Paweł Dawidowicz, Mikołaj Dzięciołowski, Stanisław Wilczyński

Showdown AI

Wstęp

W naszym projekcie zajęliśmy się tworzeniem botów do “Pokemon Showdown” - symulatora walk pokemonów, który w 2017 był jednym z konkursów na CIG. W ramach naszego zadania stworzyliśmy kilku różnych agentów i porównaliśmy ich skuteczność w systemie walki każdy z każdym. Użyliśmy frameworka zaproponowanego przez organizatorów konkursu, a nasz kod można znaleźć w repozytorium.

Wykorzystane algorytmy, zaimplementowani agenci

Pierwszym z nich był Random Agent jako baseline do porównywania z innymi agentami. Oczywiście w każdym kroku spośród możliwych ruchów gracza, wybierał on ruch losowy. Warto zauważyć, że daje to duże szanse na zmianę pokemona, gdyż, przy 6-pokemonowym teamie, zmiany to aż 5 z 9 opcji.

Następnym prostym agentem był ε-greedy, który z prawdopodobieństwem $1-\varepsilon$ wybierał ruch w sposób zachłanny: za najlepszy uważał ten atak, który mógł zabrać przeciwnikowi najwięcej HP. Algorytm rozważał również zmiany pokemona w następujący sposób: jeśli nowy pokemon w kolejnej turze mógł zadać przynajmniej dwukrotnie więcej obrażeń, niż obecny, to zmiana była uznawana za lepsze posunięcie niż atak. Oczywiście spośród możliwych zmian wybierano tę, która prowadziła do zadania największych obrażeń.
Z prawdopodobieństwem ε wykonywany był ruch losowy.

Bardziej skomplikowanym algorytmem, który wykorzystaliśmy był minimax. Ze względu na fakt, że w trakcie walki gracze nie wykonują ruchów na przemian tylko o tym kto wykona ruch pierwszy decyduje szybkość pokemonów, minimax był wywoływany rekurencyjnie dopiero po wyborze ruchu przez obu graczy. W przypadku śmierci pokemona, albo ruchu wymuszającego zmianę na innego, przed wywołaniem rekurencyjnym symulowaliśmy tylko akcję gracza, który do zmiany został przymuszony. Ze względu na niezwykle wolną implementację symulatora (więcej o tym w następnej sekcji), która powodowała, że rozegranie pojedynczej partii minimaxem z głębokością 3 trwało bardzo długo (więcej w sekcji wyniki), w wykonanych symulacjach ograniczyliśmy się do głębokości 2. Do oceny stanów na maksymalnej głębokości, algorytm wykorzystywał heurystykę bazującą na sumie HP gracza i przeciwnika. Dokładniej, heurystyka liczy sumę HP pokemonów graczy i liczy ich różnicę z odpowiednimi wagami. Oprócz tego bierze pod uwagę liczbę martwych pokemonów w teamie - pomysł opiera się na założeniu, że różnica między pokemonem mającym 0HP a pokemonem mającym 1HP jest kolosalna, gdyż ten drugi może zadać

jeszcze obrażenia w kolejnej turze, jeśli jest odpowiednio szybki. Rozważaliśmy również włączenie wartości atrybutów pokemona (atak, szybkość, itp), jednak uznaliśmy, że nie poprawi to znacząco funkcji, gdyż ich istotność dla oceny stanu gry zależy od zbyt wielu czynników, aby próbować go wyrazić w tak prosty sposób: przykładowo, na niewiele zda się pokemon mający dużo ataku, jeśli ma mało hp i mało szybkości.
Problem z niepełną informacją o przeciwniku rozwiązujemy za pomocą narzędzi zastanych we frameworku. Dla pokemona przeciwnika jego możliwe ruchu sczytujemy ze schematów na temat tego konkretnego pokemona. Powoduje to, że czasem nasz algorytm opiera się na informacjach, które nie są prawdziwe. Natomiast w samym minimaxie (oraz MCTS również) nie bierzemy w ogóle pod uwagę pokemonów przeciwnika, których do tej pory nie widzieliśmy (nie rozważamy możliwych zmian na nieznanego pokemona ze względu na zbyt wiele opcji). Oczywiście samą liczbę żywych pokemonów przeciwnika i ich HP uwzględniamy w heurystyce do oceny stanów walki.

Kolejnym był algorytm MCTS (Monte-Carlo Tree Search), który polega na wykonaniu ustalonej liczbie tur, w których każda składa się z czterech kroków:

1. Wybór - Zaczynając od korzenia drzewa R, wybieramy kolejne węzły potomne aż dojdziemy do liścia drzewa L.
2. Rozrost - o ile L nie kończy gry, tworzymy w nim węzły potomne (dla wszystkich możliwości - liczbie możliwych ruchów danego gracza w danym stanie gry) i wybieramy z nich (losowo) jeden węzeł P.
3. Symulacja - rozgrywamy losową symulację z węzła P.
4. Propagacja wstecz - na podstawie wyniku rozegranej symulacji uaktualniamy informację w węzłach na ścieżce prowadzącej od P do R.

Oczywiście pojawia się pytanie jak dokonuje się wyboru kolejnych węzłów potomnych - najpopularniejszym wzorem do tego jest wybranie takiego węzła potomnego, dla którego wartość C jest największa, gdzie:

C = K + $\sqrt{2\frac{ln(n_L)}{n_P}}$, zaś K = $\frac{w_P}{n_P}$, jeżeli wybieramy węzeł dla naszego ruchu, lub K = 1-$\frac{w_P}{n_P}$ jeżeli wybieramy węzeł dla przeciwnika (on maksymalizuje swoje zwycięstwa, my swoje), gdzie w_p, w_L - odpowiednio liczba wygranych węzła P i L (L jest rodzicem P), zaś n_L, n_P - odpowiednio liczba przeprowadzonych symulacji dla węzłów L i P. Taki sposób wyboru węzłów polega na wybraniu najlepszej opcji dla danego gracza z uwzględnieniem możliwości odkrywania nowych rozwiązań. Dodatkowo do liczby przeprowadzonych symulacji n_p dodajemy pewien epsilon, żeby nie dzielić przez zero.

Liczba iteracji w tym algorytmie jest uzależniona od czasu na ruch - niestety implementacja symulatora walk w wykorzystywanym przez nas frameworku, jest okropnie wolna - walka w pokemonach online trwa około 5 minut. Zaś dla MCTS przy ilości iteracji równej 100, symulacja walki trwa około godziny, więc kompletnie nie nadaje się do walki online. Sam branching factor jest równy około 9 (ale w rzadkim najgorszym przypadku może dojść do 18), co sprawia, że MCTS praktycznie nie różni się od algorytmu losowego przy tak małej liczbie symulacji. MCTS polega między innymi na szybkiej symulacji - w końcu wykonujemy losowe ruchy, jednak nawet tak prosta rzecz w naszym symulatorze jest wolna. Przez co ograniczyliśmy długość symulacji (zazwyczaj do około 50) i symulację uznawaliśmy za wygraną, jeżeli przeciwnik stracił więcej punktów procentowych swoich HP niż my. Ze

względu na brak pełnej informacji o przeciwniku, przestawaliśmy symulować, jeśli umarł mu ostatni znany nam pokemon. Konsekwencją niepełnej informacji było też to, że podczas symulacji mogliśmy często pójść w gałąź nie odpowiadającą rzeczywistemu stanowi gry (wykonujemy pokemonem przeciwnika ruch, którego on nie ma). Niestety nie dało się przetestować MCTS na większej liczbie iteracji - przy 1000 trwało to 11 godzin i zajmowało około 3gb ramu. Jest to dość zaskakujące biorąc pod uwagę nieduże skomplikowanie gry. Jednak za każdym razem, gdy tworzymy nowy węzeł musimy kopiować stan gry, a ten ma mnóstwo pól, które często są duplikowane w innych strukturach (np. BattleSide).

Wyniki i przykładowe czasy działania

Tabela poniżej przedstawia różnego rodzaju wyniki, dla liczby x symulacji, gdzie każda pojedyncza symulacja to 6 walk (3 różnych). Metody zaczynające się od “our_” są naszymi implementacjami. Pierwszy argument dla “our_MinMax” to głębokość MinMax’a, zaś pierwszy i drugi argument dla “our_MCTS” to odpowiednio liczba iteracji i głębokość symulacji dla algorytmu MCTS.

Wygrany	Przegrany	Wygrane (procenty)	Liczba symulacji	średni czas jednej symulacji (minuty)
our_MinMax(2)	our_MinMax(3)	100	1	587
our_MCTS(1000,100)	RandomAgent	50	1	660
our_EpsGreedy	our_MCTS(1000,100)	100	1	350
our_EpsGreedy	RandomAgent	96,7	5	0.05
our_MiniMax(2)	RandomAgent	93,3	5	19
our_MCTS(300,50)	RandomAgent	53,3	5	156
our_MiniMax(2)	our_EpsGreedy	56,7	5	15
our_EpsGreedy	our_MCTS(300,50)	91,7	2	52
our_MiniMax(2)	our_MCTS(300,50)	91,7	2	160
our_EpsGreedy	minimax	75	2	30
our_EpsGreedy	Random	100	2	0.071
our_EpsGreedy	BFS	100	2	3

our_EpsGreedy	MLQLearner	100	2	0.08
our_EpsGreedy	QLearner	100	2	0.1
our_EpsGreedy	Pessimist	83	2	0.1
our_EpsGreedy	OTL	50	2	0.08

Podsumowanie

Minimax Algorytm Minimax wydawał się dobrym wyborem do rozważanej gry. Turowy charakter gry i prostota heurystyki wydającej się całkiem rozsądnym rozwiązaniem, wydawały się być obiecującą prognozą. Istotnym problemem okazał się "nie do końca turowy" charakter gry (kolejność na podstawie szybkości pokemonów) oraz niepełne informacje o stanie gry. Algorytm pokonywał RandomAgenta z niemal 100% skutecznością, co świadczy o tym, że wykonywał ruchy generalnie prowadzące w stronę zwycięstwa. W porównaniu z algorytmem Eps-Greedy wykazał się podobną skutecznością, wygrywając w około 50% rozgrywek z nim. Z powodu zejść rekurencyjnych przy rosnącym branching factor (ze względu na schemat ruchów zawierający więcej niż 4) jego decyzje trwały jednak znacznie dłużej (średni czas rozgrywki ok. 20 min vs 0.05 min). Ciekawą obserwacją jest fakt, że Minimax o głębokości 2 okazał się działać dużo lepiej niż o głębokości 3.
Epsilon-Greedy W praktyce podejście zachłanne okazało się dość skuteczne. Algorytm osiągnął wyniki porównywalne z Minimaxem przy znacznie szybszym czasie działania i oszczędności pamięci. Pozwoliło to wykorzystać go w większej liczbie symulacji i porównać z algorytmami zaproponowanymi przez twórców symulatora; Z większością z nich radzi sobie dobrze osiągając skuteczność niewiele niższą od 100%. Wyjątkiem jest algorytm OTL (jednak jest to algorytm bardzo podobny do algorytmu Epsilon-Greedy, więc wygrane rzędu 50% wydają się naturalnym wynikiem) oraz algorytm minimax, o którego skuteczności napisaliśmy już wcześniej. Warto zauważyć, że nasza implementacja algorytmu minimax osiągnęła lepsze wyniki w starciu z Epsilon-Greedy od zaproponowanej przez twórców symulatora.
MCTS Algorytm oparty na losowych symulacjach okazał się najsłabszym spośród testowanych przez nas rozwiązań. Nawet z losowym agentem osiągał on wyniki na poziomie 50% nawet przy zwiększonej liczbie iteracji. Na pewno wpływ miała (podobnie jak w przypadku minimaxa) niepełna informacja o pokemonach przeciwnika i ich atakach. Głównym czynnikiem na pewno była jednak mała liczba iteracji spowodowana ograniczeń czasowym związanym z wydolnością symulatora, przez co aproksymacja wartości wierzchołków naszego drzewa gry była zbyt niedokładna. Niestety implementacja naszego symulatora jest na tyle skomplikowana, że próba jej optymalizacji to zadanie na miesiące pracy. Jesteśmy jednak pewni, że dałoby się to poprawić, że nawet gra przy 10000 iteracjach trwałaby 5 minut. Być może właśnie to jest powodem, dla którego powstaje nowa wersja, tym razem pisania w Javie. Wspomniana szybkość już na samym początku pracy pozbawiła nas jakichkolwiek złudzeń, co do możliwości zastosowania jakichkolwiek algorytmów polegających na uczeniu.