Rozdział 5

Gry

W którym rozważamy problemy pojawiające się, gdy próbujemy planować z góry w świecie zawierającym wrogiego agenta.

5.1 Wprowadzenie: gry jako problemy szukania

Gry angażowały zdolności intelektualne ludzi - czasami do alarmującego stopnia - tak długo, jak istnieje cywilizacja. Gry planszowe, takie jak szachy czy Go, są interesujące, ponieważ oferują czystą, abstrakcyjną rozgrywkę, bez awantur oraz zaprzątania sobie głowy zbieraniem armii i prowadzeniem wojny. To właśnie abstrakcja czyni gry pociągającym celem badań AI. Stan gry jest łatwy do reprezentacji, zaś agenci są zwykle ograniczeni do dość ubogiej liczby dobrze zdefiniowanych działań. To czyni grę idealizacją światów, w których wrodzy agenci działają na rzecz zniszczenia czyjegoś dobra. Mniej abstrakcyjne gry, jak krykiet czy piłka nożna, nie przyciągnęły dużego zainteresowania społeczeństwa AI.

Gry to również jeden z najstarszych terenów wysiłków w sztucznej inteligencji. W 1950, prawie tak szybo jak komputery stały się programowalne, pierwsze programy szachowe zostały napisane przez Claude'a Shannona (człowieka, który wymyślił teorię informacji) i przez Alana Turinga. Od wtedy następował stabilny postęp w standardzie gry, do punktu, w którym obecne systemy potrafią sprzeciwić się ludzkiemu mistrzowi świata bez strachu lub zakłopotania.

Wcześni badacze wybrali szachy z wielu powodów. Komputer grający w szachy byłby dowodem istnienia maszyny wykonującej coś myślącego, wymagającego inteligencji. Co więcej, prostota zasad i fakt, że stan świata jest w pełni dostępny dla programu [tzn. agent może postrzegać wszystko, co wiadomo o środowisku; w teorii gier szachy są grą o doskonałej informacji] oznacza, że łatwo jest reprezentować grę jako szukanie w przestrzeni możliwych pozycji. Komputerowa reprezentacja gry właściwie może być poprawna w każdym istotnym szczególe - w odróżnieniu od reprezentacji problemu prowadzenia wojny, na przykład.

Obecność przeciwnika czyni problem decyzyjny trochę bardziej skomplikowanym niż problemy szukania opisane w rozdziale 3. Przeciwnik wprowadza niepewność, ponieważ nigdy nie wiadomo, gdzie zamierza pójść. Wszystkie programy grające w gry muszą radzić sobie z problemem sytuacji awaryjnej zdefiniowanej w rozdziale 3. Niepewność nie jest taka jak wprowadzona przez, powiedzmy, rzucanie kośćmi lub przez pogodę. Przeciwnik będzie próbował tak szybko jak to możliwe wykonać najmniej niezłośliwy ruch, podczas gdy o kościach i pogodzie zakładamy (być może błędnie), że są obojętne wobec celów agenta. Tę komplikację opisano w sekcji 5.2.

Ale tym, co czyni gry naprawdę różne jest fakt, że są one zwykle o wiele za trudne do rozwiązania. Na przykład szachy mają średni współczynnik rozgałęzienia w okolicach 35, a gra często daje 50 możliwości ruchów dla gracza, więc drzewo przeszukiwania ma około 35^100 węzłów (acz istnieje "tylko" około 10^40 różnych legalnych pozycji). Kółko i krzyżyk jest nudne dla dorosłych z tej przyczyny, że łatwo wyznaczyć właściwy ruch. Złożoność gier wprowadza zupełnie nowy rodzaj niepewności, jakiego jeszcze nie widzieliśmy; niepewność nie pojawia się ze względu na brak informacji, ale ponieważ nie mamy czasu, by obliczyć dokładne konsekwencje każdego ruchu. Zamiast tego trzeba wykonywać najlepsze zgadywanie oparte na przeszłym doświadczeniu i działać przed uzyskaniem pewności co do podejmowanego ruchu. W tym odniesieniu gry są o wiele bardziej podobne do realnego świata, niż omawiane do tej pory standardowe problemy

przeszukiwania.
Ponieważ w grach jest zwykle ograniczenie czasowe, nieudolność jest surowo karana. Podczas gdy implementacja przeszukiwania A*, które jest 10% mniej efektywne, będzie kosztować odrobinkę więcej działania do zakończenia, program szachowy będący 10% mniej efektywny w używaniu dostępnego czasu prawdopodobnie zostanie wgnieciony w ziemię, a inne rzeczy pozostaną równe. Zatem badania dotyczące gier zapoczątkowały wiele ciekawych idei związanych z najlepszym użyciem czasu by uzyskać dobre decyzje, gdy osiąganie optymalnych decyzji jest niemożliwe. Należy pamiętać o tych pomysłach podczas czytania reszty tej książki, ponieważ problemy złożoności pojawiają się w każdej dziedzinie AI. Wrócimy do nich w rozdziale 16.
Rozpoczynamy naszą dyskusję od analizy, jak znaleźć teoretycznie najlepszy ruch. Następnie patrzymy na techniki wyboru dobrego ruchu przy ograniczonym czasie. Przycinanie pozwala nam ignorować części drzewa poszukiwań nie mające wpływu na ostateczny wybór, a heurystyczne funkcje oceny pozwalają szacować prawdziwą użyteczność stanu bez wykonywania pełnego szukania. Sekcja 5.5 omawia gry takie jak backgammon, które zawierają elementy ryzyka. Na koniec patrzymy, jak najlepsze programy grające w gry radzą sobie z silnym przeciwnikiem ludzkim.

5.2 Doskonałe decyzje w grach dwuosobowych

Teraz rozważymy ogólny przypadek gier z dwoma graczami, których będziemy nazywać MAX i MIN, z przyczyn, które niedługo okażą się oczywiste. MAX rusza pierwszy, a następnie obaj wykonują ruchy, dopóki gra się nie skończy. Na końcu gry punkty są przyznawane zwycięzcy (lub czasami kary dla przegranego). Gra może zostać formalnie zdefiniowana jako rodzaj problemu przeszukiwania z następującymi składnikami:
- Stan początkowy, zawierający pozycję na planszy i wskaźnik, czyj jest ruch.
- Zbiór operatorów, które definiują legalne ruchy możliwe do wykonania przez gracza.
- Ostateczny test, który wyznacza, kiedy gra jest skończona. Stany, w których gra się skończyła, nazywamy stanami ostatecznymi.
- Funkcja użyteczności (zwana też funkcją wypłaty), która zwraca numeryczną wartość rezultatu gry. W szachach rezultat to zwycięstwo, przegrana lub remis, reprezentowane przez wartości +1, -1 lub 0. Niektore gry mają szerszą gamę możliwych wyników, np. wypłaty w backgammonie są z zakresu +192 do -192.
Jeśli byłby to normalny problem szukania, to wszystko, co musiałby robić MAX, to szukanie ciągu ruchów prowadzących do stanu ostatecznego, w którym by wygrał (zgodnie z funkcją użyteczności), a wtedy iść dalej i zrobić pierwszy ruch w ciągu. Niestety, MIN też ma coś do powiedzenia. Zatem MAX musi znaleźć strategię prowadzącą do zwycięskiego stanu ostatecznego bez względu na to, co robi MIN, przy czym startegia zawiera właściwy ruch dla MAX dla każdego możliwego ruchu MIN. Rozpoczniemy od pokazania, jak znaleźć optymalną (i racjonalną) strategię, nawet jeśli normalnie nie będziemy mieli wystarczająco dużo czasu do jej policzenia.
Rys. 5.1 pokazuje część drzewa przeszukiwania dla gry w kółko i krzyżyk. Z początkowego stanu, MAX ma wybór dziewięciu możliwych ruchów. Gra następuje naprzemian pomiędzy MAX kładącym krzyżyki i MIN kładącym kółka, dopóki nie osiągniemy liści odpowiadających stanom ostatecznym: stany, gdzie jeden gracz ma trzy w rzędzie lub wszystkie kwadraty są zapełnione. Numer każdego liścia wskazuje wartość użyteczności stanu ostatecznego z punktu widzenia MAXa; wysokie wartości są dobre dla MAXa i złe dla MINa (stąd nazwy graczy). Zadaniem MAXa jest używać drzewa przeszukiwania (w szczególności użyteczności stanów ostatecznych), by wyznaczyć najlepszy ruch.

Nawet tak prosta gra jak kółko i krzyżyk jest zbyt skomplikowana, by pokazać całe drzewo przeszukiwania, więc zajmiemy się absolutnie trywialną grą na rys. 5.2. Możliwe ruchy dla MAX są oznaczone A1, A2, A3. Możliwe odpowiedzi na A1 dla MIN to A11, A12, A13, itd. Ta szczególna gra kończy się po jednym ruchu z każdej strony. (W żargonie gier mówimy, że to drzewo jest głębokości 1, składające się z dwóch pół-ruchów). Użyteczności stanów ostatecznych w tej grze są z zakresu 2 do 14.
Algorytm minimax jest zaprojektowany, by wyznaczać optymalną strategię dla MAX, a więc decydować, jaki jest najlepszy ruch. Algorytm składa się z pięciu kroków:
- Wygeneruj całe drzewo gry, w dół do stanów ostatecznych.
- Zastosuj funkcję użyteczności dla każdego stanu ostatecznego, by uzyskać jego wartość.
- Użyj użyteczności stanów ostatecznych do wyznaczenia użyteczności węzłów jeden poziom wyżej w drzewie przeszukiwania. Rozważ trzy liście z lewej strony na rys. 5.2. W węźle V nad nim, MIN ma opcję ruchu, i najlepszą opcją dla MIN jest wybrać A11, co prowadzi do minimalizacji wyniku, 3. Zatem mimo, że funkcja użyteczności nie jest od razu stosowalna do tego węzła V, możemy przypisać mu wartość użyteczności 3, z założeniem, że MIN zrobi, co trzeba. Z podobnym rozumowaniem, inne dwa węzły V dostają wartość użyteczności 2.
- Kontynuuj gromadzenie wartości z liści w kierunku korzenia, jedna warstwa na raz.
- Na końcu zgromadzone wartości osiągają szczyt drzewa; w tym punkcie MAX wybiera ruch prowadzący do najwyższej wartości. W najbardziej szczytowym węźle A (rys. 5.2) MAX ma wybór trzech ruchów prowadzących do stanów z użytecznością 3, 2 i 2 odpowiednio. A więc najlepszy otwierający ruch MAXa to A11. Jest to tzw. decyzja minimaksowa, ponieważ maksymalizuje użyteczność przy założeniu, że przeciwnik zagra perfekcyjnie, by ją zminimalizować.
Rys. 5.3 pokazuje bardziej formalny opis algorytmu minimax. Funkcja z najwyższego poziomu, MINIMAX-DECISION, wybiera z dostępnych ruchów, które są obliczane kolejno przez funkcję MINIMAX-VALUE.
Jeśli maksymalna głębokość drzewa to m, zaś jest b legalnych ruchów w każdym punkcie, to złożoność czasowa algorytmu minimax wynosi O(b^m). Algorytm jest przeszukaniem depth-first (acz tutaj implementacja jest przez rekursję, a nie przez kolejkę węzłów), więc wymagania pamięciowe są tylko liniowe w m i b. Dla prawdziwych gier koszt czasu jest zupełnie niepraktyczny, ale algorytm służy jako podstawa dla bardziej realistycznych metod i dla matematycznej analizy gier.

5.3 Niedoskonałe decyzje

Algorytm minimax zakłada, że program ma czas na przeszukanie całej drogi do stanu ostatecznego, co jest zwykle niepraktyczne. Oryginalny papier Shannon'a o szachach proponował, żeby zamiast przechodzenia całej drogi do stanu ostatecznego i używania funkcji użyteczności, program obcinał szukanie wcześniej i stosował heurystyczną funkcję oceny do liści drzewa. Innymi słowy, sugestia jest następująca: zmienić minimax na dwa sposoby: funkcja użyteczności jest zamieniona przez funkcję oceny EVAL, a test ostateczności jest zastąpiony przez test obcięcia CUTOFF-TEST.

Funkcje oceny.

Funkcja oceny zwraca oszacowanie oczekiwanej użyteczności gry z danej pozycji. Idea nie była nowa, gdy proponował ją Shannon. Przez wieki gracze w szachy (i oczywiście fani innych gier) rozwinęli sposoby oceniania szans na zwycięstwo każdej strony bazujące na łatwo obliczanych

cechach pozycji. Na przykład podręczniki wstępne do szachów dają przybliżone wartości materialne każdej figury: pion jest warty 1, skoczek lub goniec 3, wieża 5, hetman 9. Inne cechy, jak "dobra struktura pionowa" i "bezpieczeństwo króla" mogą być warte, powiedzmy, pół piona. Gdy wszystkie inne rzeczy są równe, strona posiadająca bezpieczną przewagę materialną piona lub więcej prawdopodobnie wygra grę, a 3-punktowa przewaga jest wystarczająca dla prawie pewnego zwycięstwa. Rys. 5.4 pokazuje cztery pozycje z ich ocenami.

Powinno być jasne, że wydajność programu grającego w grę jest zależna od jakości funkcji oceny. Jeśli jest ona niedokładna, doprowadzi program do pozycji wyglądających na "dobre", ale faktycznie katastrofalnych. Jak dokładnie mierzymy jakość?

Po pierwsze, funkcja oceny musi się zgadzać z funkcją użyteczności na stanach ostatecznych. Po drugie, nie może trwać zbyt długo! (Jak wspomniano w rozdziale 4, jeśli nie narzuciliśmy ograniczeń na jej złożoność, mogłaby wywołać minimax jako podprocedurę i obliczyć dokładną wartość pozycji.) Zatem istnieje zamiana pomiędzy dokładnością funkcji oceny i kosztem czasu. Po trzecie, funkcja oceny powinna dokładnie odzwierciedlać rzeczywiste szanse zwycięstwa.

Można by się zastanawiać nad frazą "szanse zwycięstwa". Jakby nie było, szachy nie są grą ryzyka. Ale jeśli obetniemy wyszukiwanie w pewnym stanie nieostatecznym, nie wiemy co się stanie w następnych ruchach. Dla dokładności, załóżmy że funkcja oceny liczy tylko wartości materialne. Wtedy, w pozycji otwierającej, ocena jest równa 0, ponieważ obie strony mają to samo. Wszystkie pozycje do pierwszego przejęcia będą również mieć ocenę równą 0. Jeśli MAX potrafi przechwycić gońca bez straty figury, to wynikowa pozycja będzie mieć wartość oceny równą 3. Ważna uwaga: dana wartość oceny pokrywa wiele różnych pozycji - wszystkie pozycje, gdzie MAX jest do przodu o gońca, są zgrupowane razem w kategorię z metką "3". Teraz widzimy, jaki jest sens słowa "szansa": funkcja oceny powinna odzwierciedlać szansę, że wybrana pozycja (w sposób losowy) z takiej kateogrii prowadzi do zwycięstwa (lub remisu, lub porażki), na podstawie poprzedniego doświadczenia.

To sugeruje, że funkcja oceny powinna być określona przez zasady prawdopodobieństwa: jeśli pozycja A ma 100% szans na wygranie, powinna mieć ocenę 1.00, a jeśli pozycja B ma szansę 50% na wygranie, 25% na przegranie i 25% na remis, funkcja oceny powinna być +1 x .50 + -1 x .25 + 0 x .25 = .25. Ale nie musimy być tak precyzyjni; właściwe numeryczne wartości funkcji oceny nie są ważne, dopóki A jest w rankingu wyżej niż B.

Funkcja oceny licząca przewagę materialną zakłada, że wartość figury może być osądzona niezależnie od innych figur obenych na planszy. Ten rodzaj funkcji oceny nazywamy ważoną funkcją liniową, ponieważ można ją wyrazić jako $w1f1 + w2f2 + ... + wnfn$, gdzie w to wagi, a f to cechy poszczególnych pozycji. W to byłyby wartości figur (1 dla piona, 3 dla gońca, itd.), zaś f byłyby numerami każdego rodzaju figury na planszy. Teraz widzimy, skąd pochodzą określone wartości figur: dają one najlepsze przybliżenie prawdopodobieństwa wygrania w indywidualnych kategoriach.

Większość programów grających w gry używa liniowych funkcji oceny, acz ostatnio nieliniowe funkcje osiągnęły sukces. (Rozdział 19 daje przykład sieci neuronowej, która jest wytrenowana, by uczyć się nieliniowej funkcji oceny dla backgammona.) Przy konstruowaniu liniowego wzoru trzeba najpierw wskazać cechy, a potem dostosować wagi, dopóki program nie gra dobrze. To ostatnie zadanie można zautomatyzować przez posiadanie programu grającego dużo przeciwko sobie, ale póki co, nikt nie ma dobrego pomysłu, jak automatycznie wskazywać dobre cechy.

Przycinanie szukania

Najprostszym podejściem do kontrolowania ilości szukania jest ustawienie ustalonego limitu głębokości, tak że test obcięcia jest pozytywny dla wszystkich węzłów na lub poniżej głębokości

d. Głębokość jest tak dobierana, żeby ilość użytego czasu nie przekroczyła tego, na co pozwalają zasady gry. Trochę silniejsze podejście to zastosowanie iteratywnego zagłębiania, zdefiniowane w rozdziale 3. Gdy kończy się czas, program zwraca ruch wybrany na najgłębszym ukończonym poszukiwaniu.

Te podejścia mogą mieć pewne katastrofalne konsekwencje ze względu na przybliżoną naturę funkcji oceny. Rozważmy ponownie prostą funkcję oceny dla szachów bazującą na przewadze materialnej. Przypuśćmy, że program szuka do granicy głębokości, osiągając pozycję pokazaną na rys. 5.4(d). Zgodnie z funkcją materialną, biały jest do przodu o skoczka i prawie pewny zwycięstwa. Jednak, skoro jest ruch czarnych, biały hetman jest stracony, ponieważ czarny skoczek może go zdjąć bez straty na rzecz białych. Zatem w rzeczywistości pozycja jest wygrana dla czarnych, ale można to zobaczyć patrząc do przodu jeden ruch więcej.

Oczywiście, potrzebny jest bardziej złożony test obcięcia. Funkcja oceny powinna być stosowana tylko do pozycji stabilnych [quiescent], tj. nie zapowiadających okazania dzikich wahań wartości w najbliższej przyszłości. W szachach na przykład pozycje, w których możliwe są korzystne przejęcia, nie są stabilne wobec funkcji liczącej tylko wartość materialną. Niestabilne pozycje można rozszerzać, dopóki pozycje stabilne nie zostaną osiągnięte. To dodatkowe wyszukiwanie nazywamy przeszukiwaniem stanów stabilnych [quiescence search]. Czasami jest ono ograniczone, by rozważać tylko określone typy ruchów, jak ruchy przejmujące, które szybko rozwiążą niepewności w pozycji.

Trudniejszy do eliminacji jest problem horyzontu. Pojawia się on, gdy program dostaje ruch przeciwnika, który powoduje poważną stratę i jest nieunikniony. Rozważmy szachy na rys. 5.5. Czarny jest lekko do przodu z materiałem, ale jeśli białe awansują piona z siódmej linii, stanie się on hetmanem i będzie łatwo wygrać białym. Czarne mogą opóźnić to o kilka ruchów szachując wieżą, ale pion w sposób nieunikniony stanie się królową. Problem z szukaniem o określonej głębokości jest taki, że wierzy ono, że te ruchy spowodują uniknięcie zdobycia hetmana - mówimy, że opóźniające ruchy pchają nieunikniony ruch "nad horyzontem" do miejsca, gdzie nie można tego wykryć. Obecnie nie jest znane ogólne rozwiązanie problemu horyzontu.

5.4 Przycinanie alpha - beta

Załóżmy, że zaimplementowaliśmy wyszukiwanie minimax z rozsądną funkcją oceny dla szachów i z rozsądnym testem przycięcia, z wyszukiwaniem stanów stabilnych. Z dobrze napisanym programem na zwykłym komputerze można prawdopodobnie przeszukać 1000 pozycji na sekundę. Jak dobrze będzie grał nasz program? W szachach turniejowych mamy około 150 sekund na ruch, więc możemy popatrzeć na 150 tys. pozycji. W szachach współczynnik rozgałęzienia wynosi ok. 35, więc nasz program będzie w stanie patrzeć do przodu tylko trzy lub cztery ruchy i będzie grał na poziomie kompletnego nowicjusza. Nawet przeciętni ludzcy gracze mogą planować sześć lub osiem ruchów do przodu, więc nasz program zostanie łatwo ograny.

Na szczęście można liczyć poprawne decyzje minimaksowe bez patrzenia na każdy węzeł w drzewie przeszukiwania. Proces eliminacji gałęzi z rozważanego drzewa przeszukiwania bez sprawdzania nazywamy przycinaniem drzewa wyszukiwania. Szczególna technika, jaką rozważamy, jest nazywana przycinaniem alpha - beta. Gdy zastosować ją do standardowego drzewa minimax, zwraca ten sam ruch co minimax, ale obcina gałęzie, które nie mogą wpłynąć na ostateczną decyzję.

Rozważmy dwuruchową grę z rys. 5.2 pokazaną ponownie na rys. 5.6. Przeszukiwanie działa jak poprzednio: A1, A11, A2, A13, a węzeł pod A1 dostaje wartość minimaksową 3. Teraz następuje A2 i A21, który ma wartość 2. W tym punkcie zdajemy sobie sprawę, że jeśli MAX gra A2, MIN

ma opcję osiągnięcia pozycji wartej 2 i pewne inne opcje poza tym. Zatem możemy już powiedzieć, że ruch A2 jest warty co najwyżej 2 dla MAX. Ponieważ wiemy już, że ruch A1 jest warty 3, nie ma sensu patrzeć dalej pod A2. Innymi słowy, możemy przyciąć drzewo poszukiwań w tym miejscu i ufać, że przycinanie nie będzie miało wpływu na wynik.
Ogólna zasada jest następująca. Rozważmy węzeł n gdzieś w drzewie (rys. 5.7), tż. Player ma wybór ruszenia do tego węzła. Jeśli Player ma lepszy wybór m czy to w węźle przodka n, czy to w dowolnym punkcie wyboru wyżej, to n nigdy nie zostanie osiągnięty w faktycznej grze. Więc jak dowiedzieliśmy się dostatecznie dużo o n (przez badanie niektórych jego potomków), by uzyskać tę konkluzję, możemy go przyciąć.
Pamiętajmy, że przeszukiwanie minimax jest w głąb, więc w dowolnym momencie musimy rozważyć węzły wzdłuż pojedynczej ścieżki w drzewie. Niech a będzie wartością najlepszego wyboru znalezionego do tej pory w dowolnym punkcie wyboru wzłuż ścieżki dla MAX, a b wartością najlepszego (tj. o najmniejszej wartości) wyboru znalezionego do tej pory w dowolnym punkcie wyboru wzdłuż ścieżki dla MIN. Przeszukiwanie alpha-beta aktualizuje wartości a i b podczas działania oraz przycina drzewo (tj. kończy rekursywne wywołanie) gdy tylko wiadomo, że jest gorsze od obecnej wartości a lub b.
Opis algorytmu na rys. 5.8 jest podzielony na funkcje MAX-VALUE i MIN-VALUE. Stosują się one odpowiednio do węzłów MAX i MIN, ale każda robi to samo: zwraca wartość minimax węzła, z wyjątkiem węzłów do obcięcia (w takim przypadku zwracana wartość i tak jest ignorowana). Funkcja szukania alpha-beta jest kopią funkcji MAX-VALUE z dodatkowym kodem do pamiętania i zwracania najlepszego znalezionego ruchu.

Efektywność przycinania alfa-beta
Efektywność alfa-beta zależy od kolejności, w której badane są następniki. Jest to jasne z rys. 5.6, gdzie nie mogliśmy przyciąć A3, poniważ A31 i A32 (najgorsze ruchy z punktu widzenia MIN) były wygenerowane wcześniej. To sugeruje, że może warto by było zbadać pierwsze następniki, które mają szansę być najlepszymi.
Jeśli założymy, że można to zrobić, wychodzi, że alfa-beta potrzebuje tylko O(b^(d/2)) węzłów do wybrania najlepszego ruchu, zamiast O(b^d) w minimax. To oznacza, że efektywny współczynik rozgałęzienia jest równy sqrt(b) zamiast b - dla szachów 6 zamiast 35. Innymi słowy oznacza to, że alfa-beta może patrzeć do przodu dwa razy dalej niż minimax tym samym kosztem. Więc przez generowanie 150000 węzłów w przedziale czasu, program może przewidzieć 8 ruchów zamiast czterech. Myśląc uważnie, jakie obliczenia właściwie wpływają na decyzję, jesteśmy w stanie przekształcić program z nowicjusza w eksperta.
Efektywność przycinania alfa-beta była analizowana głęboko przez Knutha i Moore'a (1975). Oprócz najlepszego przypadku opisanego w poprzednim paragrafie, analizowali oni przypadek, w którym następniki są uporządkowane losowo. Okazuje się, że złożoność asymptotyczna jest równa O((b/logb)^d), co wydaje się żałosne, ponieważ efektywny współczynnik rozgałęzienia b/logb nie jest wiele mniejszy niż b. Z drugiej strony, wzór asymptotyczny jest dokładny tylko dla b > 1000 (około) - innymi słowy, nie dla gier, w które możemy rozsądnie grać używając tych technik. Dla rozsądnego b pełna liczba badanych węzłów będzie ok. O(b^(3d/4)). W praktyce dość prosta funkcja porządkująca (próba przejęcia najpierw, wtedy zagrażanie, wtedy ruchy do przodu, wtedy ruchy do tyłu) przybliża nas do wyniku z najlepszego przypadku, a nie do wyniku losowego. Inne popularne podejście to wykonanie iteracyjnego zagłębiania szukania i użycia zachowanych wartości z jednej iteracji do wyznaczenia uporządkowania następników w następnej iteracji.
Warto też zauważyć, że wszystkie wyniki o złożoności w grach (i ogólniej, o problemach szukania) muszą zakładać wyidealizowany model drzewa, by uzyskać ich wyniki. Na przykład model użyty

dla alfa-beta w poprzednim akapicie zakłada, że wszystkie węzły mają ten sam współczynnik rozgałęzienia b; że wszystkie ścieżki osiągają ustalony limit głębokości d; i że oceny liści są losowo rozprzestrzenione na ostatniej warstwie drzewa. To ostatnie założenie jest poważnie dziurawe: na przykład, jeśli ruch w górę drzewa jest katastrofalną gafą, to większość jego potomków będzie wyglądać źle dla gracza, który popełnił gafę. Wartość węzła jest więc prawdopodobnie wysoce skorelowana z wartościami rodzeństwa. Rozmiar korelacji zależy mocno od określonej gry i pozycji w korzeniu. Zatem mamy nieunikniony składnik nauki empirycznej zaangażowany w badanie nad grami, uciekający od potęgi analizy matematycznej.

5.5 Gry zawierające element ryzyka

W prawdziwym życiu, nie jak w szachach, istnieje wiele nieprzewidywalnych zewnętrznych zdarzeń, które stwarzają nieprzewidziane sytuacje. Wiele gier odzwierciedla tę nieprzewidywalność przez zawarcie losowych elementów, jak rzucenie kością. W ten sposób stawiają krok w stronę rzeczywistości. Warto zobaczyć, jak to wpływa na proces podejmowania decyzji.

Backgammon jest typową grą, która łączy szczęście i umiejętności. Kości są toczone na początku kolejki gracza, by wyznaczyć zbiór legalnych ruchów dostępnych dla gracza. W pozycji na rys. 5.9 biały wyrzucił 6-5 i ma cztery możliwe ruchy.

Białe wiedzą, jakie mają legalne ruchy, ale nie wiedzą, co wyrzuci czarny, a więc nie wiedzą, jakie będą legalne ruchy czarnego. To znaczy, że białe nie mogą skonstruować pełnego drzewa gry w rodzaju tego z szachów czy kółka i krzyżyka. Drzewo gry w backgammonie musi zawierać węzły ryzyka poza węzłami MIN i MAX. Węzły ryzyka są pokazane jako okręgi w rys. 5.10. Gałęzie prowadzące z każdego węzła ryzyka oznaczają możliwe rzuty kośćmi, a każdy jest oznaczony rzutem i szansą, że wypadnie. Jest 36 sposobów rzucenia kośćmi, każdy równie prawdopodobny; ale ponieważ 6-5 jest takie samo jak 5-6, jest tylko 21 różnych rzutów. Sześć dwójek (1-1 do 6-6) mają 1/36 szansy wypadnięcia, pozostałe 15 różnych rzutów ma szansę 1/18 (każdy).

Następny krok to zrozumienie, jak podejmować poprawne decyzje. Oczywiście, ciągle chcemy wybrać ruch z A1, ..., An, który prowadzi do najlepszej pozycji. Jednakże każda z możliwych pozycji nie ma już jasnej wartości minimax (która w deterministycznych grach była użytecznością liścia osiągniętego przez najlepszą grę). Zamiast tego możemy tylko liczyć średnią lub oczekiwaną wartość, gdzie średnia jest wzięta nad wszystkimi możliwymi rzutami kością, jakie mogą paść.

Prosto jest policzyć wartości oczekiwanych węzłów. Dla węzłów ostatecznych używamy funkcji użyteczności, jak w grach deterministycznych. Idąc krok wyżej w drzewie, trafiamy na węzeł ryzyka. Na rys. 5.10 węzły ryzyka to okręgi; rozważymy ten oznaczony C. Niech di będzie możliwym rzutem kością, a P(di) szansą lub prawdopodobieństwem uzyskania tego rzutu. Dla każdego rzutu liczymy użyteczność najlepszego ruchu dla MIN, a wtedy dodajemy użyteczności, ważone przez szanse, że uzyskamy określony rzut. Jeśli S(C, di) oznacza zbiór pozycji wygenerowanych przez zastosowanie legalnych ruchów dla rzutu P(di) do pozycji C, to możemy policzyć tzw. wartość expectimax C używając wzoru

expectimax(C) = suma P(di) max s /in S(C, di) (utility(s)).

To daje oczekiwaną użyteczność pozycji w C zakładającą najlepszy ruch. Idąc jeden poziom dalej do węzłów MIN, możemy zastosować normalny wzór na wartość minimax, ponieważ przypisaliśmy wartości użyteczności wszystkim węzłom ryzyka. Wtedy idziemy wyżej do węzła ryzyka B, gdzie liczymy wartość expectimin używając analogicznego wzoru jak ten dla expectimax.

Ten proces można powtarzać rekurencyjnie całą drogę w górę drzewa, z wyjątkiem szczytowego poziomu, gdzie rzut kością jest już znany. By policzyć najlepszy ruch, po prostu zamieniamy MINIMAX-VALUE w rys. 5.3 na EXPECTIMINIMAX-VALUE, którego implementację zostawiamy jako ćwiczenie.

Ocena pozycji w grach z węzłami ryzyka

Tak jak w minimaksie, oczywiste przybliżenie czynione z expectiminimax to przycięcie szukania w jakimś punkcie i zastosowanie funkcji oceny do liści. Możnaby myśleć, że funkcje oceny dla gier jak backgammon nie są różne, co do zasady, od funkcji oceny dla szachów - powinny po prostu dawać wyższe punkty lepszym pozycjom.

W rzeczywistości obecność węzłów ryzyka oznacza, że trzeba być ostrożnym co do tego, co oznacza wartość oceny. Pamiętajmy, że dla minimaksu dowolna transformacja wartości liści zachowująca porządek nie wpływa na wybór ruchu. Możemy więc używać albo wartości 1, 2, 3, 4, albo 1, 20, 30, 400 i otrzymać tę samą decyzję. To daje swobodę w projektowaniu funkcji oceny: będzie działać poprawnie, dopóki pozycje z wyższymi ocenami prowadzą do zwycięstwa częściej (średnio).

Z węzłami ryzyka tracimy tę swobodę. Rys. 5.11 pokazuje, co się dzieje: z wartościami liści 1, 2, 3, 4, ruch A1 jest najlepszy; z wartościami liści 1, 20, 30, 400, ruch A2 jest najlepszy. Zatem program zachowuje się zupełnie inaczej, jeśli zmienimy skalę wartości oceny! Okazuje się, że aby uniknąć tej wrażliwości, funkcja oceny może być tylko dodatnią liniową transformacją prawdopodobieństwa zwycięstwa z pozycji (lub, bardziej ogólnie, oczekiwaną użytecznością pozycji). To ważna i ogólna własność sytuacji, w której niepewność bierze udział, i piszemy o tym dalej w rozdziale 16.

Złożoność expectiminimaxu

Jeśli program wiedziałby z góry wszystkie rzuty kośćmi, jakie wypadną w reszcie gry, rozwiązanie gry z kością byłoby jak rozwiązanie gry bez kości, co minimax robi w czasie O(b^m). Ponieważ expectiminimax rozważa też możliwe ciągi rzutów kośćmi, zajmie O(b^m * n^m), gdzie n jest liczbą odrębnych rzutów.

Nawet jeśli ograniczyć głębokość drzewa do pewnej małej głębokości d, dodatkowy koszt porównany do minimaxu czyni nierealnym rozważanie patrzenia daleko w przód w grach jak backgammon, gdzie n jest równe 21, zaś b ok. 20, ale w pewnych sytuacjach może wynieść nawet 4000. Zapewne nie ogarniemy więcej niż 2 ruchy.

Inny sposób myślenia o problemie jest następujący: zaleta alfa-bety jest taka, że ignoruje on przyszłościowy rozwój, który nie nastąpi, przy danej najlepszej grze. Zatem koncentruje się na prawodpodobnych zdarzeniach. W grach z kośćmi nie ma prawodpodobnych ciągów ruchów, ponieważ aby one mogły nastąpić, kość musiałaby wypaść we właściwy sposób, by te ruchy były legalne. To ogólny problem, gdy niepewność wkracza do obrazka: możliwości mnożą się nienaturalnie, a formowanie szczegółowych planów działania staje się bezsensowne, ponieważ świat prawdopodobnie nie będzie grał długo.

Bez wątpienia czytelnik spotka sytuację, że może coś jak obcinanie alfa-beta mogłoby być zastosowane w drzewie gry z węzłami ryzyka. Wychodzi, że może tak być, z odrobiną pomysłowości. Rozważmy szansę węzła C na rys. 5.10 i co się stanie z jego wartością gdy badamy i wyceniamy jego dzieci; pytanie: czy można znaleźć górne ograniczenie wartości C przed spojrzeniem na wszystkie dzieci? (Przypomnijmy, że alfa-beta wymaga tego, aby przyciąć węzeł i jego poddrzewo.) Na pierwszy rzut oka to może się wydawać niemożliwe, ponieważ wartość C jest średnią wartości dzieci, zaś dopóki nie spojrzymy na wszystkie rzuty kośćmi, ta średnia

może być dowolna, ponieważ niezbadane dzieci mogą mieć dowolną wartość. Ale jeśli nałożymy ograniczenia na możliwe wartości funkcji użyteczności, możemy dotrzeć do ograniczeń na średnią. Np. jeśli mówimy, że wartości użyteczności są pomiędzy +1 i -1, to wartość liści jest ograniczona, zatem możemy nałożyć górne ograniczenie na wartość węzła ryzyka bez patrzenia na wszystkie dzieci. Projektowanie procesu przycinania jest odrobinkę bardziej skomplikowane niż dla alfa-bety i zostawiamy go jako ćwiczenie.

5.6 Najlepsze programy grające

Projektowanie programów grających ma dwa cele: lepiej zrozumieć, jak wybierać działania w złożonych dziedzinach z niepewnymi wynikami i rozwinąć systemy o wysokiej wydajności dla danej gry. Badamy postęp w kierunku tego drugiego celu.

Szachy.
Szachy stanowią do tej pory największy obiekt zainteresowania w grach. Nie spełniono obietnicy uczynionej przez Simona w 1957, że za 10 lat komputery pokonają ludzkiego mistrza świata, ale obecnie są już bliskie osiągnięcia tego celu. W szachach szybkich komputery pokonały mistrza świata, Garriego Kasparowa, zarówno w grach 5-, jak i 25-minutowych, ale w pełnych grach turniejowych mają ranking pośród pierwszych 100 graczy w czasie pisania. Rys. 5.12 pokazuje notowania mistrzów ludzkich i komputerowych na przestrzeni lat. Kusi, aby spróbować extrapolować i zobaczyć, gdzie linie się przetną.
Postęp powyżej poziomu średniego był początkowo bardzo wolny: niektóre programy we wczesnych latach 1970-tych były bardzo złożone, z różnymi rodzajami sztuczek do eliminowania pewnych gałęzi przeszukiwania, do generowania możliwych ruchów, itd., ale programy, które wygrały mistrzostwa ACM używały bezpośredniego przeszukiwania alfa-beta wspartego książkowymi otwarciami i nieomylnymi algorytmami końcówek. (To daje interesujący przykład, jak wysoka wydajność wymaga hybrydowej architektury podejmującej decyzje do zaimplementowania funkcji agenta.)
Pierwszy rzeczywisty skok wydajności przyszedł nie z lepszych algorytmów czy funkcji oceny, ale z hardware'u. Belle, pierwszy komputer szachowy specjalnego przeznaczenia (Condon i Thompson, 1982) użył specjalnych obwodów scalonych do zaimplementowania generowania ruchów i oceny pozycji, pozwalając na przeszukanie szeregu milionów pozycji do uczynienia pojedynczego ruchu. Uzyskał ranking około 2250, na skali, gdzie ludzie mają 1000 i mistrz świata ok. 2750; stał się pierwszym programem mistrzowskiego poziomu.
System HITECH, również komputer specjalnego przeznaczenia, został zaprojektowany przez byłego korespondencyjnego mistrza świata Hansa Berlinera i jego studenta Carla Ebelinga; pozwalał błyskawicznie liczyć bardzo skomplikowane funkcje oceny. Generował ok. 10 milionów pozycji na ruch i używał prawdopodobnie najdokładniejszej oceny pozycji do tej pory; stał się mistrzem świata w 1985. Był pierwszym programem, który pokonał ludzkiego mistrza, Arnolda Denkera, w 1987. W swoim czasie notowany pomiędzy 800 najlepszymi graczami na świecie.
Obecnie najlepszy system to Deep Thought 2. Sponsorowany przez IBM, który zatrudnił część teamu, który stworzył Deep Thought w Uniwersytecie Carnegie Mellon. Deep Thought 2 używa prostych funkcji oceny, ale bada ok. pół miliarda pozycji na ruch, co pozwala osiągnąć głębokość 10 lub 11, ze specjalną prowizją pomijania linii wymuszonych ruchów (raz znalazł mata w 37 ruchach). W lutym 1993, Deep Thought 2 walczył z duńską drużyną olimpijską i wygrał 3:1 pokonując mistrza i remisując z innym. Ranking FIDE około 2600, miejsce w pierwszej setce ludzi.

Następna wersja systemu, Deep Blue, użyje 1024 równoległych chipów VLSI. To pozwoli przeszukać równoważność miliarda pozycji na sekundę (100-200 miliardów na ruch) i osiągnąć głębokość 14. 10-procesorowa wersja zagra przeciwko drużynie Izraela w maju 1995.

Warcaby

Rozpoczynając w 1952, Arthur Samuel z IBM, pracując w wolnym czasie, rozwinął program grający w warcaby, który uczył się własnej funkcji oceny grając przeciwko sobie tysiące razy. Program zaczął jako nowicjusz, ale po kilku dniach treningu był w stanie rywalizować w niektórych silnych ludzkich turniejach. Jeśli wziąć pod uwagę wyposażenie (IBM 704), 10000 słów pamięci, taśma magnetyczna jako pamięć masowa, czas cyklu - prawie milisekunda, to pozostanie to jednym z największych wyczynów w AI.

Kilka innych ludzi chciało być lepszymi, aż Jonathan Schaeffer i koledzy rozwinęli Chinook, działający na zwykłych komputerach i używający szukania alfa-beta, ale używający licznych technik, jak baza danych doskonałych rozwiązań dla wszystkich sześciofigurowych pozycji, co uczyniło grę końcową druzgocącą. Chinook wygrał US Open w 1992 i stał się pierwszym programem, który oficjalnie wygrał w prawdziwych mistrzostwach świata. Później walczyl ponownie z problemem, w postaci Mariona Tinsley'a. Dr Tinsley był mistrzem świata przez 40 lat, przegrywając tylko trzy gry w całym czasie. W pierwszym meczu przeciwko Chinookowi, Tinsley przegrał czwartą i piątą walkę, ale wygrał mecz 21.5 - 18.5. Bardziej niedawno, mecz mistrzostw świata w sierpniu 1994 pomiędzy Tinsleyem i Chinookiem skończył się przedwcześnie, gdy Tinsley musiał wycofać się z powodów zdrowotnych. Chinook został oficjalnym mistrzem świata.

Othello

Othello, zwany też Reversi, jest prawdopodobnie bardziej popularne jako gra komputerowa niż planszowa. Ma mniejszą przestrzeń poszukiwań niż szachy, zwykle 5 do 15 legalnych ruchów, ale wycena musiała zostać opracowana od zera. Mimo to, programy Othello na normalnych komputerach są dużo lepsze niż ludzie, którzy ogólnie odmawiają bezpośrednich walk w turniejach.

Backgammon

Jak wspomniano, włączenie niepewności z rzucania kośćmi tworzy z szukania drogi luksus w backgammonie. Pierwszy program, który miał poważny wpływ, BKG, używał tylko jednoruchowego szukania, ale bardzo skomplikowanej funkcji oceny. W nieformalnym meczu w 1980 pokonał ludzkiego mistrza świata 5-1, ale był dość szczęśliwy z kośćmi. Ogólnie, gra jak silny amator.

Bardziej niedawno Gerry Tesauro (1992) połączył metodę uczenia Samuela z technikami sieci neuronowych, by rozwinąć nową funkcję oceny. Jego program jest stabilnie notowany między najlepszymi trzema graczami na świecie.

Go

Go jest najpopularniejszą grą planszową w Japonii, wymagającą co najmniej tyle dyscypliny od graczy, co szachy. Czynnik rozgałęzienia to w przybliżeniu 360, więc zwykłe metody szukania są całkowicie przegrane. Systemy oparte na wielkich bazach wiedzy reguł do sugerowania możliwych ruchów chyba mają jakąś szansę, ale wciąż grają bardzo słabo. Wyznaczono 2 mln dolarów nagrody za pierwszy program, który pokona gracza z najwyższego poziomu - zatem Go wydaje się terenem, który zyska na intensywnych badaniach nad użyciem skomplikowanych metod rozumowania.

5.7 Dyskusja

Ponieważ obliczanie optymalnych decyzji w grach jest niepodatne w większości przypadków, wszystkie algorytmy musza czynić jakieś założenia i przybliżenia. Standardowe podejście, bazowane na minimaksie, funkcje oceny i alfa-beta to tylko jeden sposób zrobienia tego. Prawdopodobnie, ponieważ zaproponowano to wcześnie, zostało to rozwinięte intensywnie i dominuje nad innymi metodami w grze turniejowej. Niektórzy w dziedzinie wierzą, że spowodowało to rozwód gier i głównego nurtu badań AI, ponieważ standardowe podejście nie dostarcza dostatecznie dużej przestrzeni dla nowego spojrzenia na ogólne kwestie podejmowania decyzji. W tej sekcji patrzymy na alternatywy, rozważając jak zluzować założenia i może wyprowadzić nowe spojrzenie.

Po pierwsze, rozważmy minimax. Minimax jest optymalną metodą wyboru ruchu z drzewa poszukiwań pod warunkiem, że oceny liści są dokładne. W rzeczywistości, oceny są zwykle szacunkami wartości pozycji i można sądzić, żę mają duże błędy. Rys. 5.13 pokazuje dwuruchową grę (drzewo), dla którego minimax wydaje się niewłaściwy. Minimax sugeruje wzięcie gałęzi z prawej strony, podczas gdy całkiem prawdopodobne jest, że prawdziwa wartość lewej gałęzi jest wyższa. Wybór minimaksu polega na założeniu, że wszystkie węzły oznaczone wartościami 100, 101, 102 i 100 są rzeczywiście lepsze niż węzeł oznaczony 99. Jeden ze sposobów poradzenia sobie z tym problemem to posiadanie oceny, która zwraca rozproszenie prawdopodobieństwa na możliwe wartości. Wtedy można liczyć rozproszenie prawdopodobieństwa dla wartości rodziców używając standardowych technik statystycznych. Niestety, wartości rodzeństwa są zwykle wysoko skorelowane, więc może to być drogie obliczenie i może wymagać szczegółowej informacji o korelacji, co trudno uzyskać.

Następnie rozważmy algorytm przeszukiwania, który generuje drzewo. Celem projektanta algorytmu jest określić obliczenie, które kończy się w określonym czasie i zwraca dobry wybór ruchu. Najbardziej oczywistym problemem algorytmu alfa-beta jest to, że jest on zaprojektowany nie by wybrać dobry ruch, ale też obliczyć wartości wszystkich legalnych ruchów. By zobaczyć, czemu ta dodatkowa informacja jest zbędna, rozważmy pozycję, w której jest tylko jeden legalny ruch. Alfa-beta wygeneruje i wyceni duże, zupełnie bezużyteczne, drzewo szukania. Oczywiście możemy wstawić test do algorytmu, ale to tylko pozornie schowa problem - wiele obliczeń wykonywanych przez alfa-beta są w dużej mierze niepotrzebne. Posiadanie tylko jednego legalnego ruchu nie różni się wiele od posiadania wielu legalnych ruchów, z których jeden jest w porządku, a reszta oczywiście katastrofalna. W sytuacji z jasnym faworytem [clear-favorite] jak ta, byłoby lepiej osiągnąć szybką decyzję po małej ilości szukania niż tracić czas, który można by lepiej użyć później dla problematycznej pozycji. To prowadzi do pomysłu użyteczności rozszerzenia węzła. Dobry algorytm szukania powinien wybierać rozszerzenia węzła o wysokiej użyteczności - tj. takie, które prawodpodobnie prowadzi do odkrycia znacząco lepszego ruchu. Jeśli nie ma rozszerzeń węzła, których użyteczność jest wyższa niż ich koszt (w terminach czasu), to algorytm powinien zatrzymać szukanie i wykonać ruch. Zauważmy, że to działa nie tylko dla sytuacji jasny-faworyt, ale też dla przypadku ruchów symetrycznych, gdzie ilość przeszukiwania pokaże, ze jeden ruch jest lepszy niż inny.

Ten rodzaj rozumowania o tym, co robią obliczenia, nazywamy metarozumowaniem (rozumowanie o rozumowaniu). Stosuje się go nie tylko do gier, ale do wszelkiego rodzaju rozumowań. Wszystkie obliczenia wykonywane są w służbie spróbowania osiągnięcia lepszej decyzji, wszystkie mają koszty i wszystkie mają jakieś prawodpodobieństwo spowodowania określonych ulepszeń jakości decyzji. Alfa-beta angażuje najprostszy rodzaj metarozumowania, tj.

twierdzenie do efektu, że określone gałęzie drzewa można zignorować bez straty. Można zrobić dużo lepiej.
Na koniec spójrzmy ponownie na naturę przeszukiwania. Algorytmy szukania heurystycznego i grania w gry działają przez generowanie ciągów stanów zaczynając ze stanu początkowego i stosując funkcję oceny. Jest jasne, że ludzie nie grają w ten sposób. W szachach mamy często jeden określony cel w umyśle - np. złapanie w pułapkę hetmana przeciwnika - i używamy go do generowania planów osiągnięcia tego. Ten rodzaj rozumowania (planowania) ukierunkowanego na cel czasami całkowicie eliminuje wyszukiwanie kombinatoryczne. PARADISE Davida Wilkinsa jest jedynym programem, który używa rozumowania goal-directed z powodzeniem w szachach: był w stanie rozwiązać niektóre problemy szachowe wymagające 18-ruchowych kombinacji. Jak dotąd, jednakże, nie rozumiemy dobrze, jak łączyć dwa rodzaje algorytmów w silny i wydajny system. Taki system byłby znaczącym osiągnięciem nie tylko dla badań nad grami, ale też dla badań nad AI w ogólności, ponieważ stałoby się dużo bardziej prawdopodobne zastować go na problemie, z jakim zmaga się ogólny inteligentny agent.

5.8 Podsumowanie

Gry są fascynujące, a pisanie programów grających chyba jeszcze bardziej. Moglibyśmy powiedzieć, że granie w gry ma się do AI jak wyścigi motocyklów Grand Prix do przemysłu samochodowego: wyspecjalizowane zadanie i ekstremalne współzawodnictwo prowadzą do opracowania systemów, które nie wyglądają jak [...]
Najważniejsze są następujące idee:
- Gra moze być zdefiniowana przez stan początkowy (ustawienie planszy), operatory (definiujące legalne ruchy), stan ostateczny (który mówi, czy gra się zakończyła) i funkcja użyteczności / wypłaty (która mówi, kto wygrał i o ile).
- W grach dla dwóch graczy z doskonałą informacją algorytm minimax może wyznaczyć najlepszy ruch dla gracza (zakładając, że przeciwnik gra doskonale) przez wyliczenie całego drzewa gry.
- Przycinanie alfa-beta liczy to samo, co minimax, ale jest wydajniejszy, ponieważ przycina gałęzie drzewa, o których może udowodnić, że są nieważne dla ostatecznego wyniku.
- Zwykle jest niewykonalnym rozważyć całe drzewo gry (nawet z użyciem alfa-beta), więc musimy przycinać szukanie w pewnym momencie i stosować funkcję oceny, która daje oszacowanie użyteczności stanu.
- Z grami z ryzykiem można sobie radzić przez rozszerzenie algorytmu minimax, który oblicza węzły ryzyka przez wzięcie średniej użyteczności wszystkich dzieci, ważonych przez prawdopodobieństwo dziecka.