Silnik do gry w Santorini

Informacje o grze

- Strona Santorini na BGG
- Santorini do gra planszowa bez losowości, w której gracze mają pełną informację o planszy
- Gra polega na ruszaniu się i budowaniu pionkami według określonych zasad, zwycięzcą zostaje gracz, który wszedł na 3 poziom

Informacje o grze

- Strona Santorini na BGG
- Santorini do gra planszowa bez losowości, w której gracze mają pełną informację o planszy
- Gra polega na ruszaniu się i budowaniu pionkami według określonych zasad, zwycięzcą zostaje gracz, który wszedł na 3 poziom
- Branching factor tej gry to średnio ok. 70, a liczba ruchów w jednej partii to 30-40

Użyte narzędzia

- Silnik został zaimplementowany w oparciu o dwa algorytmy: Monte Carlo Tree Search oraz algorytm minimax.

Użyte narzędzia

- Silnik został zaimplementowany w oparciu o dwa algorytmy: Monte Carlo Tree Search oraz algorytm minimax.
- MCTS tworzy drzewo możliwych pozycji, w wierzchołkach którego zapisuje liczbę symulacji i liczbę wygranych z danej pozycji. Po dojściu do liścia, przebiegi gry są generowane losowo.
- Wierzchołki były wybierane za pomocą wzoru UCT, który daje balans między eksploracją i eksploatacją - wybierane były wierzchołki z największą wartością wyrażenia $\frac{w_i}{n_i} + \sqrt{2\frac{\ln N_i}{n_i}}$.
- metoda MCTS dała raczej słabe rezultaty

Minimax

- Algorytm minimax polega na wybraniu spośród możliwych ruchów takiego, który minimalizuje maksymalną stratę - inaczej mówiąc, wybiera najlepszy ruch, biorąc pod uwagę, że przeciwnik również wybierze najlepszy ruch.
- Przeszukanie wszystkich możliwych ruchów jest niemożliwe obliczeniowo, więc w podstawowej wersji minimax ogranicza się do przeszukiwania drzewa możliwych pozycji na daną głębokość.

Funkcja ewaluacji

- żeby ocenić pozycje, w których gra się dalej toczy, wprowadza się funkcję ewaluacji pozycji na planszy - najprostsza po prostu zwraca 1 w przypadku wygranej pozycji, 0 w przypadku nierozstrzygniętej i -1 w przypadku porażki. Taki algorytm dobrze działa w pozycjach bliskich końowi rozgrywki, ale na początku jego ruchy są losowe (Jest jednak w stanie pokonać algorytm Monte Carlo)

Funkcja ewaluacji

- żeby ocenić pozycje, w których gra się dalej toczy, wprowadza się funkcję ewaluacji pozycji na planszy - najprostsza po prostu zwraca 1 w przypadku wygranej pozycji, 0 w przypadku nierozstrzygniętej i -1 w przypadku porażki. Taki algorytm dobrze działa w pozycjach bliskich końowi rozgrywki, ale na początku jego ruchy są losowe (Jest jednak w stanie pokonać algorytm Monte Carlo)
- Po napisaniu ręcznie własnej funkcji ewaluacji, silnik korzystający z minimaxa osiągał już całkiem niezłe rezultaty - grał mniej więcej na poziomie doświadczonego człowieka (czyli mnie)

qsearch

- Quiet search to wariacja algorytmu minimax, radząca sobie z problemem horyzontu - ucinamy zawsze przeszukiwanie na danej głębokości, mimo że mogą być ruchy, które znacząco zmieniają pozycję. W przypadku qsearcha, po zejściu na maksymalną głębokość, sprawdzamy jeszcze najbardziej obiecujące ruchy.
- zaimplementowanie qsearcha poprawiło działanie silnika - wygrywał on z silnikiem wykorzystującym zwykłego minimaxa z taką samą funkcją ewaluacji.

qsearch

- Quiet search to wariacja algorytmu minimax, radząca sobie z problemem horyzontu - ucinamy zawsze przeszukiwanie na danej głębokości, mimo że mogą być ruchy, które znacząco zmieniają pozycję. W przypadku qsearcha, po zejściu na maksymalną głębokość, sprawdzamy jeszcze najbardziej obiecujące ruchy.
- zaimplementowanie qsearcha poprawiło działanie silnika - wygrywał on z silnikiem wykorzystującym zwykłego minimaxa z taką samą funkcją ewaluacji.

Ostatecznie, RandomEngine $<$ MonteCarlo $<$ Minimax(SimpleEval) $<$ Qsearch(SimpleEval) $<$ Minimax(MyEval) $<$ Qsearch(MyEval).

Dalsza ścieżka rozwoju

- Do ewaluacji pozycji na planszy można użyć sieci neuronowych. Trenowanie sieci poprzez rozgrywanie z danej pozycji całej gry i funkcja straty na podstawie jej rezultatu byłyby bardzo nieefektywne.
- Zamiast tego możemy użyć techniki Temporal Difference Learning - jeżeli dla pozycji p wykonamy optymalny ruch i otrzymamy pozycję p', to $val(p) \approx val(p')$.
- Możemy więc uczyć sieć w następujący sposób: losowo generujemy pozycję, używamy aktualnego silnika do zagrania np. 10 ruchów i błąd liczymy jako suma $\Sigma_{i=1}^{10} \lambda^i |val(p_0) - val(p_i)|$.

Dalsza ścieżka rozwoju

- Do ewaluacji pozycji na planszy można użyć sieci neuronowych. Trenowanie sieci poprzez rozgrywanie z danej pozycji całej gry i funkcja straty na podstawie jej rezultatu byłyby bardzo nieefektywne.
- Zamiast tego możemy użyć techniki Temporal Difference Learning - jeżeli dla pozycji p wykonamy optymalny ruch i otrzymamy pozycję p', to $val(p) \approx val(p')$.
- Możemy więc uczyć sieć w następujący sposób: losowo generujemy pozycję, używamy aktualnego silnika do zagrania np. 10 ruchów i błąd liczymy jako suma $\Sigma_{i=1}^{10} \lambda^i |val(p_0) - val(p_i)|$.
- Niestety, nie zdążyłem zaimplementować sieci neuronowej pozwalającej na działanie silnika.