WSI zad3

Michał Mokrzycki 324874

November 2023

1 Opis zadania

Celem ćwiczenia jest implementacja algorytmu minimax z obcinaniem $\alpha - \beta$. Dla różnych ruchów o tej samej jakości, algorytm powinien zwracać losowy z nich. Następnie należy wykorzystać implementację do porównania skuteczności algorytmu dla różnych głębokości przeszukiwania dla prostej gry o następujących zasadach:

Na początku gry dany jest zbiór N identycznych żetonów, z którego gracze na zmianę zabierają żetony.

- Dwóch graczy wykonuje ruchy na zmianę.
- Podczas swojej tury gracz usuwa od 1 do K żetonów.
- Przegrywa gracz, który zabierze ostatni żeton.
- Eksperymenty powinny uwzględniać sytuację w której $K = 3$, a N to liczba losowana z zakresu $8 - 20$ (z rozkładem jednostajnym)

2 Algorytmy

W celu poprawnego wykonania zadania zaimplementowana kilka funkcji używanych potem przy symulacji rozgrywki komputer-komputer. Graczem maksymalizującym będziemy nazywać gracza wykonującego 1 ruch Są to:

- funkcja oceny - zwracająca w naszym wypadku 1 gdy pozycja jest **wygrywająca** dla gracza maksymalizującego i -1 gdy jest ona dla niego **przegrywająca**. Gdy stan dla którego będziemy dokonywać oceny będzie znajdował się na maksymalnej głębokości oraz nie będzie on stanem terminalnym. Algorytm zwróci wtedy 0, ponieważ nie da się heurystycznie ocenić która pozycja jest lepsza od innej. Jest to gra w której nie ma remisów i zakładając bezbłędną grę, każda pozycja początkowa ma jasno określonego zwycięzce zależnego od tego kto pierwszy wykona ruch.
- Algorytm **mini-max z przycinaniem alfa-beta**
- funkcja **best move** znajdująca najlepszy ruch dla danej pozycji/stanu. Wykorzystuje ona funkcje **mini max a b** i zwraca ruch o najwyższej ocenie dla gracza maksymalizującego i najniższa dla minimalizującego. W przypadku gdy oceny ruchów są takie same(a w naszym przypadku dzieje się to bardzo często bo mamy tylko 3 różne warianty oceny - 1,-1 oraz 0) funkcja zwraca losowy z nich przy pomocy biblioteki random
- funkcja **game** zwracająca 1 gdy wygra gracz maks lub 0 dla gracza min oraz czas rozgrywki
- funkcja **percentage of wins**, korzystająca z funkcji game i pozwalająca na zwrócenie procentu zwycięstw gracza rozpoczynającego oraz średni czas dla danej liczby iteracji
- pozostałe funkcje odpowiadają za generowanie tabel lub danych wykorzystywanych w sprawozdaniu

3 Deterministyczna strategia zwycięstwa

Przedstawiona gra jest stosunkowo mało złożona i dla niewielkich wartości żetonów początkowych z łatwością można by przeszukać nawet całe drzewo gry. Jako że jest to gra deterministyczna o pełnej informacji, przy odpowiedniej głębokości przeszukiwania możemy znaleźć idealną strategię dla graczy, prowadzącą w każdym przypadku do wygranego jednego z nich dla danej początkowej liczby żetonów. I tak dla przykładu jeśli zakładamy , że jednorazowo można zabrać maksymalnie 3 żetony, na pierwszy rzut oka jest zauważalne, że dla początkowej liczby żetonów 2, 3 lub 4, gracz który pierwszy wykona ruch z łatwością może doprowadzić do

sytuacji gdy na planszy zostanie jeden żeton a to oznacza przegraną kolejnego gracza. Można nazwać te pozycje początkowe *wygrywającymi*. Jednak gdy rozpoczynamy grę od przykładowo 5 żetonów to niezależnie czy pierwszy gracz zdejmie z planszy jeden, dwa czy trzy żetony, zostawi kolejnego gracza na pozycji *wygrywającej*. Analogicznie jeśli pierwszy gracz rozpocznie grę z 6,7,8 żetonami to wykonując dobry ruch zostawi gracza drugiego na 5 żetonach, czyli pozycji przegrywającej itd. Przesymulujmy wyniki gier przy bardzo dużej głębokości (takiej aby na pewno drzewo gry było przeszukane do podstawy czyli np. dla 10 żetonów maksymalna liczba ruchów może wynosić 10 itd.) dla początkowej liczby żetonów od 8 do 20.

N	% wygranych gracza rozpoczynającego	Średni czas gry (s)	Liczba gier
8	100	0.0003	50
9	0	0.0005	50
10	100	0.0009	50
11	100	0.0011	50
12	100	0.0017	50
13	0	0.0031	50
14	100	0.0064	50
15	100	0.0077	50
16	100	0.0102	50
17	0	0.0207	50
18	100	0.0499	50
19	100	0.0509	50
20	100	0.093	50

Widzimy, że przy głębokości przewyższającej maksymalną wysokość drzewa, zwycięzca jest z góry określony.

3.1 Wyniki dla mniejszych głębokości

Przetestujmy na początku przypadki gdy rozpoczynający gracz jest na pozycji przegrywającej czyli 9,13 oraz 17.

Initial tokens	K	depth	Max player win %	average_time(sec)
9	3	1	54.0%	0.0001
9	3	2	36.0%	0.0001
9	3	3	0.0%	0.0001
9	3	4	0.0%	0.0002
9	3	5	0.0%	0.0002
9	3	6	0.0%	0.0003
9	3	7	0.0%	0.0003
9	3	8	0.0%	0.0003
13	3	1	60.0%	0.0
13	3	2	48.0%	0.0001
13	3	3	62.0%	0.0002
13	3	4	32.0%	0.0004
13	3	5	0.0%	0.0005
13	3	6	0.0%	0.0009
13	3	7	0.0%	0.0014
13	3	8	0.0%	0.0017
17	3	1	52.0%	0.0001
17	3	2	52.0%	0.0001
17	3	3	54.0%	0.0002
17	3	4	52.0%	0.0005
17	3	5	32.0%	0.0008
17	3	6	26.0%	0.0016
17	3	7	0.0%	0.0026
17	3	8	0.0%	0.0066

Dla każdej głębokości wykonywane było po 50 eksperymentów. Nie trudno zauważyć, ze dla niewielkich głębokości przeszukiwania algorytm nie jest w stanie przypisać oceny danej pozycji, dlatego przypisuje jej 0, a ruch wybierany jest losowo. Przykładowa symulacja gry dla głębokości 1.

```
Game simulation AI–AI N.o 1
Number of initial tokens: 17
Depth: 1
Best moves: [1, 2, 3]
Maximizing player takes: 1 tokens, remaining tokens: 16
Best moves: [1, 2, 3]
Minimizing player takes: 2 tokens, remaining tokens: 14
Best moves: [1, 2, 3]
Maximizing player takes: 3 tokens, remaining tokens: 11
Best moves: [1, 2, 3]
Minimizing player takes: 3 tokens, remaining tokens: 8
Best moves: [1, 2, 3]
Maximizing player takes: 3 tokens, remaining tokens: 5
Best moves: [1, 2, 3]
Minimizing player takes: 2 tokens, remaining tokens: 3
Best moves: [2]
Maximizing player takes: 2 tokens, remaining tokens: 1
Best moves: [1]
Minimizing player takes: 1 tokens, remaining tokens: 0
Minimizng player loses
```

Oraz dla mniejszej ilości tokenów początkowych ale większej głębokości.

```
Game simulation AI–AI N.o 2
Number of initial tokens: 13
Depth: 3
Best moves: [1, 2, 3]
Maximizing player takes: 1 tokens, remaining tokens: 12
Best moves: [1, 2, 3]
Minimizing player takes: 1 tokens, remaining tokens: 11
Best moves: [1, 2, 3]
Maximizing player takes: 1 tokens, remaining tokens: 10
Best moves: [1, 2, 3]
Minimizing player takes: 2 tokens, remaining tokens: 8
Best moves: [3]
Maximizing player takes: 3 tokens, remaining tokens: 5
Best moves: [1, 2, 3]
Minimizing player takes: 1 tokens, remaining tokens: 4
Best moves: [3]
Maximizing player takes: 3 tokens, remaining tokens: 1
Best moves: [1]
Minimizing player takes: 1 tokens, remaining tokens: 0
Minimizng player loses
```

Widzimy jakie błędy popełnia algorytm. Przykładowo gdy w drugiej symulacji na planszy zostaje 10 żetonów, gracz minimalizujący powinien zabrać z planszy dokładnie jeden żeton, pozostawiając wtedy gracza maksymalizującego z 9 żetonami, czyli pozycji dla niego przegrywającej. Algorytm nie "widzi"jednak zakończenia tej ścieżki ruchu z powodu zbyt niskiej głębokości. Z drugiej strony dla odpowiednio dużej głębokości algorytm może oczywiście przeszukać całe drzewo i może określić wszystkie ścieżki prowadzące do zwycięstwa, tak jak w symulacji poniżej. Gracz maksymalizujący ma zawsze ocenione najwyżej wszystkie ruchy i wybiera po prostu losowy, ponieważ algorytm wie, że każdy wybór i tak doprowadzi do wygranej minimalizującego.

```
Number of initial tokens: 17
Best moves: [1, 2, 3]
Maximizing player takes: 2 tokens, remaining tokens: 15
Best moves: [2]
Minimizing player takes: 2 tokens, remaining tokens: 13
Best moves: [1, 2, 3]
Maximizing player takes: 1 tokens, remaining tokens: 12
Best moves: [3]
Minimizing player takes: 3 tokens, remaining tokens: 9
Best moves: [1, 2, 3]
Maximizing player takes: 3 tokens, remaining tokens: 6
```

```
Best moves: [1]
Minimizing player takes: 1 tokens, remaining tokens: 5
Best moves: [1, 2, 3]
Maximizing player takes: 3 tokens, remaining tokens: 2
Best moves: [1]
Minimizing player takes: 1 tokens, remaining tokens: 1
Best moves: [1]
Maximizing player takes: 1 tokens, remaining tokens: 0
Maximizing player loses
```

4 Przewaga gracza maksymalizującego

Dokonajmy teraz eksperymentu gdzie początkowa liczba żetonów to liczba losowana z zakresu 8-20 (z rozkładem jednostajnym), K=3 dla różnych głębokości. Wykonamy 1000 różnych iteracji dla każdej głębokości aby zmniejszyć wpływ losowości rozkładu jednostajnego na wynik. Z przykładu dla wysokich głębokości wiemy, że tylko 3 rozpoczęcia są przegrywające dla gracza maks 77 procent gier powinno kończyć się wygraną gracza Maks.

range of distribution	K	depth	chances	average_time(sec)	iterations
(8,20)	3	0	50.7%	0.0024	1000
(8,20)	3	1	48.1%	0.0021	1000
(8,20)	3	2	49.3%	0.0009	1000
(8,20)	3	3	48.1%	0.0042	1000
(8,20)	3	4	55.9%	0.003	1000
(8,20)	3	5	56.2%	0.003	1000
(8,20)	3	6	61.2%	0.0038	1000
(8,20)	3	7	68.6%	0.0041	1000
(8,20)	3	8	70.5%	0.0056	1000
(8,20)	3	9	78.1%	0.0071	1000

Widzimy, że dla niższych głębokości czasami to nawet gracz minimalizujący ma przewagę, chociaż na przykład dla głębokości 3 dopiero od początkowej liczby żetonów wynoszącej 10 nie da się od początku ściśle określi ścieżki prowadzącej do wygranej jednego z graczy. Dla głębokości 8 gracz maksymalizujący wie że musi zabrać 3 żetony, wtedy gdy pozostanie 5 żetonów, niezależnie od wyboru gracza Min gra zakończy się w 4 ruchy. UWAGA: głębokość 0 nie oznacza, że gracz nie przegląda żadnego ruchu, implementacja naszej funkcji best move zakłada że gracz przeszukuje o 1 więcej ruchów w przód niż głębokość. Jest to pewna nieścisłość związana z własną interpretacja głębokości przeszukiwania. Można ją obronić tym, że głębokość zero oznacza, że algorytm jest w stanie ocenić ruchy tuż "pod"danym stanem. Dla większych głębokości zauważalne jest, że procent wygranej gracza zbiega do wartości określonej przy pełnym przeszukiwania drzewa, a wahania od tej wartości spowodowane są losowością wyboru poczatkowej liczby żetonów.

range of distribution	K	depth	chances	average_time(sec)	iterations
(8,20)	3	10	74.6%	0.0094	1000
(8,20)	3	11	79.3%	0.0117	1000
(8,20)	3	12	78.0%	0.0134	1000
(8,20)	3	13	77.9%	0.0141	1000
(8,20)	3	14	75.5%	0.0138	1000
(8,20)	3	15	77.4%	0.0146	1000
(8,20)	3	16	75.2%	0.0147	1000
(8,20)	3	17	77.5%	0.0148	1000
(8,20)	3	18	79.0%	0.0148	1000
(8,20)	3	19	77.5%	0.0138	1000

5 Wypływ przycinania Alfa-Beta na szybkość działania programu

Zbadamy jak bardzo implementacja przycinania Alfa-Beta przyspieszyła działanie Mini-Maxa.

range of distribution	K	depth	chances	average_time(sec)	iterations
(8,20)	3	0	46.0%	0.0	100
(8,20)	3	1	56.0%	0.0	100
(8,20)	3	2	51.0%	0.0001	100
(8,20)	3	3	48.0%	0.0002	100
(8,20)	3	4	54.0%	0.0006	100
(8,20)	3	5	60.0%	0.0014	100
(8,20)	3	6	67.0%	0.003	100
(8,20)	3	7	72.0%	0.006	100
(8,20)	3	8	69.0%	0.0097	100
(8,20)	3	9	81.0%	0.0221	100
(8,20)	3	10	82.0%	0.0186	100
(8,20)	3	11	71.0%	0.0203	100
(8,20)	3	12	82.0%	0.0466	100
(8,20)	3	13	73.0%	0.0421	100
(8,20)	3	14	82.0%	0.037	100
(8,20)	3	15	76.0%	0.0383	100
(8,20)	3	16	74.0%	0.0372	100
(8,20)	3	17	74.0%	0.0385	100
(8,20)	3	18	75.0%	0.0212	100
(8,20)	3	19	75.0%	0.0464	100

Tabela przedstawia wyniki dla algorytmu mini-max bez przeszukiwania alfa-beta. Widzimy, że czas średni czas wykonania jednej gry wrasta jednak dalej dla tak prostej gry z tak mało rozbudowanym drzewem przeszukiwań są to mało znaczące opóźnienia.

6 Podsumowanie i wnioski

Podstawową wadą w naszym algorytmie wynikającej po trochu z właściwości wygranych gry, jest uboga funkcja oceny i brak heurystyki dla stanów nieterminalnych - byłaby one w naszym przypadku trudna do zaimplementowania z racji tego, że nie da się określić przypisać oceny danemu stanowi jak na przykład ma to miejsce w szachach, możemy określić po prostu czy jest on wygrywający czy też nie. Można by rozszerzyć naszą funkcje o przypisanie np. wartości 1 dla stanów 2,3,4,6,7,8 a -1 dla 5,9,13 itd, ponieważ wiemy już które z tych pozycji są wygrywające dla którego gracza. Pozwoliłoby to na pewno na otrzymywanie poprawnych wyników dla eksperymentów z większą ilością początkowych żetonów przy jednoczesnym zmniejszeniu głębokości, ponieważ algorytm nie musiałby dochodzić do stanu 0 aby przypisać ocenę innemu stanowi "wyżej". Implementacja przycinania alfa-beta jest zaimplementowana w naszym algorytmie poprawnie jednak różnicę w czasach działania z i bez przycinania zauważamy dopiero dla głębokości większych od 10 i mimo że czasami algorytm działa nawet 4 razy wolniej to dalej, wykonanie nawet 100 iteracji nie przekracza paru sekund działania programu. Nasza gra jest po prostu zbyt mało złożona. Z innych oczywistych obserwacji, wraz ze wzrostem głłębkości rośnie czas działania operacji, ale także skuteczność jest wyższa.