currentDepth + 1,
21 -beta, -max(alpha, bestScore))
22 currentScore = -recursedScore
23
24 # Update the best score.
25 if currentScore > bestScore:
26 bestScore = currentScore
27 bestMove = move
28
29 # If we're outside the bounds, prune by exiting immediately.
30 if bestScore >= beta:
31 break
32
33 return bestScore, bestMove
```

Może to wynikać z funkcji formularza:

```
1 function getBestMove(board: Board, maxDepth: int) -> Move:
2 # Get the result of a minimax run and return the move.
3 score, move = abNegamax(board, maxDepth, 0, -INFINITY, INFINITY)
```

4 return move

Struktury danych i interfejsy

Ta implementacja opiera się na tej samej klasie planszy, co w przypadku zwykłego negamaxa.

Wydajność

Po raz kolejny algorytm to O(d) w pamięci, gdzie d to maksymalna głębokość przeszukiwania, a kolejność O(nd) w czasie, gdzie n to liczba możliwych ruchów na każdej pozycji planszy. Po co więc optymalizacja, skoro uzyskujemy taką samą wydajność? Kolejność wyników może być taka sama, ale negamax AB przewyższa normalny negamax w prawie wszystkich przypadkach. Jedyna sytuacja, w której tak się nie stanie, to taka kolejność ruchów, aby przycinanie nie było możliwe. W takim przypadku algorytm będzie miał dodatkowe porównanie, które nigdy nie jest prawdziwe i dlatego będzie wolniejsze. Taka sytuacja byłaby prawdopodobna tylko wtedy, gdyby posunięcia zostały nakazane celowo w celu jej wykorzystania. W zdecydowanej większości przypadków wydajność jest znacznie lepsza od podstawowego algorytmu.

OKNO WYSZUKIWANIA AB

Odstęp między wartościami alfa i beta w algorytmie AB nazywa się oknem wyszukiwania. Uwzględniane są tylko nowe sekwencje ruchów z wynikami w tym oknie. Wszystkie inne są przycinane. Im mniejsze okno wyszukiwania, tym większe prawdopodobieństwo, że gałąź zostanie przycięta. Początkowo AB algorytmy są wywoływane z nieskończenie dużym oknem wyszukiwania: (-∞;+ ∞). Gdy działają, okno wyszukiwania jest skrócone. Wszystko, co może skrócić okno wyszukiwania tak szybko, jak to możliwe, zwiększy liczbę suszonych śliwek i przyspieszy działanie algorytmu.

Przenieś zamówienie

Jeśli najbardziej prawdopodobne ruchy zostaną rozważone jako pierwsze, okno wyszukiwania skurczy się szybciej. Mniej prawdopodobne ruchy zostaną rozważone później i są bardziej prawdopodobne, że zostaną przycięte. Ustalenie, które ruchy są lepsze, to oczywiście cały sens AI. Gdybyśmy znali najlepsze ruchy, nie musielibyśmy uruchamiać algorytmu. Istnieje więc kompromis między możliwością wykonywania mniej poszukiwań (poprzez poznanie z góry, które ruchy są najlepsze) a koniecznością posiadania mniejszej wiedzy (i koniecznością wyszukiwania więcej). W najprostszym przypadku możliwe jest użycie funkcji oceny statycznej na ruchach w celu ustalenia właściwej kolejności. Ponieważ funkcja oceny daje przybliżone wskazanie, jak dobra jest pozycja planszy, może skutecznie zmniejszyć rozmiar wyszukiwania podczas przycinania AB. Często jednak zdarza się, że wielokrotne wywoływanie w ten sposób funkcji oceny spowalnia działanie algorytmu. Jednak jeszcze skuteczniejszą techniką porządkowania jest wykorzystanie wyników poprzednich wyszukiwań mini-max. Mogą to być wyniki poszukiwań na poprzednich głębokościach przy użyciu iteracyjnego algorytmu pogłębiania lub mogą to być wyniki poszukiwań minimaksowych na poprzednich zakrętach. Rodzina algorytmów testowych ze zwiększoną pamięcią wyraźnie wykorzystuje to podejście do porządkowania ruchów przed ich rozważeniem. Pewna forma kolejności ruchów może być również dodana do dowolnego algorytmu AB minimax.

Nawet bez jakiejkolwiek formy kolejności ruchów, wydajność algorytmu AB może być 10 razy lepsza niż samego minimaxu. Dzięki doskonałemu porządkowaniu ruchów może być ponownie ponad 10 razy szybszy, czyli 100 razy szybszy niż zwykły minimax. To często różnica między przeszukaniem drzewa a kilkoma dodatkowymi zakrętami.

Wyszukiwanie aspiracji

Posiadanie małego okna wyszukiwania jest tak ogromnym przyspieszeniem, że warto sztucznie ograniczyć okno. Zamiast wywoływać algorytm z przedziałem (-∞,+∞), można go wywołać z przedziałem oszacowanym. Ten zakres nazywa się aspiracją, a algorytm AB nazywany w ten sposób jest czasami nazywany wyszukiwaniem aspiracji. Ten mniejszy zakres spowoduje przycięcie znacznie większej liczby gałęzi, przyspieszając działanie algorytmu. Z drugiej strony, w danym zakresie wartości może nie być odpowiednich sekwencji ruchów. W takim przypadku algorytm powróci z niepowodzeniem: nie zostanie znaleziony najlepszy ruch. Wyszukiwanie można następnie powtórzyć z szerszym oknem. Dążenie do wyszukiwania często opiera się na wynikach poprzedniego wyszukiwania. Jeśli podczas poprzedniego wyszukiwania plansza uzyskała 5 punktów, to gdy gracz znajdzie się na tej planszy, wykona wyszukiwanie według aspiracji za pomocą (5 − rozmiar okna, 5 + rozmiar okna). Rozmiar okna zależy od zakresu wyników, które mogą być zwrócone przez funkcję oceny. Prosta funkcja kierowcy, która może przeprowadzić wyszukiwanie aspiracji, wyglądałaby następująco:

```
1 function aspiration(board: Board, maxDepth: int, prev: float) -> Move:
2 alpha = prev - WINDOW_SIZE
3 beta = prev + WINDOW_SIZE
4
5 while true:
6 result, move = abNegamax(board, maxDepth, 0, alpha, beta)
7 if result <= alpha:
8 alpha = -NEAR_INFINITY
9 else if result >= beta:
10 beta = NEAR_INFINITY
11 else:
12 return move
```

NEGASCOUT

Zawężenie okna wyszukiwania można posunąć do skrajności, mając okno wyszukiwania o zerowej szerokości. To wyszukiwanie przytnie prawie wszystkie gałęzie drzewa, dzięki czemu wyszukiwanie jest bardzo szybkie. Niestety, przytnie wszystkie przydatne gałęzie, a także te bezużyteczne. Więc jeśli nie uruchomisz algorytmu z poprawnym wynikiem, zakończy się niepowodzeniem. Jako test można uznać zerowy rozmiar okna. Sprawdza, czy rzeczywisty wynik jest równy domysłowi. Nic dziwnego, że w tej formie nazywa się to „Testem”. Rozważana do tej pory wersja AB negamax jest czasami nazywana wersją „fail-soft”. Jeśli się nie powiedzie, zwraca najlepszy wynik, jaki miał do tej pory. Najbardziej podstawowa wersja AB negamax zwróci wynik alfa lub beta tylko w przypadku niepowodzenia (w zależności od tego, czy zawodzi wysoki lub zawodzi niski). Dodatkowe informacje w wersji soft-fail mogą pomóc w znalezieniu rozwiązania. Pozwala nam to przenieść nasze początkowe przypuszczenie i powtórzyć wyszukiwanie z bardziej rozsądnym oknem. Bez fail-soft nie miałbyś pojęcia, jak daleko posunąć swoje przypuszczenie. Oryginalny algorytm scout łączył wyszukiwanie minimaksowe (z przycinaniem AB) z wywołaniami testu zerowej szerokości. Ponieważ opiera się na wyszukiwaniu minimax, nie jest powszechnie używany. Algorytm negascout wykorzystuje algorytm AB negamax do przeprowadzenia testu. Negascout wykonuje pełne badanie pierwszego ruchu z każdej pozycji na

planszy. Odbywa się to za pomocą szerokiego okna wyszukiwania, aby algorytm nie zawodził. Kolejne ruchy są rozpatrywane za pomocą przepustki harcerskiej z okienkiem na podstawie wyniku z pierwszego ruchu. Jeśli to przejście się nie powiedzie, jest powtarzane z oknem o pełnej szerokości (tak samo jak zwykły AB negamax). Początkowe przeszukiwanie w szerokim oknie od pierwszego ruchu daje dobre przybliżenie dla testu zwiadowczego. Pozwala to uniknąć zbyt wielu niepowodzeń i wykorzystuje fakt, że test harcerski przycina dużą liczbę gałęzi.

Pseudo kod

Połączenie sterownika wyszukiwania aspiracji z algorytmem negascout tworzy potężną sztuczną inteligencję w grze. Aspiracja negascout to algorytm będący sercem większości najlepszych programów do gry na świecie, w tym programów do gry w szachy, warcaby i Reversi, które mogą pokonać mistrzów. Sterownik aspiracji jest taki sam, jak został zaimplementowany wcześniej:

```
1 function abNegascout(board: Board,
2 maxDepth: int,
3 currentDepth: int,
4 alpha: float,
5 beta: float) -> (float, Move):
6 # Check if we're done recursing.
7 if board.isGameOver() or currentDepth == maxDepth:
8 return board.evaluate(player), null
9
10 # Otherwise bubble up values from below.
11 bestMove: Move = null
12 bestScore: float = -INFINITY
13
14 # Keep track of the Test window value.
15 adaptiveBeta: float = beta
16
17 # Go through each move.
18 for move in board.getMoves():
19 newBoard: Board = board.makeMove(move)
20
21 # Recurse.
22 recursedScore, currentMove = abNegamax(
23 newBoard, maxDepth, currentDepth + 1,
```

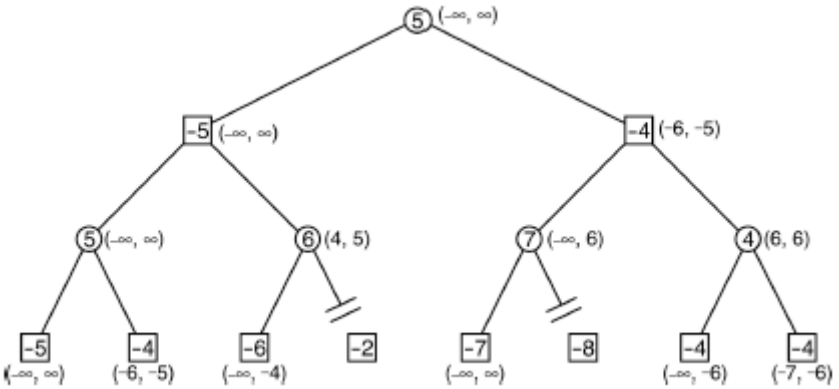
```
24 -adaptiveBeta, -max(alpha, bestScore))
25 currentScore = -recursedScore
26
27 # Update the best score.
28 if currentScore > bestScore:
29 # If we are in 'narrow-mode' then widen and do a regular
30 # AB negamax search.
31 if adaptiveBeta == beta or currentDepth >= maxDepth - 2:
32 bestScore = currentScore
33 bestMove = move
34
35 # Otherwise we can do a Test.
36 else:
37 negativeBestScore, bestMove = abNegascout(
38 newBoard, maxDepth, currentDepth,
39 -beta, -currentMoveScore)
40 bestScore = -negativeBestScore
41
42 # If we're outside the bounds, prune by exiting.
43 if bestScore >= beta:
44 return bestScore, bestMove
45
46 # Otherwise update the window location.
47 adaptiveBeta = max(alpha, bestScore) + 1
48
49 return bestScore, bestMove
```

Struktury danych i interfejsy

Ta aukcja korzysta z tego samego interfejsu planszy co poprzednio i można ją zastosować do dowolnej gry.

Wydajność

Zgodnie z przewidywaniami, algorytm jest ponownie O(d) w pamięci, gdzie d jest maksymalną głębokością wyszukiwania, a kolejność O(nd) w czasie, gdzie n jest liczbą możliwych ruchów w każdej pozycji planszy. Rysunek przedstawia drzewo gry użyte do wprowadzenia wartości AB negamax.



Wydaje się, że wartości alfa i beta skaczą bardziej niż w przypadku negamax, ale podążanie za algorytmem negascout eliminuje dodatkową gałąź z wyszukiwania. Ogólnie rzecz biorąc, negascout dominuje AB negamax; zawsze sprawdza te same lub mniej tablic. Do niedawna aspiracja negascout była niekwestionowanym mistrzem algorytmów gier. Kilka nowych algorytmów opartych na podejściu do testów ze wzmocnioną pamięcią (MT) okazało się od tego czasu w wielu przypadkach lepsze. Teoretycznie żaden z nich nie jest lepszy, ale odnotowano znaczne przyspieszenie w podejściu MT. Algorytmy MT są opisane w dalszej części.

Przenieś Zamawianie i Negascout

Negascout opiera się na wyniku pierwszego ruchu z każdej pozycji na planszy, aby kierować podaniem skauta. Z tego powodu ma jeszcze lepsze przyspieszenia niż AB negamax, gdy ruchy są uporządkowane. Jeśli najlepsza sekwencja ruchów jest pierwsza, to początkowe podanie przez szerokie okno będzie bardzo dokładne, a podanie harcerskie będzie rzadziej zawodzić. Ponadto, ze względu na konieczność ponownego przeszukiwania części drzewa gry, algorytm negascout korzysta w dużym stopniu z systemu pamięci (patrz następna sekcja), który może przywoływać wyniki poprzednich wyszukiwań.

Wyszukiwanie głównych odmian

Negascout jest ściśle powiązany z algorytmem zwanym Principal Variation Search (PVS). Kiedy negascout zawiedzie na swojej przepustce zwiadowczej, powtarza wyszukiwanie, wywołując siebie z szerszym oknem. PVS używa w tej sytuacji wywołania AB negamax. PVS ma również kilka mniejszych różnic w stosunku do negamaxa, ale ogólnie rzecz biorąc negascout działa lepiej w rzeczywistych zastosowaniach. Często nazwa PVS jest niejednoznacznie używana w odniesieniu do oryginalnego algorytmu negascout.

TABELE TRANSPOZYCJI I PAMIĘĆ

Do tej pory algorytmy, którym się przyglądaliśmy, zakładają, że każdy ruch prowadzi do unikalnej pozycji na planszy. Jak widzieliśmy wcześniej, ta sama pozycja na planszy może powstać w wyniku różnych kombinacji ruchów. W wielu grach ta sama pozycja na planszy może nawet wystąpić wiele razy w tej samej grze. Aby uniknąć wykonywania dodatkowej pracy przy kilkukrotnym przeszukiwaniu tej samej pozycji na planszy, algorytmy mogą wykorzystać tabelę transpozycji. Chociaż tabela transpozycji została zaprojektowana tak, aby uniknąć powielania pracy nad transpozycjami, ma ona dodatkowe

korzyści. Kilka algorytmów opiera się na tablicy transpozycji jako pamięci roboczej uwzględnionych pozycji planszy. Wszystkie techniki, takie jak test na pamięć, iteracyjne pogłębianie i myślenie o turze przeciwnika, wykorzystują tę samą tabelę transpozycji (wszystkie są przedstawione w tym rozdziale). W tabeli ranspozycji rejestrowane są pozycje na tablicy i wyniki wyszukiwania z tej pozycji. Kiedy algorytm otrzymuje pozycję płytki, najpierw sprawdza, czy płytka znajduje się w pamięci i używa zapisanej wartości, jeśli tak jest. Porównywanie kompletnych stanów gry jest kosztowną procedurą, ponieważ stan gry może zawierać dziesiątki lub setki elementów informacji. Porównywanie ich ze stanami przechowywanymi w pamięci zajęłoby dużo czasu. Aby przyspieszyć sprawdzanie tabeli transpozycji, używana jest wartość skrótu.

HASHOWANIE STANÓW GRY

Chociaż w zasadzie każdy algorytm haszujący będzie działał, istnieją szczególne cechy haszowania stanu gry dla tabel transpozycji. Większość możliwych stanów planszy w grze planszowej nigdy nie wystąpi. Reprezentują one wynik nielegalnych lub dziwacznych sekwencji ruchów. Dobry schemat haszowania rozłoży prawdopodobne pozycje tak szeroko, jak to możliwe, w zakresie wartości haszowania. Ponadto, ponieważ w większości gier plansza zmienia się bardzo niewiele z ruchu na ruch, przydatne jest posiadanie wartości skrótu, które zmieniają się znacznie, gdy na planszy wprowadzana jest tylko niewielka zmiana. Zmniejsza to prawdopodobieństwo zderzenia dwóch pozycji planszy, gdy występują one w tym samym wyszukiwaniem.

Klucze Zobrist

Istnieje wspólny algorytm haszowania tabeli przejść, zwany kluczami Zobrist. Klucz Zobrist to zestaw losowych wzorów bitowych o stałej długości, przechowywanych dla każdego możliwego stanu każdej możliwej lokalizacji na tablicy. Szachy mają 64 pola, a każde pole może być puste lub zawierać 1 z 6 różnych pionów, w każdym z dwóch możliwych kolorów. Klucz Zobrist do gry w szachy musi mieć długość 64 x 2 x 6 = 768 wpisów. Dla każdego niepustego kwadratu klucz Zobrist jest wyszukiwany i poddawany XOR z bieżącą sumą hash. Mogą istnieć dodatkowe klucze Zobrist dla różnych elementów stanu gry. Na przykład stan podwajania kości w tryktraku wymagałby sześcioelementowego klucza Zobrist. Szereg innych kluczy Zobrist jest wymaganych w szachach do reprezentowania zasady potrójnego powtórzenia, zasady 50 ruchów i innych subtelności. Niektóre implementacje pomijają te dodatkowe klucze, oczekując, że są one potrzebne tak rzadko, że oprogramowanie będzie cierpieć z powodu niejednoznaczności między sporadycznymi stanami w celu szybszego mieszania w zdecydowanej większości przypadków. Ten i inne problemy z tabele transpozycji zostaną omówione później. Dodatkowe klucze Zobrist są używane w ten sam sposób: ich wartości są wyszukiwane i poddawane XOR z bieżącą wartością skrótu. Ostatecznie zostanie wygenerowana ostateczna wartość skrótu. W celu implementacji długość wartości skrótu w kluczu Zobrist będzie zależeć od liczby różnych stanów tablicy. Gry w szachy mogą zadowolić się 32-bitowymi bitami, ale najlepsze są z 64-bitowym kluczem. Warcaby działają wygodnie z 32 bitami, podczas gdy bardziej złożona gra turowa może wymagać 128 bitów. Klucze Zobrist muszą zostać zainicjowane losowymi ciągami bitów o odpowiednim rozmiarze. Znane są problemy z funkcją rand w języku C (która jest często pokazywana jako funkcja losowa w wielu językach), a niektórzy programiści zgłaszali problemy podczas używania jej do inicjowania kluczy Zobrist. Inni programiści zgłosili pomyślne użycie rand. Ponieważ problemy z jakością generowania liczb losowych są trudne do debugowania (mają tendencję do obniżania wydajności, które jest trudne do wyśledzenia), prawdopodobnie bezpieczniej byłoby użyć jednego z wielu dostępnych generatorów liczb losowych o większej niezawodności niż skraj.

Implementacja skrótu

Ta implementacja pokazuje trywialny przypadek skrótu Zobrist dla kółka i krzyża. Każdy z dziewięciu kwadratów może być pusty lub zawierać jeden z dwóch elementów; dlatego w tablicy jest 9x2 = 18 elementów.

```
1 # The Zobrist key.

2 zobristKey: int[9 * 2]

3

4 # Initialize the key.

5 function initZobristKey():

6 for i in 0..(9 * 2):

7 zobristKey[i] = randomInt()
```

Na maszynie 64-bitowej ta implementacja wykorzystuje klucze 64-bitowe (16 bitów byłoby wystarczająco duże dla Tic-Tac-Toe, ale arytmetyka 64-bitowa jest zwykle szybsza). Opiera się na funkcji randomInt, która zwraca losową wartość. Po skonfigurowaniu klucza tablice można haszować. Ta implementacja funkcji mieszającej wykorzystuje strukturę danych tablicy zawierającą dziewięcioelementową tablicę reprezentującą zawartość każdego kwadrat na planszy:

```
1 # Calculate a hash value.

2 function hash(ticTacToe: Board) -> int:

3 # Start with a clear bitstring.

4 result: int = 0

5

6 # XOR each occupied location in turn.

7 for i in 0..9:

8 # Find what piece we have.

9 piece = board.getPieceAtLocation(i)

10

11 # If its unoccupied, lookup the hash value and xor it.

12 if piece != UNOCCUPIED:

13 result = result xor zobristKey[i * 2 + piece]

14

15 return result
```

Przyrostowe haszowanie Zobrist

Jedną ze szczególnie fajnych cech kluczy Zobrist jest to, że można je aktualizować stopniowo. Ponieważ każdy element jest XORed razem, dodanie elementu jest tak proste, jak XORing innej wartości. W powyższym przykładzie dodanie nowego elementu jest tak proste, jak XORowanie klucza Zobrist dla

tego nowego elementu. W grze takiej jak Chess, w której ruch polega na usunięciu pionka z jednego miejsca i dodaniu go do innego, odwracalny charakter operatora XOR oznacza, że aktualizacja może nadal być przyrostowa. Klucz Zobrist dla kawałka i starego kwadratu jest XOR z wartością hash, a następnie kluczem dla kawałka i nowego kwadratu. Przyrostowe haszowanie w ten sposób może być znacznie szybsze niż obliczanie haszowania na podstawie pierwszych zasad, zwłaszcza w grach z wieloma dziesiątkami lub setkami elementów jednocześnie.

Powrót do klasy gry

Aby obsługiwać haszowanie, a w szczególności przyrostowe haszowanie Zobrist, używaną przez nas klasę Board można rozszerzyć o ogólną metodę haszowania:

```
1 class Board:

2 # The current hash value for this board. This saves it being

3 # recalculated each time it is needed.

4 hashCache: int

5

6 function hashValue() -> int

7 function getMoves() -> Move[]

8 function makeMove(move: Move) -> Board

9 function evaluate() -> float

10 function currentPlayer() -> id

11 function isGameOver() -> bool
```

Wartość skrótu można teraz przechowywać w instancji klasy. Gdy wykonywany jest ruch (w metodzie move), wartość skrótu może być aktualizowana przyrostowo bez konieczności pełnego ponownego przeliczania.

CO PRZECHOWYWAĆ W TABELI

Tablica mieszająca przechowuje wartość powiązaną z pozycją tablicy, więc nie trzeba jej ponownie obliczać. Ze względu na sposób, w jaki wyniki są zbierane w górę drzewa w algorytmach negamax, znamy również najlepszy ruch z każdej pozycji planszy (to ten, którego wynikowa plansza ma najwyższy wynik odwrotny). Ten ruch można również przechowywać, dzięki czemu w razie potrzeby możemy wykonać ruch bezpośrednio. Celem poszukiwań jest poprawa dokładności funkcji oceny statycznej. Wartość minimax dla tablicy będzie zależeć od głębokości wyszukiwania. Jeśli szukamy do głębokości dziesięciu ruchów, nie będziemy zainteresowani wpisem w tabeli, który zawiera wartość obliczoną przez wyszukiwanie tylko trzy ruchy do przodu: nie byłby wystarczająco dokładny. Wraz z wartością wpisu w tabeli przechowujemy głębokość użytą do obliczenia tej wartości. Podczas wyszukiwania za pomocą przycinania AB nie jesteśmy zainteresowani obliczaniem dokładnego wyniku dla każdej pozycji planszy. Jeśli wynik znajduje się poza oknem wyszukiwania, jest ignorowany. Kiedy przechowujemy wartości w tabeli transpozycji, możemy przechowywać dokładną wartość lub możemy przechowywać wartości „fail-soft", które wynikają z przycinania gałęzi. Ważne jest, aby odnotować, czy wartość jest dokładna, czy jest to niska wartość (przycinanie alfa), czy wysoka (przycinanie beta). Można to osiągnąć za pomocą prostej flagi. Każdy wpis w tablicy mieszającej wygląda mniej więcej tak:

```
1 class TableEntry:
2 enum ScoreType:
3 ACCURATE
4 FAIL_LOW
5 FAIL_HIGH
6
7 # The hash value for this entry.
8 hashValue: int
9
10 # The type and score value.
11 scoreType: ScoreType
12 score: float
13
14 # The best move to make (as found on a previous calculation).
15 bestMove: Move
16
17 # The depth of calculation at which the score was found.
18 depth: int
```

IMPLEMENTACJA TABELI HASH

Dla szybkości, używana implementacja tablicy mieszającej jest często tablicą mieszającą. Ogólna tablica mieszająca ma tablicę list; tablice są często nazywane „wiaderkami”. Gdy element jest zahaszowany, wartość haszowania wyszukuje właściwy segment. Każdy element w zasobniku jest następnie sprawdzany pod kątem zgodności z wartością skrótu. Prawie zawsze jest mniej wiader niż możliwych kluczy. Klucz podlega modularnemu mnożeniu przez liczbę wiader, a nową wartością jest indeks wiadra do zbadania. Chociaż znacznie wydajniejszą implementację tablicy mieszającej można znaleźć w dowolnej standardowej bibliotece C++, ma ona ogólną postać:

```
1 class BucketItem:
2 # The table entry at this location.
3 entry: TableEntry
4
5 # The next entry in this bucket.
6 next: BucketItem
7
```

```
8 # Returns a matching entry from this bucket, even if it comes
9 # further down the list.
10 function getElement(hashValue):
11 if entry.hashValue == hashValue:
12 return entry
13 if next:
14 return next.getElement(hashValue)
15 return null
16
17 class HashTable:
18 # The contents of the table.
19 buckets: BucketItem[MAX_BUCKETS]
20
21 # Finds the bucket in which the value is stored.
22 function getBucket(hashValue: int) -> BucketItem:
23 return buckets[hashValue % MAX_BUCKETS]
24
25 # Retrieves an entry from the table.
26 function getEntry(hashValue: int) -> TableEntry:
27 return getBucket(hashValue).getElement(hashValue)
```

Celem jest posiadanie jak największej liczby wiader z dokładnie jednym wpisem. Jeśli kubełki są zbyt pełne, spowolni to wyszukiwanie i wskaże, że potrzeba więcej kubełków. Jeśli wiadra są zbyt puste, jest miejsce do stracenia i można użyć mniej wiader. Przy wyszukiwaniu ruchów ważniejsze jest, aby wyszukiwanie skrótów było szybkie, niż gwarantowanie, że zawartość tablicy mieszającej jest trwała. Nie ma sensu przechowywać w tablicy skrótów pozycji, które prawdopodobnie nigdy nie zostaną ponownie odwiedzone. Z tego powodu używana jest implementacja tablicy mieszającej, w której każde wiadro ma rozmiar jeden. Można to zaimplementować bezpośrednio jako tablicę rekordów i upraszcza powyższy kod do:

```
1 class HashArray:
2 # The entries.
3 entries: TableEntry[MAX_BUCKETS]
4
5 # Retrieves an entry from the table.
6 function getEntry(hashValue: int) -> TableEntry:
```

```
7 entry = entries[hashValue % MAX_BUCKETS]

8 if entry.hashValue == hashValue:

9 return entry

10 else:

11 return null
```

STRATEGIE WYMIANY

Ponieważ może istnieć tylko jeden przechowywany wpis dla każdego segmentu, musi istnieć jakiś mechanizm decydowania o tym, jak i kiedy zastąpić przechowywaną wartość, gdy wystąpi konflikt. Najprostszą techniką jest zawsze nadpisywanie. Zawartość wpisu w tabeli jest zastępowana za każdym razem, gdy chce się zapisać kolidujący wpis. Jest to łatwe do wdrożenia i często całkowicie wystarczające. Inną powszechną heurystyką jest zastępowanie za każdym razem, gdy węzeł kolidujący jest przeznaczony do późniejszego ruchu. Więc jeśli plansza w ruchu 6 zderza się z planszą w ruchu 10, używana jest plansza w ruchu 10. Jest to oparte na założeniu, że deska w ruchu 10 będzie użyteczna dłużej niż deska w ruchu 6. Istnieje wiele bardziej złożonych strategii zastępowania, ale nie ma ogólnej zgody co do tego, która jest najlepsza. Wydaje się prawdopodobne, że różne strategie będą optymalne dla różnych gier. Eksperymentowanie jest prawdopodobnie wymagane. Kilka programów odniosło sukces dzięki utrzymywaniu wielu tabel transpozycji przy użyciu różnych strategii. Każda tabela transpozycji jest sprawdzana po kolei pod kątem dopasowania. Wydaje się to równoważyć słabość każdego podejścia w stosunku do innych.

KOMPLETNA TABELA TRANSPOZYCJI

Pseudokod dla pełnej tablicy transpozycji wygląda następująco:

```
1 class TranspositionTable:

2 tableSize: int

3 entries: TableEntry[tableSize]

4

5 function getEntry(hashValue: int) -> TableEntry:

6 entry = entries[hashValue % tableSize]

7 if entry.hashValue == hashValue:

8 return entry

9 else:

10 return null

11

12 function storeEntry(entry: TableEntry):

13 # Always replace the current entry.

14 entries[entry.hashValue % tableSize] = entry
```

Wydajność

Metoda getEntry i metoda storeEntry powyższej implementacji są O(1) zarówno w czasie, jak iw pamięci. Ponadto sama tablica ma wartość O(n) w pamięci, gdzie n to liczba wpisów w tablicy. Powinno to być związane z czynnikiem rozgałęzienia gry i maksymalną używaną głębokością wyszukiwania. Duża liczba sprawdzonych pozycji planszowych wymaga dużego stołu.

Uwagi dotyczące implementacji

Jeśli zaimplementujesz ten algorytm, zdecydowanie zalecam dodanie do niego danych debugowania, które mierzą liczbę wiader używanych w dowolnym momencie, liczbę nadpisań czegoś i liczbę pominięć podczas uzyskiwania wpisu, który wcześniej został dodany. Pozwoli ci to zrozumieć, jak dobrze działa tabela transpozycji. Jeśli rzadko znajdziesz przydatny wpis w tabeli, to tabela może być źle sparametryzowana (na przykład liczba wiader może być zbyt mała lub strategia wymiany może być nieodpowiednia). Z mojego doświadczenia wynika, że tego rodzaju informacje dotyczące debugowania są nieocenione, gdy Twoja sztuczna inteligencja nie gra tak, jak się spodziewałeś.

KWESTIE Z TABELĄ TRANSPOZYCJI

Tabele transpozycji są ważnym narzędziem w uzyskiwaniu użytecznej prędkości z turowej sztucznej inteligencji. Nie są jednak panaceum i mogą wprowadzać własne problemy.

Zależność ścieżki

Niektóre gry wymagają punktacji zależnej od kolejności ruchów. Na przykład trzykrotne powtórzenie tego samego zestawu pozycji na szachownicy w szachach skutkuje remisem. Wynik pozycji na planszy będzie zależał od tego, czy jest to pierwsza czy ostatnia runda takiej sekwencji. Przechowywanie tabel transpozycji będzie oznaczać, że takie powtórzenia będą zawsze punktowane identycznie. Może to oznaczać, że sztuczna inteligencja omyłkowo odrzuca zwycięską pozycję, powtarzając sekwencję. W tym przypadku problem można rozwiązać poprzez włączenie klucza Zobrist do „liczby powtórzeń” w funkcji skrótu. W ten sposób kolejne powtórzenia mają różne wartości skrótu i są rejestrowane oddzielnie. Ogólnie jednak gry, które wymagają punktacji zależnej od sekwencji, muszą mieć bardziej złożone hashowanie lub specjalny kod w algorytmie wyszukiwania, aby wykryć tę sytuację.

Niestabilność

Trudniejszym problemem jest niestabilność, gdy przechowywane wartości zmieniają się podczas tego samego wyszukiwania. Ponieważ każdy wpis w tabeli może zostać nadpisany w różnym czasie, nie ma gwarancji, że ta sama wartość zostanie zwrócona za każdym razem, gdy zostanie wyszukana pozycja. Na przykład, gdy węzeł jest brany pod uwagę w wyszukiwaniu po raz pierwszy, znajduje się w tabeli transpozycji, a jego wartość jest wyszukiwana. Później w tym samym wyszukiwaniu ta lokalizacja w tabeli jest nadpisywana przez nową pozycję na planszy. Nawet później w przeszukiwaniu pozycja planszy jest przywracana (przez inną sekwencję ruchów lub przez ponowne przeszukanie w algorytmie negascout). Tym razem wartości nie można znaleźć w tabeli i jest ona obliczana przez wyszukiwanie. Wartość zwrócona z tego wyszukiwania może różnić się od wyszukiwanej wartości. Chociaż zdarza się to bardzo rzadko, może dojść do sytuacji, w której wynik dla tablicy oscyluje między dwiema wartościami, powodując, że niektóre wersje algorytmu przeszukiwania (choć nie jest to podstawowy negascout) zapętlają się w nieskończoność.

KORZYSTANIE Z CZASU NAMYŚLENIA PRZECIWNIKA

Tabeli transpozycji można użyć, aby umożliwić sztucznej inteligencji usprawnienie wyszukiwania, podczas gdy człowiek myśli. Podczas tury gracza komputer może wyszukać ruch, który wykonałby,

gdyby grał. Po przetworzeniu wyniki tego wyszukiwania są zapisywane w tabeli transpozycji. Kiedy sztuczna inteligencja przyjdzie na swoją kolej, jej wyszukiwanie będzie szybsze, ponieważ wiele pozycji na planszy będzie już uwzględnionych i zapisanych. Większość komercyjnych programów do gier planszowych wykorzystuje czas myślenia przeciwnika na dodatkowe wyszukiwanie i przechowywanie wyników w pamięci.

ALGORYTMY TESTOWE WZMOCNIONEJ PAMIĘCI

Algorytmy testów wzmocnionych pamięcią (MT) opierają się na istnieniu wydajnej tablicy transpozycji, która działa jako pamięć algorytmu. MT to po prostu negamax AB o zerowej szerokości, wykorzystujący tabelę transpozycji, aby uniknąć powielania pracy. Istnienie pamięci umożliwia algorytmowi przeskakiwanie po drzewie wyszukiwania, patrząc najpierw na najbardziej obiecujące ruchy. Rekurencyjny charakter algorytmu negamax oznacza, że nie może on skakać; musi bulgotać i powracać w dół.

TEST WDROŻENIA

Ponieważ rozmiar okna dla Testu jest zawsze równy zero, test jest często pisany od nowa, aby zaakceptować tylko jedną wartość wejściową (wartości A i B są takie same). Nazwiemy tę wartość „gamma”. Ten sam test został użyty w algorytmie negamax, ale w tym przypadku negamax został nazwany jako test i jako zwykły negamax, więc potrzebne były oddzielne parametry alfa i beta. Do uproszczonego algorytmu negamax dodano kod dostępu do tablicy transpozycji. W rzeczywistości znaczna część tego kodu to po prostu dostęp do pamięci.

Pseudo kod

Funkcję testową można zaimplementować w następujący sposób:

```
1 function test(board: Board,
2 maxDepth: int,
3 currentDepth: int,
4 gamma: float) -> (float, Move):
5 if currentDepth > lowestDepth:
6 lowestDepth = currentDepth
7
8 # Lookup the entry from the transposition table.
9 entry: TableEntry = table.getEntry(board.hashValue())
10
11 if entry and entry.depth > maxDepth - currentDepth:
12 # Early outs for stored positions.
13 if entry.minScore > gamma:
14 return entry.minScore, entry.bestMove
15 else if entry.maxScore < gamma:
```

```
16 return entry.maxScore, entry.bestMove
17 else:
18 # We need to create the entry.
19 entry.hashValue = board.hashValue()
20 entry.depth = maxDepth - currentDepth
21 entry.minScore = -INFINITY
22 entry.maxScore = INFINITY
23
24 # Now we have the entry, we can get on with the text.
25 # Check if we're done recursing.
26 if board.isGameOver() or currentDepth == maxDepth:
27 entry.minScore = entry.maxScore = board.evaluate()
28 table.storeEntry(entry)
29 return entry.minScore, null
30
31 # Now go into bubbling up mode.
32 bestMove: Move = null
33 bestScore: float = -INFINITY
34 for move in board.getMoves():
35 newBoard: Board = board.makeMove(move)
36
37 # Recurse.
38 recursedScore, currentMove = test(
39 newBoard, maxDepth, currentDepth + 1, -gamma)
40 currentScore = -recursedScore
41
42 # Update the best score.
43 if currentScore > bestScore:
44 # Track the current best move.
45 entry.bestMove = move
46 bestScore = currentScore
```

```
47 bestMove = move
48
49 # If we pruned, then we have a min score, otherwise we have a max.
50 if bestScore < gamma:
51 entry.maxScore = bestScore
52 else:
53 entry.minScore = bestScore
54
55 # Store the entry and return the best score and move.
56 table.storeEntry(entry)
57 return bestScore, bestMove
```

Tabela transpozycji

Ta wersja testu wymaga nieco innej struktury danych wpisów w tabeli. Przypomnijmy, że w frameworku negamax wynik wpisu w tabeli może być dokładny lub może być wynikiem wyszukiwania „fail-soft". Ponieważ wszystkie wyszukiwania w MT mają okno o zerowej szerokości, jest mało prawdopodobne, abyśmy otrzymali dokładny wynik, ale możemy zbudować wyobrażenie o możliwym zakresie wyników dla kilku wyszukiwań. W tabeli transpozycji rejestrowane są zarówno minimalne, jak i maksymalne wyniki. Działają one podobnie do wartości alfa i beta w algorytmie przycinania AB. Ponieważ tylko te dwie wartości muszą być przechowywane, nie ma potrzeby przechowywania typu partytury. Nowa struktura wpisów w tabeli wygląda następująco:

```
1 class TableEntry:
2 hashValue: int
3 minScore: float
4 maxScore: float
5 bestMove: Move
6 depth: int
```

ALGORYTM MTD

Procedura MT jest wywoływana wielokrotnie z procedury sterownika. Jest to procedura sterownika, która jest odpowiedzialna za wielokrotne używanie MT do powiększania prawidłowej wartości minimaksowej i wypracowywania następnego ruchu w procesie. Algorytmy tego typu nazywane są sterownikami testowymi wykorzystującymi pamięć lub MTD. Pierwsze algorytmy MTD miały bardzo różną strukturę, przy użyciu złożonych zestawów specjalnego kodu przypadku i logiki porządkowania wyszukiwania. SSS* i DUAL*, najbardziej znane, zostały pokazane aby uprościć do szczególnych przypadków algorytmu MTD. Proces uproszczenia rozwiązał również pewne nierozstrzygnięte problemy z oryginalnymi algorytmami. Wspólny algorytm MTD wygląda następująco:

• Śledź górną granicę wartości punktacji. Nazwij to gamma (aby uniknąć pomyłki z alfa i beta).

• Niech gamma będzie pierwszym przypuszczeniem co do wyniku. Może to być dowolna stała wartość lub może być wyprowadzona z poprzedniego przebiegu algorytmu.

• Oblicz kolejne przypuszczenie, wywołując Test na bieżącej pozycji planszy, maksymalnej głębokości, zero dla bieżącej głębokości i wartości gamma. (Wartość nieco mniejsza niż wartość gamma jest zwykle używana: gamma−ϵ, gdzie ϵ jest mniejsze niż najmniejszy przyrost funkcji oceny. Pozwala to procedurze testowej na uniknięcie użycia operatora ==, który powoduje asymetrię, gdy punkt widok jest odwracany wraz ze znakami partytur podczas rekurencji).

• Jeśli domysł nie zgadza się z wartością gamma, wróć ponownie do 3. Potwierdza to, że przypuszczenie jest teraz dokładne. Czasami niestabilności liczbowe mogą sprawić, że nigdy nie stanie się to prawdą, i zwykle nakłada się limit na liczbę iteracji.

• Zwróć przypuszczenie jako wynik; to jest dokładne.

Algorytmy MTD przyjmują parametr zgadywania. Jest to pierwsze przypuszczenie co do wartości minimax oczekiwanej przez algorytm. Im dokładniejsze jest to przypuszczenie, tym szybciej będzie działał algorytm MTD.

Odmiany MTD

Wykazano, że algorytm SSS* jest powiązany z MTD zaczynając od domysłu nieskończoności (znanego jako MT-SSS lub MTD+∞). Podobnie algorytm DUAL* może być emulowany przy użyciu minus nieskończoności jako początkowego odgadnięcia (MTD−∞). Najpotężniejszy ogólny algorytm MTD, MTD-f, wykorzystuje zgadywanie oparte na wynikach poprzedniego wyszukiwania. Istnieje wariant MTD, MTD-best, który nie oblicza dokładnych wyników dla każdej pozycji na planszy, ale może oddać najlepszy ruch. Jest nieznacznie szybszy niż MTD-f, ale znacznie bardziej złożony i nie określa, jak dobre są ruchy. W większości gier turowych ważne jest, aby wiedzieć, jak dobre są ruchy, więc MTD-best nie jest tak powszechnie używany.

Rozmiar pamięci

MTD opiera się na dużej pamięci. Jego wydajność znacznie się pogarsza, gdy w tabeli transpozycji Wydajnośćują kolizje, a różne pozycje płytek są mapowane do tego samego wpisu w tabeli. W najgorszym przypadku algorytm może nie być w stanie zwrócić wyniku, jeśli pamięć, której potrzebuje, będzie nadpisywana. Wymagany rozmiar tabeli zależy od współczynnika rozgałęzienia, głębokości wyszukiwania i jakości schematu mieszającego. W przypadku sztucznej inteligencji grającej w szachy z głębokim wyszukiwaniem powszechne są tabele rzędu dziesiątek megabajtów (kilka milionów wpisów w tabelach). Mniejsze wyszukiwania lub prostsze gry mogą wymagać o kilka rzędów wielkości mniej. Podobnie jak w przypadku wszystkich problemów z pamięcią, należy uważać, aby nie popaść w problemy z wydajnością pamięci typowe dla dużych struktur danych. Trudno jest właściwie zarządzać wydajnością pamięci podręcznej dla 32-bitowego komputera przy użyciu struktur danych o rozmiarze większym niż megabajt.

PSEUDO-KOD

Pseudokod implementacji MTD, który może być użyty z podanym wcześniej kodem testowym, wygląda następująco:

```
1 function mtd(board: Board, maxDepth: int, guess: float) -> Move:

2 for i in 0..MAX_ITERATIONS:

3 gamma: float = guess
```

4 guess, move = test(board, maxDepth, 0, gamma-1)

5

6 # If there’s no more improvement, stop looking.

7 if gamma == guess:

8 break

9

10 return move

W tej formie MTD może być wywołane z nieskończonością jako pierwszym odgadnięciem (MT-SSS) lub może być uruchomione jako MTD-f z domysłem opartym na poprzednim wyszukiwaniu. W tym celu można użyć statycznej oceny ruchu lub można ją uruchomić jako część iteracyjnego algorytmu pogłębiania, który śledzi domysły od wyszukiwania do wyszukiwania.

Wydajność

Kolejność wykonywania tego algorytmu jest nadal taka sama jak poprzednio dla czasu (O(nd), gdzie n to liczba ruchów na planszy, a d to głębokość drzewa). W pamięci jest to O(s), gdzie s to liczba wpisów w tablicy transpozycji. MTD-f rywalizuje z aspiracją negascout jako najszybsze wyszukiwanie drzewa gry. Testy pokazują, że MTDf jest często znacznie szybszy, ale wciąż trwa debata, czy każdy algorytm można dalej zoptymalizować, aby poprawić jego wydajność. Chociaż wiele najlepszych programów do gier planszowych korzysta z negascout, większość współczesnej sztucznej inteligencji opiera się teraz na rdzeniu MTD. Podobnie jak w przypadku wszystkich problemów z wydajnością w sztucznej inteligencji, jedynym pewnym sposobem sprawdzenia, które z nich będą szybsze w twojej grze, jest wypróbowanie obu i sprofilowanie ich. Na szczęście żaden z algorytmów nie jest złożony i oba mogą używać tego samego kodu bazowego (tabele transpozycji, funkcja AB negamax i klasa gry).

WYSZUKIWANIE DRZEW MONTE CARLO (MCTS)

Podejścia Minimax sprawdzają się dobrze w grach z małymi czynnikami rozgałęzienia, gdzie dostępna jest funkcja oceny statycznej dobrej jakości. Często jednak brakuje jednego lub obu tych kryteriów. W 1987 roku Bruce Abramson zaproponował alternatywne podejście losowe [1], które stało się znane jako przeszukiwanie drzewa Monte Carlo (MCTS). Pomimo pewnych korzyści, jego użycie zostało przyćmione przez podejścia minimax, dopóki nie zostało z powodzeniem wykorzystane jako część technologii głębokiego uczenia Alpha Go, aby pokonać najlepszych graczy Go na świecie. Jest szczególnie odpowiedni do użytku w głębokim uczeniu się, ale nie tylko: był używany samodzielnie do gry ze sztuczną inteligencją zarówno w grach planszowych, jak i strategicznych.

CZYSTE WYSZUKIWANIE DRZEW MONTE CARLO

Metody Monte Carlo są losowe: nazwane na cześć słynnego kasyna w Monako. Dążą do uzyskania ogólnego wyniku, przeprowadzając dużą liczbę losowych prób. W przypadku MCTS próby są rozgrywkami (znanymi jako playouts), a wynik, do którego dążymy, jest najlepszym ruchem do wykonania. Mając do czynienia z zestawem możliwych ruchów, możemy spróbować każdego po kolei, a następnie przeprowadzić serię losowych zagrań z powstałych pozycji. Zapisy wygranych/przegranych tych rozgrywek mogą przybliżać, jak dobry jest ruch. Takie podejście nie wymaga statycznej funkcji oceny. Rekord wygranych/przegranych mówi nam, jak dobry jest ruch: skutecznie odkrywamy funkcję oceny. Jest problem z takim podejściem. Prawdziwa gra nie będzie rozgrywana przez wybieranie losowych ruchów, a robiąc to, prawdopodobnie przeoczymy najlepsze ruchy lub najbardziej

prawdopodobne odpowiedzi przeciwnika. Podczas rozgrywania chcielibyśmy, aby symulacja była nieco dokładniejsza niż losowe wybieranie ruchów. Osiąga się to poprzez rekurencyjne wykonywanie tego samego procesu. Określając, jaka może być reakcja przeciwnika, patrzymy na statystyki wygranych/przegranych jego ruchów. Które z kolei opierają się na statystykach wygranych/przegranych dla naszych kolejnych ruchów. W końcu kończą nam się te dane i dopiero wtedy przeprowadzamy playout w sposób losowy. Rysunek 9.12 pokazuje przykładowe drzewo dla gry, która nie może zakończyć się remisem, ze statystykami wygranych/rozegranych na każdym węźle. W każdym przypadku węzeł jest kształtowany dla gracza, który ma się poruszyć, a całkowita wygrana jest z jego perspektywy. Jeśli i = 1::n dzieci węzła mają wartości wygrane/zagrane równe wi/pi, wtedy węzeł będzie miał sumy w/p, gdzie

$$w = p - \sum_{i=1}^{n} w_i$$

i

$$p = \sum_{i=1}^{n} p_i$$

Zauważ, że na diagramie znajdują się tylko węzły ze statystykami. Drzewo przetwarzane przez algorytm jest podzbiorem całego drzewa gry, a mianowicie podzbiorem, dla którego dostępne są statystyki.

Algorytm

Algorytm jest iteracyjny i może być uruchamiany tyle razy, ile byśmy chcieli. Im więcej iteracji, tym lepiej działa. Zazwyczaj pozostawia się go do upływu określonego czasu, po czym używany jest najlepszy wynik.

Każda iteracja działa w czterech krokach:

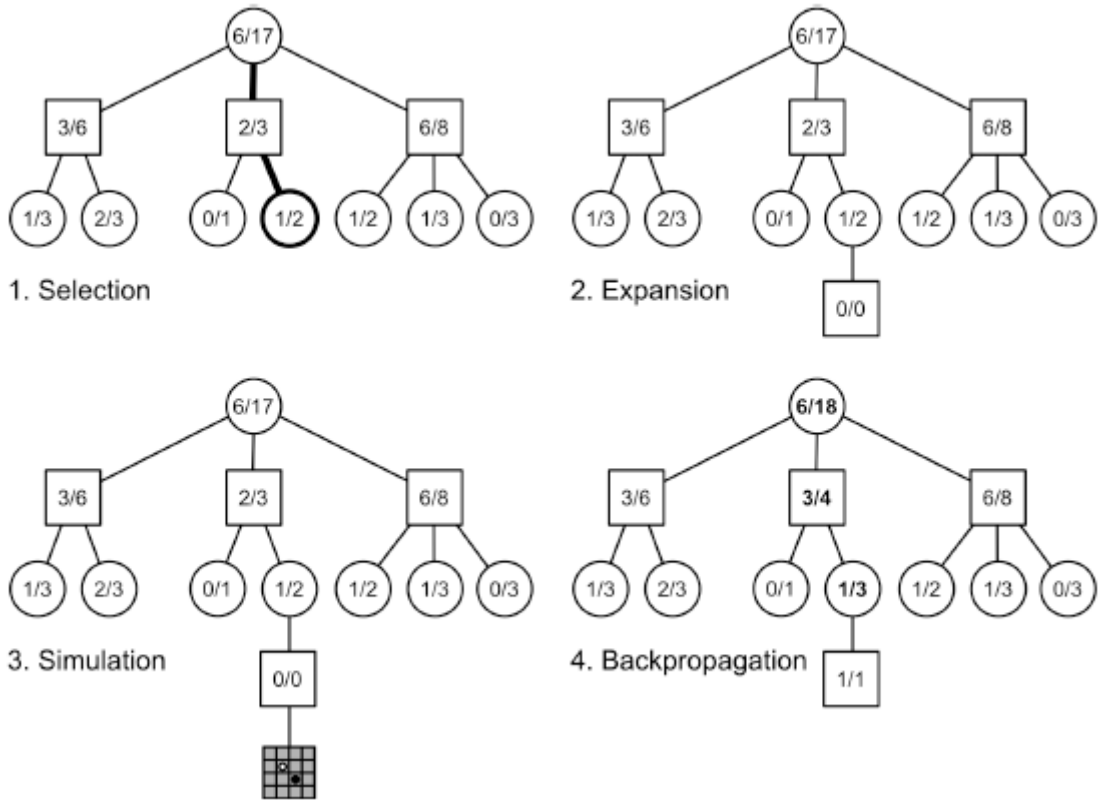
1. Wybór. Zejdź po drzewie, wybierając ruch na podstawie dotychczasowych sum, aż dojdziemy do węzła, który nie jest w pełni zbadany, tj. którego dzieci nie mają wszystkich statystyk.

2. Ekspansja. Wybierz losowo niezbadany ruch i dodaj go jako nowy węzeł do poddrzewa.

3. Symulacja. Rozegraj czysto losową grę z nowego węzła.

4. Propagacja wsteczna. Zaktualizuj sumy w drzewie w zależności od tego, czy rozgrywka doprowadziła do wygranej lub przegranej.

Te cztery kroki zostały zilustrowane w oparciu o poprzednie drzewo na rysunku.



Z biegiem czasu algorytm przeszukuje coraz głębiej najbardziej obiecujące gałęzie. Statystyki sukcesu wracają do góry, poprawiając ocenę, który ruch jest najlepszy z aktualnej pozycji (wierzchołek poddrzewa).

Wybór ruchu

W kroku wyboru algorytmu przechodzimy w dół drzewa, wybierając ruchy na podstawie dotychczasowych statystyk wygranych. Należy tu zachować równowagę. Z jednej strony chcemy więcej analizować blisko ruchów, które wyglądają obiecująco, z drugiej strony chcemy upewnić się, że wystarczająco zbadamy alternatywy. Ponieważ otrzymujemy więcej danych i możemy być bardziej pewni statystyk dotyczących ruchu, sensowne jest nakłanianie naszych poszukiwań do głębszego rozważenia tego. Ale nie chcemy zamykać się w dziwactwach losowości algorytmu i skupiać się na ruchu tylko dlatego, że na początku wyglądało to na udane. Jest to kompromis między poszukiwaniem nowej wiedzy a wykorzystaniem wiedzy, którą już posiadamy. Istnieją różne podejścia. Jedna z najprostszych i najczęstszych, znana jako UCT (górna granica ufności stosowana do drzew) waży każdy możliwy interfejs użytkownika ruchu zgodnie ze wzorem:

$$u_i = \frac{w_i}{n_i} + k\sqrt{\frac{\ln N}{n_i}}$$

gdzie w_i to liczba wygranych węzła, p_i to jego liczba playoutów, N to całkowita liczba playoutów dla wszystkich kandydujących węzłów, a k to stała, która kontroluje ilość eksploracji. Na każdym etapie spaceru wybierany jest węzeł o najwyższej wartości u_i.

Pseudo kod

Oto pseudokod do wykonywania iteracji przeszukiwania drzewa metodą Monte Carlo:

```
1 function mcts(board: Board):
2 # 1. Selection
3 current: Board = board
4 moveSequence: Board[] = [current]
5 while current.fullyExplored():
6 move: Move = current.selectMove()
7 current = current.makeMove(move)
8 moveSequence.push(current)
9
10 # 2. Expansion.
11 move = current.chooseUnexploredMove()
12 current = current.makeMove(move)
13 moveSequence.push(current)
14
15 # 3. Simulation.
16 winner: id = playout(current)
17
18 # 4. Backpropagation.
19 for current in moveSequence:
20 if winner == current.currentPlayer:
21 current.wins += 1
22 current.playouts += 1
```

Funkcja playout wykonuje czysto losowe rozegranie gry z danej pozycji na planszy i zwraca wynik. Jeśli wyszukiwanie w drzewie Monte Carlo było używane do zwrócenia ruchu, który ma wykonać sztuczna inteligencja, można użyć następującego prostego sterownika:

```
1 function mctsDriver(board: Board, thinkingTime: float) -> Move:
2 deadline = time() + thinkingTime
3 while time() < deadline:
4 mcts(board)
5
6 # Select a move with no further exploration.
```

```
7 return board.selectMove(0)
```

Struktury danych

Liczbę wygranych i liczbę playoutów należy zapisać dla każdej pozycji na planszy. Powyższy pseudokod zakłada, że tablica ma następującą strukturę:

```
1 class Board:
2 wins: int = 0
3 playouts: int = 0
4
5 function fullyExplored() -> bool
6 function selectMove(exploreCoeff: float = EXPLORE) -> Move
7 function chooseUnexploredMove() -> Move
8
9 function getMoves() -> Move[]
10 function makeMove(move:Move) -> Board
```

Funkcja fullExplored zwraca wartość true, jeśli wszystkie ruchy z bieżącej planszy prowadzą do pozycji, które mają statystyki wygranych. Można go zaimplementować po prostu jako:

```
1 class Board:
2 function fullyExplored() -> bool:
3 for move in getMoves():
4 next = makeMove(move)
5 if next.playouts == 0:
6 return false
7 return true
```

Wybór ruchu jest wykonywany przez selectMove przy użyciu obliczeń UCT:

```
1 class Board:
2 function selectMove(exploreCoeff: float = EXPLORE) -> Move:
3 lnN = ln(playouts)
4 bestMove = None
5 bestUCT = 0
6 for move in getMoves():
7 next = makeMove(move)
8 uct = next.wins / next.playouts +
```

```
9 exploreCoeff * sqrt(lnN / next.wins)

10 if uct > bestUCT:

11 bestUCT = uct

12 bestMove = move

13 return bestMove
```

W tej implementacji współczynnik eksploracji można przekazać jako parametr funkcji, chociaż domyślnie jest on ustawiony na wartość globalną. Pozwala to na użycie tej samej funkcji do zwrócenia najlepszego znalezionego ruchu (tj. UCT z zerowym współczynnikiem eksploracji) przez funkcję mctsDriver powyżej. I na koniec losowy nowy ruch jest wybierany do eksploracji przez funkcję selectUnexploredMove:

```
1 class Board:

2 function chooseUnexploredMove() -> Move:

3 unexplored = []

4 for move in getMoves():

5 next = makeMove(move)

6 if next.playouts == 0:

7 unexplored.push(move)

8 return randomChoice(unexplored)
```

Pozostałe funkcje w interfejsie płytki są takie same. Wdrożenie może być jednak inne. W przeciwieństwie do poprzednich pozycji na planszach nie można generować za każdym razem, gdy są potrzebne, muszą zachować swoje dane dotyczące wygranych. Wykonanie tego samego ruchu na tej samej planszy powinno zwrócić ten sam obiekt na planszy. W przypadku innych wyszukiwań drzew wystarczyło odesłanie znudzonego tym samym haszem. Jest to możliwe dzięki temu algorytmowi, ale statystyki wygranych musiałyby być wtedy przechowywane w tablicy mieszającej, a nie w samej strukturze tablicy.

Wydajność

Wyszukiwanie drzewa Monte Carlo to O(n) w pamięci, gdzie n jest liczbą węzłów w drzewie, dla których dostępne są dane wygranej. Czas wykonania jest nieokreślony, dominuje czas potrzebny na rozegranie gry. Teoretycznie może to być nieograniczone (chociaż w takim przypadku praktyczna implementacja powinna kończyć się najlepszymi przewidywaniami, czy gra zostanie wygrana). Z wyłączeniem odtwarzania (tj. zakładając, że czas odtwarzania jest stały), algorytm jest O(log n) w czasie, ponieważ obejmuje przejście w dół drzewa i wsteczną propagację danych.

DODAWANIE WIEDZY

Jak opisano powyżej, przeszukiwanie drzewa metodą Monte Carlo jest algorytmem bez wiedzy. Nie musi nic wiedzieć o grze, z wyjątkiem tego, czy rozgrywka kończy się wygraną. To sprawia, że jest to bardzo atrakcyjne dla ładowania AI: szybko produkuje coś, co może grać w tę grę; lub do wspierania gier, w których przyswajanie wiedzy jest trudne. Nie jest to jednak magiczna kula. Złota zasada AI nadal obowiązuje. Im mniej wiedzy dostarczymy, tym więcej będziemy musieli przeprowadzić poszukiwań.

W praktyce jest kilka punktów, w których możemy wprowadzić więcej wiedzy do systemu, zmniejszając wyszukiwanie i potencjalnie poprawiając jakość ostatecznej sztucznej inteligencji.

Ciężkie playouty

Zamiast wykonywać playout wyłącznie losowymi ruchami, możemy wybrać ruchy, które są bardziej prawdopodobne. Im więcej wyrafinowania włożono w wybór ruchu, który należy wykonać podczas rozgrywki, mówi się, że cięższe jest to przedstawienie. W najcięższym momencie moglibyśmy przeprowadzić playout, używając algorytmu minimax grającego przeciwko sobie, szukając najlepszego ruchu do wykonania w każdej turze. Ale to w dużej mierze pokonałoby cel korzystania z MCTS. Może to również mieć negatywny wpływ na jakość sztucznej inteligencji: MCTS daje lepsze wyniki, gdy w rozgrywkach jest trochę niedokładnych gier, więc wie, jak unikać nadchodzących pułapek. Częściej używa się heurystyki bez wyszukiwania. Rozważany jest każdy możliwy ruch i wybierany jest najlepszy. Widzieliśmy już w tym rozdziale mechanizm do tego: funkcję oceny statycznej. Heurystyki oceny dla ciężkich rozgrywek są często znacznie prostsze niż statyczne funkcje oceny używane w przeszukiwaniu drzewa minimaksowego, ale jest to tylko różnica stopnia. W obu przypadkach oceniamy pozycję na planszy i zwracamy liczbę. Alternatywnie, zamiast oceniać pozycje na planszy, możliwe jest stworzenie heurystyki, która patrzy tylko na ruch. To zależy od gry. Gry, w których miejsca na planszy wypełniają się pionkami, mogą uznać ten sam ruch za mniej więcej tak samo dobry, gdy gra się go w dowolnym momencie na drzewie. Jest to mniej skuteczne w grach takich jak Szachy.

Priorytety współczynnika wygranych

Innym sposobem na wprowadzenie wiedzy do przeszukiwania drzewa Monte Carlo jest dodanie węzłów do poddrzewa z istniejącą wartością ich statystyk wygranych. Jest to znane jako obciążenie ich wcześniejszego, ponieważ wpływa na prawdopodobieństwo, że ruch zostanie wybrany w fazie wyboru algorytmu. Jeśli wybierzemy ruch za pomocą UCT lub podobnych kryteriów, zarówno liczba wygranych, jak i liczba playoutów jest znacząca. Stosunek wskazuje, jak dobry naszym zdaniem ruch jest, a liczba playoutów wskazuje, jak pewni jesteśmy w tej ocenie. Możemy obliczyć a priori w ten sam sposób, w jaki obciążyliśmy ciężkie playouty powyżej: używając heurystyki ruchu lub uproszczonej funkcji oceny statycznej.

To potężna technika, ale należy zachować ostrożność. Jeśli przeor jest zbyt daleki od prawdy, MCTS zajmie bardzo dużą liczbę iteracji, aby naprawić błąd.

Funkcja oceny statycznej

Jeśli oceniamy ruchy za pomocą funkcji oceny, możemy zastosować tę ocenę bezpośrednio do fazy selekcji, zamiast używać statystyk wygranych/rozegranych jako pośrednich. Wiąże się to ze zmianą UCT w celu włączenia funkcji oceny:

$$u_i = \frac{w_i}{n_i} + k\sqrt{\frac{\ln N}{n_i}} + ce(B_i)$$

gdzie e(Bi) jest wartością funkcji oceny na planszy po ruchu i, a c jest stałą kontrolującą znaczenie tego dla selekcji. Równoważenie c i k w tym wzorze wymaga eksperymentowania.

KSIĄŻKI OTWIERAJĄCE I INNE ROZGRYWKI Z KOMPLETÓW

W wielu grach przez wiele lat doświadczeni gracze zdobyli doświadczenie na temat tego, które ruchy są lepsze od innych na początku gry. Nigdzie nie jest to bardziej oczywiste niż w początkowej księdze Szachów. Doświadczeni gracze badają ogromne bazy danych kombinacji o stałym otwarciu, ucząc się

najlepszych reakcji na ruchy. Nie jest niczym niezwykłym, że pierwsze 20 do 30 ruchów w Grandmaster Chess jest planowane z wyprzedzeniem. Książka otwierająca to lista sekwencji ruchów wraz z pewnym wskazaniem, jak dobry będzie średni wynik przy użyciu tych sekwencji. Korzystając z tych zestawów reguł, komputer nie musi przeszukiwać za pomocą minimaxingu, aby ustalić, jaki jest najlepszy ruch. Może po prostu wybrać następny ruch z sekwencji, o ile punkt końcowy sekwencji jest dla niego korzystny. Bazy danych książek otwierających można pobrać dla kilku różnych gier, a dla znanych gier, takich jak Szachy, komercyjne bazy danych są dostępne do licencjonowania w nowej grze. W oryginalnej grze turowej otwierająca księga (jeśli jest przydatna) musi zostać wygenerowana ręcznie.

WDRAŻANIE KSIĄŻKI OTWARCIA

Często otwieranie książek jest implementowane jako tablica mieszająca bardzo podobna do tablicy transpozycji. Listy sekwencji ruchów można zaimportować do oprogramowania i przekonwertować tak, aby każda pozycja pośrednia miała wskazanie linii otwierającej, do której należy, oraz siłę każdej linii. Zauważ, że w przeciwieństwie do zwykłej tabeli transpozycji, może być więcej niż jeden zalecany ruch z każdej pozycji na planszy. Pozycje na planszy mogą często należeć do wielu różnych linii otwarcia, a otwarcia, podobnie jak reszta gry, rozgałęziają się w formie drzewa. Ta implementacja obsługuje transpozycje automatycznie: sztuczna inteligencja sprawdza aktualną pozycję planszy w księdze otwierającej i znajduje zestaw możliwych ruchów do wykonania.

Księga otwierająca i ocena

Oprócz używania książki otwierającej jako specjalnego narzędzia, można ją włączyć do algorytmu wyszukiwania ogólnego przeznaczenia. Książka otwierająca jest często implementowana jako jeden z elementów funkcji oceny statycznej. Jeśli aktualna pozycja planszy jest częścią zarejestrowanego otwarcia, funkcja oceny statycznej ma duże znaczenie dla jej rady. Kiedy gra wykracza poza księgę otwierającą, jest ignorowana i wykorzystywane są inne elementy funkcji.

NAUKA OTWIERANIA KSIĄŻEK

Niektóre programy używają początkowej biblioteki książek i dodają warstwę edukacyjną. Warstwa uczenia aktualizuje wyniki przypisane do każdej sekwencji otwierającej, dzięki czemu można wybrać lepsze otwarcia.

Można to zrobić na dwa sposoby. Najbardziej podstawową techniką uczenia się jest prowadzenie statystycznego zapisu sukcesu programu przy każdym otwarciu. Jeśli otwarcie jest wymienione jako dobre, ale program konsekwentnie na nim przegrywa, może zmienić punktację, aby uniknąć tego otwarcia w przyszłości. Wiele procesów, doświadczeń i analiz jest związanych z punktami przypisanymi do każdej linii otwarcia w komercyjnej bazie danych. Wiele z tych punktacji opiera się na długich historiach międzynarodowych gier eksperckich. Jest mało prawdopodobne, aby były one błędne w stosunku do wszystkich graczy. Ale każda grająca w grę sztuczna inteligencja będzie miala różne cechy. Otwarcie wymienione w bazie danych jako dobre może zakończyć się napiętą sytuacją strategiczną, w którą człowiek może grać dobrze, ale która powoduje, że komputer odczuwa wiele efektów horyzontu. Włączenie statystycznej warstwy uczenia się pozwala komputerowi wykorzystać jego wyjątkowe mocne strony. Niektóre gry również same uczą się sekwencji. W ciągu wielu gier (zwykle wielu tysięcy) pewne linie otwarcia będą się powtarzać w kółko. Początkowo komputer musi polegać na swoich poszukiwaniach, aby je ocenić, ale z czasem wyniki te można uśredniać (wraz z informacjami o statystycznym prawdopodobieństwie wygranej) i rejestrować. Większe bazy otwarć Szachy i większość otwarć dla mniej popularnych partii są generowane w ten sposób: silny komputer gra sam i zapisuje najkorzystniejsze linie otwarcia.

USTAW KSIĄŻKI ZABAWY

Mimo że sekwencje ruchów są najczęściej spotykane na początku gry, można je również zastosować później. Wiele gier ma ustalone kombinacje ruchów, które Wydajnośćują podczas gry. Jednak w prawie wszystkich grach zakres możliwych pozycji plansz w grze jest oszałamiający. Jest mało prawdopodobne, że jakakolwiek konkretna pozycja na tablicy będzie dokładnie taka sama jak ta w bazie danych. Wymagane jest bardziej wyrafinowane dopasowanie wzorów: poszukiwanie określonych wzorów w ogólnej strukturze deski. Najczęstszym zastosowaniem tego typu bazy danych są podsekcje tablicy. Na przykład w Reversi kluczowa jest mocna gra wzdłuż każdej krawędzi planszy. Wiele programów Reversi ma obszerne bazy danych konfiguracji krawędzi, wraz z wynikami dotyczącymi ich siły. Cztery konfiguracje krawędzi deski można łatwo wyjąć, a wyszukano wpis w bazie danych. W grze środkowej te punkty przewagi są wysoko ważone w funkcji oceny statycznej. W dalszej części gry są mniej przydatne (większość programów Reversi może całkowicie przeszukiwać ostatnie 10-15 ruchów gry, więc nie jest potrzebna funkcja oceny). Kilka programów eksperymentowało z wyrafinowanym rozpoznawaniem wzorców, aby używać zestawu gier, szczególnie w grach w Go i Szachy. Jak dotąd nie pojawiły się dominujące metody do powszechnego stosowania we wszystkich grach planszowych.

Końcowa baza danych

Bardzo późno w niektórych grach (takich jak szachy, tryktrak czy warcaby) plansza jest uproszczona. Często na tym etapie można wybrać wyszukiwanie w stylu książki otwierającej. Istnieje kilka komercyjnych baz danych końcówek (często nazywanych bazami tabel) dla szachów, obejmujących najlepszy sposób na wymuszenie mata z różnymi kombinacjami materiału. Są one jednak rzadko wymagane w grach eksperckich, gdy gracz rezygnuje, gdy zmierza do znanego przegranego zakończenia.

DALSZE OPTYMALIZACJA

Chociaż podstawowe algorytmy rozgrywki są stosunkowo proste, mają oszałamiający wachlarz różnych optymalizacji. Niektóre z tych optymalizacji, takie jak tabele przycinania AB i transpozycji, są niezbędne dla dobrej wydajności. Inne optymalizacje są przydatne do wydobycia ostatniego kawałka wydajności. W tej sekcji przyjrzymy się kilku innym optymalizacjom stosowanym w turowej sztucznej inteligencji. W przypadku większości z nich nie ma wystarczająco dużo miejsca na omówienie szczegółów implementacji. Dodatek zawiera wskazówki do dalszych informacji na temat ich wdrażania. Ponadto nie uwzględniono konkretnych optymalizacji stosowanych tylko w stosunkowo niewielkiej liczbie gier planszowych. W szczególności szachy mają całą gamę specyficznych optymalizacji, które są przydatne tylko w niewielkiej liczbie innych scenariuszy.

POGŁĘBIANIE ITERATYWNE

Jakość gry z algorytmu wyszukiwania zależy od liczby ruchów, które może wyprzedzić. W przypadku gier z dużym współczynnikiem rozgałęzień, wybieganie nawet o kilka ruchów do przodu może zająć bardzo dużo czasu. Przycinanie ogranicza wiele poszukiwań, ale większość pozycji na planszy nadal należy wziąć pod uwagę. W przypadku większości gier komputer nie może sobie pozwolić na luksus myślenia tak długo, jak chce. Gry planszowe, takie jak Szachy, wykorzystują mechanizmy czasowe, a współczesne gry komputerowe mogą umożliwiać graczom grę we własnym tempie. Ponieważ algorytmy minimaksowania przeszukują do ustalonej głębokości, nie ma gwarancji, że wyszukiwanie zostanie zakończone w czasie, gdy komputer będzie musiał wykonać ruch. Aby uniknąć złapania bez ruchu, można zastosować technikę zwaną pogłębianiem iteracyjnym. Iteracyjne pogłębiające wyszukiwanie minimaksów wykonuje regularne minimaksy ze stopniowo zwiększającymi się głębokościami. Początkowo algorytm przeszukuje jeden ruch do przodu, następnie, jeśli ma czas,

przeszukuje dwa ruchy do przodu i tak dalej, aż skończy się czas. Jeśli czas upłynie przed zakończeniem wyszukiwania, wykorzystuje wynik wyszukiwania z poprzedniej głębokości.

Wdrożenie MTD

Algorytm MTD z iteracyjnym pogłębianiem, MTD-f, wydaje się być najszybszym algorytmem ogólnego przeznaczenia do wyszukiwania gier. Omawianą wcześniej implementację MTD można wywołać z następującego iteracyjnego frameworka pogłębiającego:

```
1 function mtdf(board: Board, maxDepth: int) -> (float, Move):

2 guess: float = 0

3

4 # Iteratively deepen the search.

5 for depth in 2..maxDepth:

6 guess, move = mtd(b, depth, guess)

7

8 # Check if we need a result.

9 if outOfTime():

10 break

11

12 return guess, move
```

Początkowa głębokość pogłębiania iteracyjnego wynosi dwa. Początkowe głębokie przeszukiwanie o jeden poziom często nie ma przewagi szybkości; na tym poziomie jest niewiele przydatnych informacji. Jednak w niektórych grach z dużymi czynnikami rozgałęziania lub gdy czas jest krótki, należy uwzględnić głębokie wyszukiwanie na jednym poziomie. Funkcja outOfTime zwraca wartość true, jeśli wyszukiwanie nie powinno być kontynuowane.

Heurystyka historii

W algorytmach wykorzystujących tablice transpozycji lub inną pamięć iteracyjne pogłębianie może być pozytywną zaletą algorytmu. Algorytmy, takie jak negascout i AB negamax, można radykalnie ulepszyć, rozważając najpierw najlepsze ruchy. Iteracyjne pogłębianie pamięcią pozwala na szybką analizę ruchu na płytkim poziomie, a następnie powrót do niego na głębszej. Wyniki wyszukiwania płytkiego można wykorzystać do uporządkowania ruchów do głębszego wyszukiwania. Zwiększa to liczbę suszonych śliwek, które można wykonać, i przyspiesza działanie algorytmu. Wykorzystanie wyników poprzedniej iteracji do uporządkowania ruchów nazywa się heurystyką historii. Jest to heurystyka, ponieważ opiera się na zasadzie kciuka, że poprzednia iteracja da dobre oszacowanie najlepszego ruchu.

PODEJŚCIA O ZMIENNEJ GŁĘBOKOŚCI

Przycinanie AB jest przykładem algorytmu o zmiennej głębokości. Nie wszystkie gałęzie są przeszukiwane na tej samej głębokości. Niektóre gałęzie są przycinane, jeśli komputer uzna, że nie musi ich już uwzględniać. Ogólnie jednak wyszukiwania mają stałą głębokość. Warunek w wyszukiwaniu sprawdza, czy osiągnięto maksymalną głębokość i kończy tę część algorytmu. Algorytmy można

zmieniać, aby umożliwić przeszukiwanie o zmiennej głębokości na dowolnej liczbie obszarów, a różne techniki przycinania wyszukiwania mają różne nazwy. Nie są to nowe algorytmy, ale po prostu wskazówki, kiedy przestać przeszukiwać gałąź.

Rozszerzenia

Główną słabością graczy komputerowych do gier turowych jest efekt horyzontu. Efekt horyzontu występuje, gdy ustalona sekwencja ruchów kończy się na czymś, co wydaje się doskonałą pozycją, ale jeden dodatkowy ruch pokaże, że ta pozycja jest w rzeczywistości okropna. Na przykład w szachach komputer może znaleźć serię ruchów, które pozwolą mu schwytać wrogą królową. Niestety zaraz po tym zbiciu przeciwnik może od razu dać mata. Gdyby komputer szukał na nieco większej głębokości, zobaczyłby ten wynik i nie wybrałby fatalnego ruchu. Niezależnie od tego, jak głęboko wygląda komputer, ten efekt może nadal Wydajnośćować. Jeśli jednak wyszukiwanie jest bardzo głębokie, komputer będzie miał wystarczająco dużo czasu, aby wybrać lepszy ruch, gdy problem zostanie w końcu wykryty. Jeśli poszukiwania nie mogą być kontynuowane na dużej głębokości z powodu dużych rozgałęzień i jeśli efekt horyzontu jest zauważalny, algorytm minimaksów może użyć techniki zwanej rozszerzeniami. Rozszerzenia to technika o zmiennej głębokości, w której kilka najbardziej obiecujących sekwencji ruchów jest przeszukiwanych na znacznie większą głębokość. Wybierając tylko najbardziej prawdopodobne ruchy do rozważenia w każdej turze, rozszerzenie może mieć wiele poziomów. Często zdarza się, że rozszerzenia o 10 do 20 ruchów są brane pod uwagę przy podstawowej głębokości wyszukiwania 8 lub 9 ruchów.

Rozszerzenia są często przeszukiwane przy użyciu iteracyjnego podejścia pogłębiającego, w którym tylko najbardziej obiecujące ruchy z poprzedniej iteracji są dalej rozszerzane. Chociaż często może to rozwiązać problemy z efektem horyzontu, w dużym stopniu opiera się na funkcji oceny statycznej, a słaba ocena może prowadzić do rozszerzenia komputera na bezużyteczny zestaw opcji.

Przycinanie w spoczynku

Jest wiele gier, w których gracz, który wydaje się wygrywać, może zmieniać się bardzo szybko, nawet z każdą turą. W tych grach efekt horyzontu jest bardzo wyraźny i może bardzo utrudnić implementację turowej sztucznej inteligencji. Często te szaleńcze zmiany przywództwa są tymczasowe i ostatecznie prowadzą do stabilnych stanowisk w zarządzie z wyraźnym liderem. Kiedy następuje okres względnego spokoju, głębsze poszukiwania często nie dostarczają dodatkowych informacji. Lepiej wykorzystać czas komputera na przeszukanie innego obszaru drzewa lub wyszukanie rozszerzeń w najbardziej obiecujących liniach. Przycinanie poszukiwań w oparciu o stabilność deski nazywa się przycinaniem w spokoju. Gałąź zostanie przycięta, jeśli jej wartość heurystyczna nie zmieni się znacząco na kolejnych głębokościach wyszukiwania. To prawdopodobnie oznacza, że wartość heurystyczna jest dokładna i nie ma sensu kontynuować tam poszukiwań. W połączeniu z rozszerzeniami, spokojne przycinanie pozwala na skupienie większości poszukiwań na obszarach drzewa, które są najbardziej krytyczne dla dobrej gry. Daje to lepszego przeciwnika komputerowego.

WIEDZA O GRZE

Algorytmy, na których koncentrowałem się do tej pory w tym rozdziale, to algorytmy przeszukiwania. Skutecznie rozważają możliwe ruchy (lub tak skutecznie, jak pozwala na to złożone drzewo gry). Jednak na własną rękę mogą grać tylko w najprostsze gry. Opierają się na wiedzy. W szczególności korzystają z dwóch źródeł wiedzy, jednego podstawowego i jednego opcjonalnego, które pozwalają algorytmom wnioskować o prowadzonej grze.

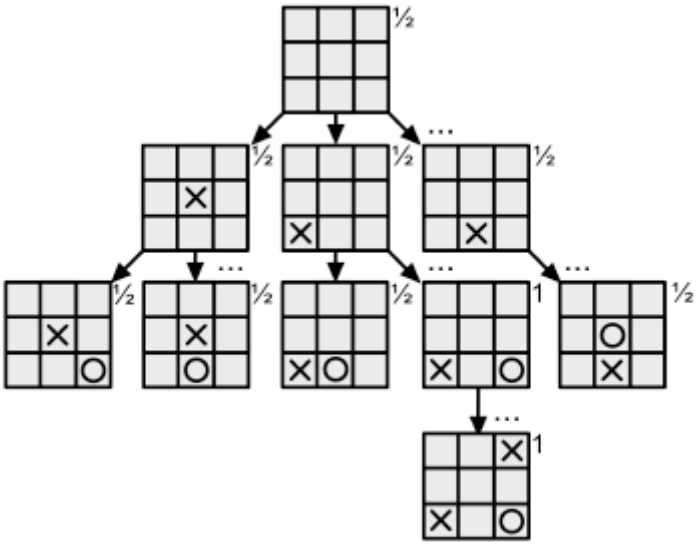
1. Funkcja oceny statycznej reprezentuje wiedzę o tym, jak dobra jest pozycja na planszy z perspektywy jednego gracza.

2. Porządkowanie ruchów może radykalnie poprawić wydajność wyszukiwania, uwzględniając najpierw najbardziej obiecujące ruchy. Gdzie „najbardziej obiecująca" jest wiedza o grze.

Kolejność ruchów jest opcjonalna, ponieważ zawsze można ją określić pod kątem funkcji oceny statycznej: najbardziej obiecującym ruchem jest ten, który prowadzi do tablicy z wyższym wynikiem. Do niedawna rzadkością było posiadanie oddzielnej i dedykowanej funkcji zamawiania ruchów i nadal nie jest jasne, czy większość gier planszowych przynosi jakiekolwiek korzyści. W przypadku bardzo dużych drzew w grach takich jak Go wykazano, że inna heurystyka kolejności ruchów może poprawić ogólną wydajność: mianowicie najbardziej obiecujący ruch to ten, który najchętniej wybierze przeciwnik. To może nie być to samo, co najlepszy ruch, jak ocenia funkcja oceny statycznej. Regularnie różni się to w grach takich jak Go, rozgrywanych przeciwko elitarnym ludzkim przeciwnikom, którzy są kształceni w długiej i dobrze ugruntowanej grze setowej (znanej jako joseki in Go). Jeśli ma być oddzielona funkcja oceny ruchu od funkcji oceny statycznej, jest drugim źródłem wiedzy. Jest to wciąż na tyle rzadkie, że wspominam o tym tutaj ze względu na kompletność, ale w kolejnych sekcjach skupię się na funkcji oceny statycznej.

Prawdopodobieństwo i ocena

Wartość zwracana przez funkcję oceny statycznej jest probabilistyczną oceną jakości stanowiska. Koduje zarówno dobrą pozycję, jak i pewność, że autor komputera lub funkcji jest tego pewien. Zakładając, że 1 oznacza pewną wygraną, a 0 pewną przegraną, wartości zbliżone do 1 2 oznaczają zarówno remis, jak i brak pewności co do innego wyniku. Ma to sens: wchodząc do meczu w grze planszowej, zakłada się, że każdy z graczy może wygrać, obaj mają równe szanse. W grze, która pozwala na remisy, takiej jak szachy, może to być również najbardziej prawdopodobny wynik. W doskonałej grze informacyjnej dla dwóch graczy, funkcja doskonałej oceny statycznej przypisałaby każdej pozycji tylko 0, 1 2 (jeśli gra pozwalała na remisy) lub 1. W Kółko i krzyżyk, jeśli obaj gracze zagrają najlepszy ruch w każdej turze , gra zawsze kończy się remisem. Pusta plansza ma wartość 1 2 . W rzeczywistości każda plansza na drzewie będzie miała tę samą wartość, chyba że nastąpi po błędnym ruchu. Pokazuje to rysunek.



Mówi się, że mecz jest słabo rozwiązany, jeśli znany jest wynik z pozycji wyjściowej, tj. jeśli wiemy przed rozpoczęciem gry, czy idealna gra doprowadzi do wygranej, przegranej lub remisu. Kółko i krzyżyk zostało rozwiązane, podobnie jak Warcaby (remis) i Połącz 4 (pierwszy gracz wygrywa). Silne rozwiązanie gry idzie dalej i określa optymalny ruch z dowolnej pozycji na planszy. Jest to równoznaczne z możliwością określenia idealnej funkcji oceny statycznej dla dowolnej pozycji. Gry, które nie mają doskonałej informacji, również można rozwiązać. Na przykład Zgadnij kto? to wygrana pierwszego gracza w 63% przypadków, jeśli obaj gracze grają perfekcyjnie. Statyczna funkcja oceny na początku gry, przed wylosowaniem kart, z perspektywy pierwszego gracza wynosi 0:63. Rozwiązywanie nawet skromnie skomplikowanych gier to bardzo trudne matematyczne wyzwanie. Rozwiązanie Warcabów i Połącz 4 zajęło lata, a większość ekspertów zgadza się, że rozwiązanie dla gry takiej jak Szachy nie jest bliskie. Nawet w przypadku najmocniej rozwiązywanych gier i wszystkich słabo rozwiązanych, praktyczna sztuczna inteligencja będzie musiała być probabilistyczna. Wartości w funkcji oceny statycznej obejmują zarówno jakość pozycji, jak i pewność algorytmu.

Zakres funkcji punktacji

Podczas gdy algorytmy minimax mogą działać z 0 dla przegranej i 1 dla wygranej, negamax zakłada, że wygrana i przegrana mają tę samą wartość, ale inny znak. Częściej jest przyznawane +k za wygraną i -k za przegraną, gdzie 0 oznacza remis lub brak pewności, gdzie ±k jest zakresem funkcji. Jak opisano wcześniej, niektóre algorytmy wyszukiwania wymagają wartości całkowitych dla funkcji oceny statycznej. Zwłaszcza te, które obejmują poprawną wartość w górnej i dolnej granicy i kończą się, gdy te granice się zbiegają. W praktyce zdobycie +1 za wygraną i -1 za przegraną może nie być pomocne. Często używane są większe wartości. Niektóre algorytmy wyszukiwania działają lepiej, gdy zakres wartości jest mały (na przykład ±100), podczas gdy inne preferują większe zakresy. Podczas ustawiania małego zakresu ważne jest, aby mieć wystarczająco dużo możliwych wartości, aby przedstawić subtelne zmiany jakości pozycji. Algorytm może rozróżniać tylko różne wartości: im większy zakres, tym więcej możliwych wartości pośrednich. Wiele prac nad turową sztuczną inteligencją wynikało z programów szachowych. Wyniki w szachach są często podawane w kategoriach „wartości" pionka. Wspólna skala przypisuje pionkowi wartość 100. Pozwala to na strategiczną punktację na poziomie setnej wartości pionka (tzw. „centippawn"). W tej skali wartość wygranej lub przegranej musi przekraczać to, co w przeciwnym razie może zwrócić funkcja oceny statycznej. Jeśli funkcja polega wyłącznie na liczeniu sztuk za pomocą ich konwencjonalne wartości (biskup i rycerz warte trzy pionki, wieże warte pięć, królowa warte osiem), wygrana musi zdobyć więcej niż 30 pionków (±3000 centipanów). W praktyce stosuje się znacznie większe wartości, aby uwzględnić punktację strategiczną i taktyczną, a także bicia bierek, oraz zakres ±10; 000 lub więcej byłoby mądre.

TWORZENIE FUNKCJI OCENY STATYCZNEJ

Do niedawnych postępów w uczeniu maszynowym statyczne funkcje oceny były najczęściej programowane bezpośrednio. Korzystanie z uczenia maszynowego nie jest niczym nowym. Jednym z pierwszych sukcesów gry planszowej w sztuczną inteligencję był program Warcaby Arthura Samuela, ukończony w 1956 roku [55]. Wykorzystał prostą formę uczenia maszynowego, aby dostosować strategie do przeciwników. Same strategie były jednak ręcznie kodowane. Od 2010 r. funkcje ewaluacyjne dla najlepszych AI grających w gry stały się domeną sieci neuronowych, w szczególności głębokich sieci neuronowych z własnym wbudowanym przeszukiwaniem drzewa, niezależnie od głównego algorytmu przeszukiwania drzewa. W tej sekcji skupię się na ręcznym tworzeniu funkcji oceny. W kolejnej sekcji przedstawię podejścia do uczenia maszynowego.

Pozyskiwanie wiedzy

Tworzenie funkcji oceny statycznej dla gry planszowej to zadanie polegające na zdobywaniu wiedzy. Wymaga zrozumienia dobrego człowieka i przekształcenia go w kod. W niektórych przypadkach programista jest ekspertem w grze, ale w większości potrzebni będą przynajmniej inni doradcy. Pozyskiwanie wiedzy od dawna stanowi trudność w sztucznej inteligencji, szczególnie w systemach eksperckich, które są zaprojektowane tak, aby zawierały zrozumienie eksperta przedmiotowego. Stworzenie jednej funkcji oceny, która zawiera całą wiedzę eksperta, jest bardzo trudne. Tak trudne, że rzadko się to podejmuje. Zamiast tego zakodowana jest seria funkcji oceny, z których każda obejmuje jeden element sytuacji taktycznej lub strategicznej. Na przykład w szachach może istnieć funkcja, która zwraca aktualną liczbę pionków dla każdego gracza; inny, który podaje liczbowe oszacowanie kontroli każdego gracza nad centrum (ważna uwaga taktyczna); gdzie inna stawia na bezpieczeństwo króla; i tak dalej. W końcu może ich być dziesiątki, a nawet setki. Ograniczona domena każdej oceny znacznie ułatwia kodowanie i testowanie. Ale algorytm wyszukiwania wymaga pojedynczej wartości, a problem łączenia wartości w jedną staje się znaczący. Niedługo wrócimy do tego zagadnienia. Jeśli tworzysz własną funkcję oceny statycznej, mądrze jest podzielić rzeczy w ten sposób. Ma również tę zaletę, że pozwala rozpocząć grę przeciwko komputerowi i testować sztuczną inteligencję przy użyciu tylko najbardziej oczywistych taktyk, a następnie dodać więcej, gdy wyjdą na jaw słabości w grze.

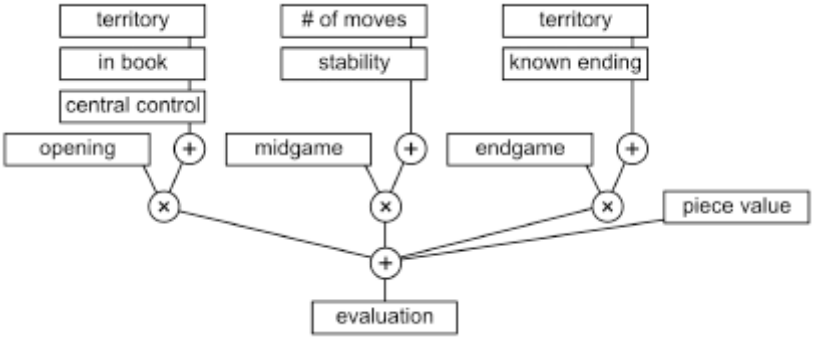
Wrażliwość na kontekst

„Statyczny” w pojęciu funkcji oceny statycznej odnosi się do faktu, że funkcja powinna zwracać tę samą wartość dla tej samej pozycji na planszy. Nie oznacza to jednak, że za każdym razem ten sam kod ewaluacyjny działa w ten sam sposób. Szczególnie, gdy funkcja oceny łączy prostsze funkcje, ważność tych elementów zmienia się w zależności od tego, jak daleko jesteśmy w grze. Na przykład w grze Reversi gracz, który kończy z największą liczbą żetonów w swoim kolorze, wygrywa. Tak więc posiadanie wielu liczników w twoim kolorze może wydawać się dobrą rzeczą. I tak jest na końcu. Ale w połowie gry najlepszą strategią jest często mieć najmniejszą liczbę żetonów, ponieważ daje to kontrolę nad inicjatywą w grze. Gracze Reversi nazywają tę mobilność. Są oczywiście subtelności, a inne błędy mogą z łatwością przeważyć nad tym czynnikiem (takie jak pozwolenie przeciwnikowi na przejęcie pola narożnego), ale z reguły w grze środkowej kontrolę sprawuje gracz z mniejszą liczbą kontr. Jeśli chcemy preferować pozycje z mniejszą liczbą kontr w grze środkowej, ale być bardziej zachłannymi w grze końcowej, mamy dwie możliwości. Możemy albo zakodować funkcję zliczającą dyski w taki sposób, aby na końcu uzyskiwała więcej dysków, ale karała je wcześniej w grze. Lub możemy przenieść tę złożoność o jeden poziom w naszym kodzie, aby funkcja oceniania dysku zawsze zwracała tę samą wartość, ale ważymy ją inaczej w czasie. W praktyce najlepszym podejściem jest zwykle kombinacja tych dwóch. Niektóre funkcje są łatwiejsze do zakodowania z wbudowaną wrażliwością na kontekst, inne są łatwiejsze do dostosowania podczas łączenia.

Łączenie funkcji punktacji

W tym samym czasie może działać wiele różnych mechanizmów punktacji. Jeden mechanizm punktacji może sprawdzać liczbę jednostek, które kontroluje każda ze stron, inny może sprawdzać schematy kontroli terytorium, a jeszcze inny może szukać konkretnych pułapek i obszarów niebezpiecznych. W złożonych grach może istnieć dziesiątki, a nawet setki mechanizmów punktacji. Wszystkie te różne funkcje oceniania muszą być połączone w jedną wartość, aby mogły być używane przez algorytm wyszukiwania. Może to być tak proste, jak dodanie wyników wraz z ustaloną wagą dla każdego z nich. Trudną częścią jest określenie, jakie powinny być wagi. Wspomniany powyżej program Samuela Checkers wykorzystywał sumę ważoną do połączenia mechanizmów punktacji, a następnie dodał prosty algorytm uczenia się, który mógł zmieniać wagi w oparciu o swoje doświadczenie. Jest to dobre podejście na początek, ale zwykle konieczne jest ręczne dostrojenie parametrów. Wrócę do nauki wag

poniżej. W ostatniej części opisałem funkcje oceny, które zmieniają się w trakcie gry i powiedziałem, że często są one implementowane poprzez zmianę wag, a nie samych funkcji. Na przykład wysoka liczba dysków w Reversi jest niezbędna pod koniec gry, ale może stanowić problem w środku. Na przykład w szachach zwyczajowo zwraca się większą uwagę na liczbę kontrolowanych pól na początku gry niż na jej końcu. Nie ma sensu posiadanie funkcji „kontrolowanej liczby kwadratów", która zmienia swoje dane wyjściowe w trakcie gry. Bardziej sensowna jest zmiana wagi. Nałożony zestaw funkcji punktacji przypomina analizy taktyczne w rozdziale 6: gdzie prymitywne taktyki są łączone w bardziej wyrafinowany obraz jakości sytuacji. I ta sama rada, jak je łączyć, dotyczy gier planszowych. Osobiście odniosłem sukces (opracowując sztuczną inteligencję dla Reversi i inną abstrakcyjną grę planszową o nazwie Atomy) dzięki prostej strukturze pokazanej na rysunku.



Niektóre oceny funkcji są sumowane, inne są łączone przez mnożenie. Jedna funkcja może reprezentować strategiczną troskę na planszy (np. liczbę kontrolowanych pól), druga może reprezentować wagę tej troski (może być ujemna w Reversi w grze środkowej i bardzo pozytywna na końcu). . Umożliwienie łatwego komponowania funkcji w ten sposób ułatwia wizualizację strategii podczas jej obliczania.

NAUKA FUNKCJI OCENY STATYCZNEJ

Ręczne kodowanie funkcji oceny statycznej jest żmudne i podatne na błędy. Od pierwszej gry planszowej, w której zastosowano sztuczną inteligencję, było jasne, że pewien rodzaj uczenia maszynowego byłby bardzo korzystny. Najwcześniejszym i najprostszym podejściem było zastosowanie uczenia maszynowego do wag poszczególnych strategii. Same strategie były proste (takie jak liczba elementów na gracza lub liczba kontrolowanych lokalizacji) i każda była wdrażana ręcznie. Algorytm uczący się następnie zdecydował, jak ważny powinien być każdy z nich. To podejście było wystarczająco silne, aby przynieść wczesne zwycięstwo w Checkers. Kiedy Tesauro zaimplementował funkcję oceny dla Backgammona [68], nie polegał na zestawie ręcznie zakodowanych strategii. Zamiast tego użył płytkiej sieci neuronowej, aby nauczyć się funkcji oceny od podstaw. To było na tyle udane, że TD-Gammon mógł rzucić wyzwanie dobrym ludzkim graczom. Ale nie stało się to wszechobecne. Kiedy Deep Blue IBM pokonał Gary'ego Kasparowa, ówczesnego mistrza świata w szachach, firma niechętnie podawała zbyt wiele szczegółów na temat swojego wewnętrznego działania (poza szczegółami sprzętu, na którym działała, co nie jest zaskoczeniem dla firmy sprzedającej sprzęt). Stwierdzili jedynie, że jego funkcja oceny była „złożona" [24] i działała częściowo w sprzęcie, a częściowo w oprogramowaniu. Niezależnie od tego, czy korzystał z nauki dostrajania parametrów, czy nie, liczba arcymistrzów konsultantów projektu sugeruje, że w dużym stopniu opierał się na zdobywaniu wiedzy eksperckiej. Niedawno deweloperzy zajęli się Go, od dawna niesławną najbardziej wymagającą grą głównego nurtu dla komputerów. Częścią trudności był brak sukcesu w rozłożeniu

strategii na indywidualne problemy. W Go nie jest do końca jasne, co naprawdę oznacza terytorium, zanim region zostanie rozegrany do ostatniego ruchu. Gracze Expert Go mówią w kategoriach „wagi" i „równowagi", a nawet „piękna" i „brzydoty", cech, które trudno zamienić w kod. A bez silnego zestawu strategii dotyczących komponentów uczenie maszynowe do dostosowywania parametrów nie jest zbyt przydatne. Innowacje w Go polegały na opracowaniu technik głębokiego uczenia się specyficznych dla gier planszowych, aby uzyskać całą funkcję oceny statycznej, na podstawie opisu planszy ([59] i [60]). W tej sekcji opiszę oba podejścia. Z mojego doświadczenia wynika, że proste podejście jest nadal przydatne, szczególnie w przypadku prostszych gier planszowych.

Optymalizacja parametrów wagi

Mając zestaw indywidualnych strategii, z których każda zwraca wartość liczbową, chcielibyśmy poznać odpowiednią serię wag. Wartość każdej strategii jest mnożona przez odpowiadającą jej wagę, a wyniki są sumowane w celu uzyskania wyniku statycznej funkcji oceny.

Naukę tę można przeprowadzić na wiele sposobów, wykorzystując prawie wszystkie techniki z Części 7 i wiele innych. Która metoda będzie słabo nadzorowana. Nie znamy prawidłowej wartości funkcji oceny statycznej, jedyne, co musimy zrobić, to skuteczność sztucznej inteligencji. Zakładamy, że im lepszy zestaw wag, tym funkcja oceny statycznej zbliża się do idealnej wartości i tym lepiej gra AI. Zasadniczo istnieją dwa sposoby podejścia do określania sukcesu.

1. Możemy grać dalej w grze i sprawdzać, czy wartość pozycji staje się wyraźniejsza, i wykorzystać tę późniejszą wartość do aktualizacji wcześniejszych wag.

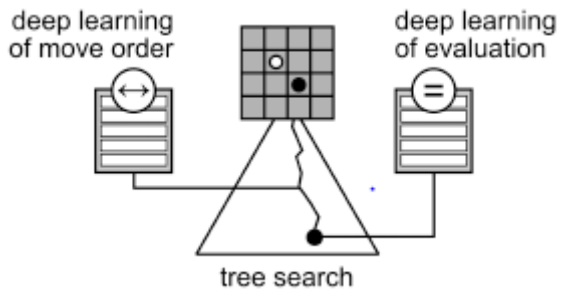
2. Możemy zagrać przeciwko sobie dwiema wersjami AI, z różnymi wagami i zobaczyć, która wygra.

Pierwsze podejście jest zasadniczo tym, które zostało użyte w programie Samuela Checkers, gdzie było znane jako „kopia zapasowa". Skutecznie wykorzystuje wyszukiwanie w drzewie (przy użyciu dowolnego algorytmu opisanego w tym rozdziale) w celu poprawy. Jeśli wyszukiwanie kilka ruchów do przodu poprawia wydajność, to być może możemy nauczyć się, aby funkcja oceny zwracała teraz to, co zwróci w kilku ruchach wyszukiwania. Jest to skuteczne i wydajne, ale wiąże się z poważnym problemem ładowania początkowego. Jeśli wagi są takie, że funkcja jest straszna, to będziemy się tylko uczyć oczekiwanej przyszłej wartości strasznego oceniającego. Aby tego uniknąć, niektóre wagi są często ustalane. W Warcabach była to liczba sztuk, w Szachach może to być całkowita wartość sztuk. W obu przypadkach ustalenie wartości zakotwicza funkcję oceny i sprawia, że jeśli bieżąca wersja gra fatalnie, zostanie to odzwierciedlone w nauce. Wydajność jest teraz mniejszym problemem niż w latach pięćdziesiątych. Możemy z łatwością uruchamiać setki kompletnych gier na sekundę na sprzęcie klasy konsumenckiej (zakładając, że przeprowadzamy bardzo mało wyszukiwania). Możliwe jest użycie formy wspinaczki górskiej, korzystając z algorytmu, który widzieliśmy w rozdziale 7.2.2 (przypomnijmy, że optymalizację możemy postrzegać jako znajdowanie wysokich punktów sprawności lub niskich punktów energii: w tym przypadku to drugie jest bardziej powszechne i algorytm ten jest zwykle nazywany „zejściem gradientowym"). Podsumowując: z początkowego zestawu losowych wag, zmutuj kolejno każdą wagę i rozegraj grę przeciwko oryginalnemu zestawowi. Następnie zostaje popełniona mutacja, która prowadzi do największej poprawy, a algorytm rozpoczyna się od nowa. Jeśli wszystkie mutacje są gorsze, zapisz aktualny zestaw i zacznij od nowa z nowymi losowymi wagami. Aby użyć tego podejścia, lepiej jest, aby ani sztuczna inteligencja, ani gra nie były bardzo losowe. Jeśli znacznie gorsza mutacja wag może się powieść wyłącznie przez przypadek, zwiększa się prawdopodobieństwo, że algorytm zboczy do uboższego obszaru przestrzeni stanów lub będzie krążył między równoważnymi wagami, podążając jedynie za przypadkowym przypadkiem. Technika ta wspiera losowość, jak widzieliśmy w rozdziale 7.2.4 o symulowanym wyżarzaniu, niewielka losowość może pomóc w uniknięciu lokalnych minimów. Ale im więcej losowości, tym wolniej algorytm zbliża się do najlepszych

wag. W praktyce oba podejścia są często łączone. System nauczy się wag, grając przeciwko sobie, ale także pozwoli na drobne poprawki tych wag za pomocą wyszukiwania z wyprzedzeniem. Wyszukiwanie z osadzaniem jako część procesu uczenia się ma również zastosowanie, gdy nie stosujemy prostszych strategii tworzenia elementów, jak zobaczymy w następnej sekcji.

Sieci neuronowe i głębokie uczenie

W poprzedniej części opisałem uczenie się funkcji oceny statycznej z wektora ocen poszczególnych strategii. Każda ocena komponentu działa na bieżącej pozycji planszy, aby zwrócić liczbę. W wielu przypadkach jest to najbardziej praktyczne, ponieważ pozwala wiedzy eksperckiej przyczynić się do jakości sztucznej inteligencji. Jednak w przypadku sieci neuronowych, a zwłaszcza nadejścia głębokiego uczenia, te pośrednie oceny można usunąć, a sztuczna inteligencja może otrzymać zadanie uczenia się funkcji oceny bezpośrednio z pozycji tablicy. TD-Gammon, odnoszący sukcesy Backgammon grający w sztuczną inteligencję z lat 90., z powodzeniem zastosował to podejście. Zastosowano uczenie różnic czasowych (TD), rodzaj sieci neuronowej. Jednak losowość Backgammona wydawała się szczególnie pasować do tego podejścia, a próby zastosowania jej w innych grach – szczególnie doskonałych grach informacyjnych – były mniej udane. Dopiero pojawienie się gry Deep Mind's Go ze sztuczną inteligencją, której kulminacją było AlphaZero (zero dla „wiedzy zerowej": tj. uczy się wszystkiego od zera), pokazało, że to podejście może z łatwością przewyższyć zestaw strategii zoptymalizowanych pod kątem rąk zarówno w Go, jak i w szachach. Alpha Go Zero wykorzystuje głęboką sieć neuronową, składającą się z wielu warstw filtrów konwolucyjnych, które pobierają dane o płytce i konwertują je na parametry. Jest to podejście podobne do opisanego w rozdziale 6, wykorzystujące filtry pochodzące z przetwarzania obrazu do obliczania informacji taktycznych. Specyficzna architektura sieci jest opisana w artykule Alpha Go, sama w sobie nie jest ani rewolucyjna, ani szczególnie niezwykła. Innowacja polega na sposobie uczenia sieci, jak pokazano na rysunku.



Sieć przyjmuje reprezentację tablicy i zwraca funkcję oceny. Zwraca również kolejność możliwych ruchów (przypomnijmy, że widzieliśmy, że kolejność ruchów ma znaczenie dla wydajności wyszukiwania). Reprezentują one prawdopodobieństwo wybrania każdego ruchu. Algorytm wielokrotnie odtwarza się za pomocą sieci neuronowej. Przy każdym ruchu wykonuje przeszukiwanie drzewa Monte Carlo. Wyszukiwanie drzew oblicza kolejność ruchów, czyli ich względną jakość. Całkowita wartość aktualnej pozycji nie jest potrzebna. Sieć jest następnie szkolona tak, aby kolejność ruchów, które generuje, była bardziej zgodna z kolejnością ruchów zwracanych przez wyszukiwanie w drzewie. Pośrednio poprawia to funkcję oceny, ale pozwala systemowi uczyć się znacznie szybciej, ponieważ dopasowuje większy zestaw kryteriów, ułatwiając zaliczenie zadania przypisania. Alpha Go Zero reprezentuje obecny stan wiedzy w grach planszowych AI. Artykuł zawiera wystarczająco dużo szczegółów, aby można go było odtworzyć za pomocą jednego z bezpłatnych zestawów narzędzi do głębokiego uczenia się (takich jak Keras i własny TensorFlow firmy Google). Kod źródłowy kilku replik jest dostępny w serwisie GitHub, ale w chwili pisania tego artykułu żadna z nich nie osiągnęła takiego

samego poziomu wydajności, jak podano w artykule. Nie wysłałem jeszcze sztucznej inteligencji do gry planszowej przy użyciu tej techniki, ale lubiłem z nią eksperymentować. I chociaż nie mogę twierdzić, że zbudowałem coś tak zdumiewającego, jak mistrz świata Go AI, stwierdziłem, że stosunkowo łatwo jest stworzyć systemy, które pokonają mnie w prostszych grach, takich jak Reversi i Atoms. To ekscytujący czas w grze planszowej AI, ale pole jest w szybkim tempie. Nie jest jeszcze jasne, na przykład, czy te podejścia całkowicie wyprą silniki ręcznie dostrajane, oparte na wiedzy eksperckiej. Chociaż opieram swoją opinię wyłącznie na intuicji, podejrzewam, że połączenie głębokiego uczenia się wiedzy zerowej i wiedzy eksperckiej może okazać się jeszcze silniejsze.