Zaawansowane Metody Inteligencji Obliczeniowej

Lab 12: Monte-Carlo Tree Search

Michał Kempka Marek Wydmuch

27 maja 2021

Fundusze Europejskie Polska Cyfrowa
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Rozwoju Regionalnego

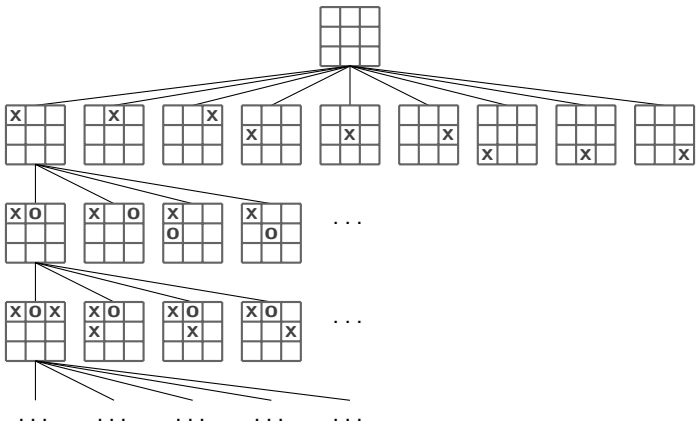
"Akademia Innowacyjnych Zastosowań Technologii Cyfrowych (AI Tech)",
projekt finansowany ze środków Programu Operacyjnego Polska Cyfrowa POPC.03.02.00-00-0001/20

1 Wprowadzenie

Do tej pory rozważaliśmy dwa rodzaje metod: **model-based** i **model-free**. Przy pierwszych zakładaliśmy, że mamy dostępny model przejść (ang. transition model), czyli łączny rozkład prawdopodobieństwa otrzymywanych nagród i wystąpienia następnych stanów ($\mathcal{P}(s', r|s, a) = Pr\{s_{t+1} = s', r_t = r|s_t = s, a_t = a\}$), stąd nazywany czasem również **modelami z rozkładem (ang. distribution model)**. Dzisiaj skupimy się na problemach w których posiadamy model środowiska, który nie zawiera dokładnego modelu przejścia, ale jest wystarczający by generować następujący stan i nagrodę zgodnie z prawdziwym rozkładem w efektywny sposób. Modele takie nazywane są czasem **modelami z próbkowaniem (ang. sample models)**. Takie modele w wielu problemach są dużo łatwiejsze do otrzymania niż model z rozkładem. Przykładem tutaj są gry planszowe, w których bazując na zasadach, możemy dokładnie symulować przebieg rozgrywki, ale niektóre rozkłady zdarzeń losowych mogą być trudne do wyliczenia. Sample model może być z powodzeniem użyty do planowania, które nie ma jednej ścisłej definicji, jednak można je rozumieć jako poszukiwanie optymalnej sekwencji akcji w przestrzeni stanów. Na grach się właśnie przede wszystkich dzisiaj skupimy.

Istotne jest dla nas wprowadzenie koncepcji drzewa stanów, czyli reprezentacji gdzie stan odpowiada wierzchołkowi w drzewie a stany po nim następujące są jego dziećmi. Liście w drzewie to stany terminalne. Schemat drzewa stanów dla gry w kółko i krzyżyk jest przedstawiony na Rysunku 1.

Pytanie: Ile wierzchołków ma drzewo stanów dla gry kółko i krzyżyk (zakładając, że gramy do zapełnienia planszy)?



Rysunek 1: Schemat drzewa stanów dla gry w kółko i krzyżyk.

Jeśli problem (gra, np. Sokoban[1]) jest środowiskiem jedno-agentowym, można go rozwiązać przeszukując jego drzewo stanów za pomocą jednego z powszechnych algorytmów przeszukiwania, np. BFS (ang. Breadth-first Search), DFS (ang. Depth-First Search) albo w wypadku gdy drzewo zawiera dużo wierzchołków można zastosować algorytm A* z heurystyką typującą najlepsze wierzchołki do odwiedzenia w pierwszej kolejności (jeśli heurystyka ta spełnia pewne założenia, algorytm A* gwarantuje znalezienie optymalnego rozwiązania w minimalnej liczbie odwiedzonych wierzchołków). Więcej o takich problemach przeszukiwania możesz przeczytać w Rozdziale 3 w Russell and Norvig [1].

W większość gier gramy jednak przeciwko innym graczom, gdzie takie przeszukiwanie przestaje działać. Zacznijmy od konkretnego rodzaju gier: dla dwóch graczy, deterministycznych, z perfekcyjną informacją i o **sumie zerowej (ang. zero-sum)**. Gra o sumie zerowej to taka gra, gdzie suma całkowitych nagród wszystkich graczy na koniec jest zawsze taka sama (nie musi być to 0). Przykładem takich gier mogą być np. szachy, gdzie nagrodę $+1$ przyznajmy za wygraną, -1 za przegraną i 0 za remis.

2 Minimax

Aby rozwiązać problem gry z dwoma graczami musimy zmodyfikować nasz algorytm przeszukiwania i uwzględnić naszego przeciwnika, którego celem jest maksymalizacja swojej własnej nagrody, a ponieważ gramy w grę o sumie zerowej jest to tożsame z minimalizacją naszej nagrody. Stąd nazwa algorytmu przeszukiwania Minimax. Algorytm ten wykonuje przeszukiwanie, które wybiera akcje maksymalizujące naszą nagrodę, zakładając że gramy przeciwko nieomylnemu przeciwnikowi, który będzie wybierać akcje minimalizujące naszą nagrodę. Mając dane drzewo stanów, wartość każdego stanu zgodnie z algorytmem Minimax wyliczamy w następujący sposób:

$$Minimax(s) = \begin{cases} CakowitaNagroda(s) & \text{if } StanKocowy(s) \\ \max_{a \in \mathcal{A}_s} Minimax(NastpnyStan(s,a)) & \text{if } Gracz(s) = max \\ \min_{a \in \mathcal{A}_s} Minimax(NastpnyStan(s,a)) & \text{if } Gracz(s) = min \end{cases} \quad (1)$$

Schemat algorytmu Minimax dla gry w kółko i krzyżyk jest widoczny na Rysunku 2 oraz ogólny na Rysunku 3. Minimax gwarantuje znalezienie optymalnej strategii przeciwko optymalnemu przeciwnikowi. Co jednak jeśli przeciwnik nie jest optymalny? Minimax uzyska lepszy wynik, może być możliwe, że inna strategia uzyska lepszy wynik przy grze z nieoptymalnym przeciwnikiem, jednak nie uzyska ona lepszego wyniku przy grze z optymalnym przeciwnikiem. Algorytm minimax można łatwo rozszerzyć do gier z większą ilością graczy i z niedeterministycznymi akcjami. O rozszerzeniach algorytmu Minimax możesz przeczytać w Rozdziale 5 w Russell and Norvig [1].

Pytanie: Pokaż, że algorytm Minimax, uzyska co najmniej tak dobry wynik przy grze przeciwko suboptymalnemu przeciwnikowi jak przy grze z optymalnym przeciwnikiem.

Pytanie: Pokaż algorytm, który może uzyskać lepszy wynik niż Minimax dla pewnego nieoptymalnego przeciwnika (wskaż przykładowego przeciwnika).

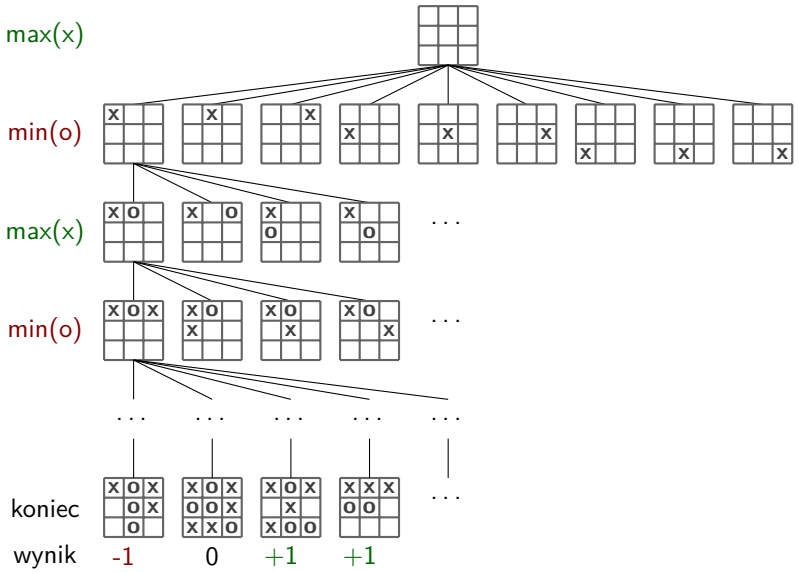
Problem z algorytmem Minimax jest taki, że wymaga przeszukania całej przestrzeni stanów, która zazwyczaj rośnie wykładniczo z głębokością drzewa. Jedną z metod jest eliminowanie całych poddrzew w drzewie, jeśli wartość $Minmax(s)$ od nich nie zależy, co prowadzi do algorytmu **Alpha-beta pruning**. Niestety nie eliminuje ona wykładniczej złożoności, ale i tak znacząco przyśpiesza działanie. Algorytm alpha-beta pruning (z licznymi rozszerzeniami, w tym heurystykami oceniającymi stany) był podstawą programu IBM Deep Blue, który wygrał z mistrzem świata Garrym Kasparovem mecz w szachy w 1997 roku. O Alpha-beta pruning i dalszych rozszerzeniach przyspieszających algorytm Minimax możesz przeczytać w Rozdziale 5 w Russell and Norvig [1].

3 Monte Carlo Tree Search

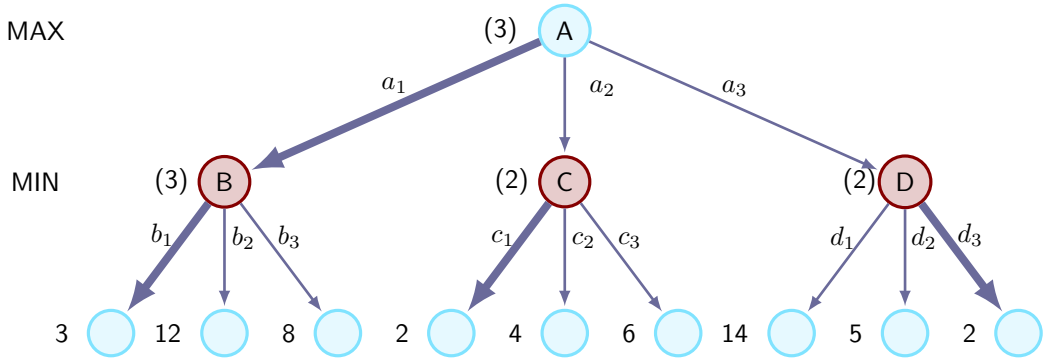
Wymyślony w 2007 roku Monte Carlo Tree Search (**MCTS**) okazał się ogromnym sukcesem stając się bazą dla nowych, lepszych i bardziej wydajnych programów do gry w szachy, go, pokera i wielu innych. Monte Carlo Tree Search stanowi również bazę algorytmu **AlphaGo**[2], który w 2016 wygrał mecz w Go z mistrzem świata Lee Sedolem.

W przeciwieństwie do podstawowego algorytmu Minimax, MCTS nie wylicza wartości akcji dla każdego z możliwych stanów, a dokonuje tzw. planowania w momencie decyzji, czyli w momencie otrzymania nowego stanu s. Algorytm MCTS używa, pod spodem, innego algorytmu, tzw. **rollout**, który estymuje wartość akcji w stanie s poprzez uśrednianie wyników symulacji epizodów, które zaczynają się w stanie s oraz wybraniem

[1] https://pl.wikipedia.org/wiki/Sokoban



Rysunek 2: Schemat algorytmu Minimax dla gry w kółko i krzyżyk.



Rysunek 3: Przykład algorytmu Minimax.

każdej z możliwych akcji $a \in \mathcal{A}_s$ aż do stanu terminalnego zgodnie z daną polityką rolloutu π (ang. rollout policy, albo ang. sample policy). Celem rolloutu nie jest estymacja optymalnej funkcji wartości albo estymacja funkcji wartości dla polityki π. Celem jest jedynie wyestymowanie wartości dla par $(s, a), a \in \mathcal{A}_s$ dla polityki π.

Dlaczego jest to istotne? Jeśli mamy dwie polityki π oraz π', które są różnią się tylko tym, że $\pi'(s) = a \neq \pi(s)$ dla jakiegoś stanu s, a $q_\pi(s, a) > v(s)$, wtedy polityka π' jest lepsza od π. Jeśli π to polityka rolloutu, to dochodzimy do tego, że celem rolloutu jest znalezienie lepszej polityki niż π, a nie znalezienie optymalnej polityki. W praktyce, nawet jeśli π jest losową polityką, rollout okazuje się wyjątkowo skuteczny dla wielu problemów.

Monte Carlo Tree Search łączy ideę rollout π_r z polityką drzewa π_t, która bazując na wartościach otrzymanych przez symulacje epizodów wybiera akcje i stany, dla których warto wykonać dalsze symulacje. Ostatecznie otrzymujemy algorytm który składa się z 4 operacji następujących po sobie daną liczbę razy dla aktualnego stanu s:

1. **Selekcja** – zaczynając od korzenia drzewa (wierzchołka stanu s), używając akcji wyznaczonych przez politykę drzewa π_t, przejdź przez drzewo T (początkowo w drzewie jest tylko s) aż nie napotkasz liścia s' (stanu, który ma nieodwiedzone dziecko). Pamiętaj, że w wypadku gry z przeciwnikiem uwzględnić to kiedy jest jego kolej wybierania akcji. Np. dla stanów odpowiadających jego ruchowi wybierać stany, które minimalizują Twoją nagrodę (zakładamy, że przeciwnik również dąży do grania w optymalny sposób), można to osiągnąć również przez odwrócenie wygranej/przegranej w fazie powrotu.
2. **Ekspansja** – dodaj do drzewa T wszystkie lub część (np. tylko najlepsze zgodnie z polityką π_t, ale jeśli nie jest ona bardzo wyszukana, a przestrzeń nie jest bardzo duża dodaj najlepiej wszystkie) dzieci stanu s'.

3. **Symulacja** – Z wierzchołka s' lub jednego z jego dzieci dokonaj symulacji epizodu zgodnie z polityką π_r.
4. **Powrót** – Nagrodę otrzymaną w symulacji uwzględnij we wszystkich wierzchołkach, które aktualnie znajdują się w drzewie T.

Tak jak powiedzieliśmy, że losowa polityka rolloutu jest efektywna dla wielu problemów, tak samo polityka drzewa może być prostą polityką, musi ona jednak dbać o odpowiednią eksplorację drzewa, dlatego może być to znana nam polityka ϵ-greedy. Dla gier dla dwóch graczy, gdzie nagroda jest przyznawana wyłącznie za wygraną grę popularnym wyborem jest też polityka **UCB**, gdzie wybierana jest akcja maksymalizująca poniższe wyrażenie:

$$\frac{w}{n} + c\sqrt{\frac{\ln t}{n}} \tag{2}$$

gdzie w to liczba wygranych symulacji przechodzących przez dany wierzchołek, n to liczba symulacji, które przeszły przez dany wierzchołek/stan, c to stała kontrolująca eksplorację, t liczba symulacji, które przeszły przez wierzchołek rodzica. Pseudokod algorytmu dla MCTS dla dwóch graczy znajduje się w Algorytmie 1.

Algorytm 1: Pseudokod dla algorytmu Monte Carlo Tree Search

Inicjalizacja:
s stan dla którego mamy wybrać akcję
π_r dana polityka rolloutu
π_t dana polityka drzewa
Liczba symulacji do wykonania I
Drzewo $T = \{s\}$
Tablica wartości V, początkowo, dla każdego stanu $V(S) = 0$
for $i \leftarrow 1, \ldots, I$ **do**
    1. Selekcja
    $S \leftarrow s$
    while $Dzieci(S) \notin T$ **do**
        Przejdź do następnego stanu S' zgodnie z $\pi_t(S)$ (uwzględniając V)
        $S \leftarrow S'$
    2. Ekspansja
    $T \leftarrow T \cup Dzieci(S)$
    3. Symulacja
    $S' \leftarrow S$
    while $\neg Terminalny(S')$ **do**
        Przejdź do następnego stanu S'' zgodnie z $\pi_r(S')$
        $S' \leftarrow S''$
    Wylicz sumaryczną nagrodę R zdobytą od S do S'
    4. Powrót
    Uaktualnij estymację $V(S)$ zgodnie z R
    while $S \neq s$ **do**
        $S' \leftarrow Rodzic(S)$
        Uaktualnij estymację $V(S')$ zgodnie z R, oraz nagrodą r z przejścia S' do S.
        $S \leftarrow S'$

Literatura

[1] Russell, S. and Norvig, P. (2010). *Artificial Intelligence: A Modern Approach*. Prentice Hall, third edition.

[2] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503.

[3] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. The MIT Press, second edition.

Fundusze Europejskie Polska Cyfrowa
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Rozwoju Regionalnego

"Akademia Innowacyjnych Zastosowań Technologii Cyfrowych (AI Tech)",
projekt finansowany ze środków Programu Operacyjnego Polska Cyfrowa POPC.03.02.00-00-0001/20