Raport
Wprowadzenie do Sztucznej Inteligencji – Ćwiczenie 3.

Dwuosobowe gry deterministyczne

Autor: Aleksandra Jamróz, nr albumu: 310 708

Treść zadania

Program buduje drzewo gry dla gry typu Isolation na planszy 4x4 (NxN, gdzie N>=3). Wejściem programu jest wielkość planszy, położenie na niej obu graczy oraz maksymalna głębokość drzewa. Program wykorzystuje do tego algorytm minimax.

Schemat gry, główne zasady oraz przyjęte założenia:

- Gra *Isolation* to dwuosobowa, deterministyczna gra planszowa;
- Jej plansza podzielona jest na kwadratowe pola jak szachownica;
- Każdy z graczy rozporządza jednym przypisanym pionkiem;
- Każdy z pionków ma przypisane pole początkowe;
- Podczas tury danego gracza, może on poruszyć się pionkiem o 1 pole: do przodu, do tyłu, na boki oraz na skos;
- Wszystkie pola są początkowo dostępne dla obu graczy. Po opuszczeniu przez pionek danego pola, zostaje ono wyłączone z gry;
- Pionki nie mogą przesunąć się na pole obecnie zajęte oraz na pole wyłączone z gry.
- Gra kończy się w momencie, kiedy jeden z pionków nie może wykonać następnego ruchu;
- W sytuacji, gdy oba pionki znajdują się na polach bez dalszej możliwości ruchu, zwycięża pionek, który jako drugi przesunął się na takie pole. (oznacza to, że nie przyjmuję remisu jako wariantu zakończenia gry)

Biblioteki

Rozwiązanie rozpoczęłam od zaimportowania odpowiednich bibliotek. Wykorzystałam:

- Copy – w celu generowania dzieci stanów
- Pygame – do wizualizacji przebiegu gry

Implementacja – klasa *Field*

Rozwiązanie zadanie zaczęłam od zaimplementowania logiki gry (plik field.py). Stworzyłam klasę reprezentującą obecny stan gry. Aby go dokładnie opisać, potrzebne są następujące elementy:

- rozpiętość planszy (liczba pól w każdej kolumnie i wierszu) (*self.rows);*
- aktualne współrzędne pionków (*self.x1 … self.y2*);
- lista pól (lista list zbudowana jak siatka) wraz ze współrzędnymi każdego pola oraz informacją o tym, czy jest dostępne dla pionków (*self.squares_list);*

- tryb gry – decyduje o sposobie i kolejności poruszania się pionków sterowanych przez komputer (*self.mode*). Wszystkie tryby:
    - 1. minimax vs minimax – domyślny;
    - 2. random vs random;
    - 3. minimax vs random;
    - 4. random vs minimax.
- głębokość drzewa – przyjmowana potem jako parametr przez algorytm minimax (*self.depth*). Domyślna głębokość = 0;

Konstruktor klasy *Field* wyżej wymienione elementy jako parametry. Poza nimi, klasa tworzy kolejne pola:

- *self.is_max* - informacja o tym, czyj ruch jest obecnie. Przyjmuje wartości True i False. True oznacza pionka rozpoczynającego grę;
- *self.move_to_safe* – pole zapamiętanie jako ruch wykonany podczas tworzenia dziecka (czym jest dziecko opiszę w dalszej części raportu);
- *self.winner* – True/False – informacja, czy gra ma już zwycięzcę i należy ją skończyć.

Zaimplementowane metody klasy planszy:

- *self.move* – (void) - główna metoda pozwalająca na grę. Jako parametr przyjmuje pionek oraz współrzędne pola, na które ma zostać przesunięte. Zamienia współrzędne pionka na podane jako parametr oraz zmienia status opuszczonego pola na wyłączone z gry, a zajętego – na zajęte;
- *self.change_turn* – (void) - zmienia turę zawodnika;
- *self.available_moves* – (-> lista) – zwraca listę pól na które może przesunąć się pionek w kolejnym ruchu. Jako parametr przyjmuje pionek, dla którego ma je znaleźć. Przyjęte założenia nie pozwalają na znalezienie pól znajdujących się poza planszą ani pól zajętych oraz wyłączonych z gry;
- *self.children_list* – (–> lista) – zwraca listę dzieci aktualnego stanu planszy;
- *self.is_winner* – (bool) – sprawdza czy gra ma już zwycięzcę. Oprócz tego aktualizuje pole *self.winner;*
- *self.describe* – (void) – metoda stworzona na potrzeby obserwacji przebiegu gry w konsoli. Wyświetla planszę w postaci siatki z listy *self.squares_list*;

Ruchy:

- *self.random_move* – (–> pole) – zwraca wylosowane pole z listy możliwych pól;
- *self.heur_move* – (–> pole) – zwraca pole wybrane przez algorytm minimax z listy możliwych pól. Jeżeli wystąpi sytuacja, że algorytm minimax zwróci jednakową najlepszą wartość dla więcej niż jednego pola, jedno z nich zostaje wylosowane;

Przeprowadzanie gry:

- *self.update* – (void) – w zależności od przyjętego trybu działa w określony trybu. Dostosowuje wybór pola zgodnie z trybem oraz aktualną kolejką i wykonuje ruch pionkiem. Przed każdym wykonaniem sprawdza, czy aktualny stan nie jest przypadkiem stanem terminalnym. W takim przypadku wyświetla na ekranie monit o zakończeniu gry i zwycięzcy;
- *self.play_game* – (–> zwycięzca gry) – rozgrywa partię sam ze sobą i zwraca zwycięzcę partii.

Heurystyki

W związku z implementacją algorytmu minimax, konieczne było dodanie funkcji oceniającej obecny stan gry. Zaimplementowałam kilka heurystyk, aby zobaczyć dla jakiej funkcji algorytm będzie działał najlepiej:

1. Heurystyka podstawowa – zwraca liczbę możliwych ruchów określonego pionka;
2. Heurystyka różnicowa – zwraca różnicę liczby możliwych ruchów pionków;
3. Heurystyka ofensywna – zwraca liczbę możliwych ruchów pionka pomniejszoną o dwukrotność liczby możliwych ruchów przeciwnika;
4. Heurystyka defensywna – zwraca podwojoną liczbę możliwych ruchów pionka pomniejszoną o liczbę możliwych ruchów przeciwnika.

Algorytm minimax

Pseudokod algorytmu minimax

```
def Minimax(s,d)            // d - głębokość przeszukiwania
if s ∈ T or d = 0 then
|       return h(s)         // heurystyka lub wypłata
end
U := successors(s)
for u in U do
|       w(u) = Minimax(u, d-1 )
end
if Max-move then            // ruch gracza Max
|       return max(w(u))
else
|       return min(w(u)) end
```

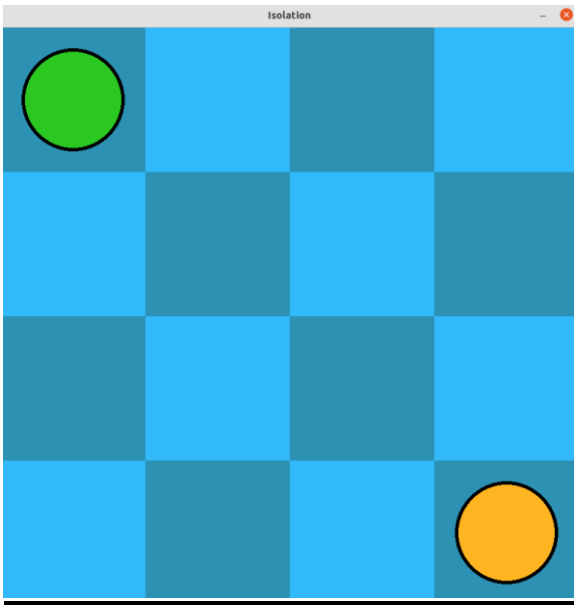
Implementacja algorytmu minimax znajduje się w pliku minimax.py.
Algorytm minimax to algorytm rekursywny. Operuje na podanym jako parametr stanie gry oraz przyjmuje głębokość budowanego drzewa. Działanie algorytmu może przebiegać inaczej w 3 wypadkach:

- podany stan gry przedstawia grę zakończoną – zwracana jest wartość dodatnia, jeżeli zwycięzcą jest gracz nr 1, ujemna jeżeli gracz nr 2;
- podana głębokość drzewa równa się 0 – zwracana jest wartość funkcji heurystycznej;
- gra nie jest skończona, a głębokość ma wartość dodatnią – algorytm tworzy listę wyników działania algorytmu dla każdego dziecka podanego stanu i zwraca największą bądź najmniejszą wartość w zależności od aktualnej kolejki.

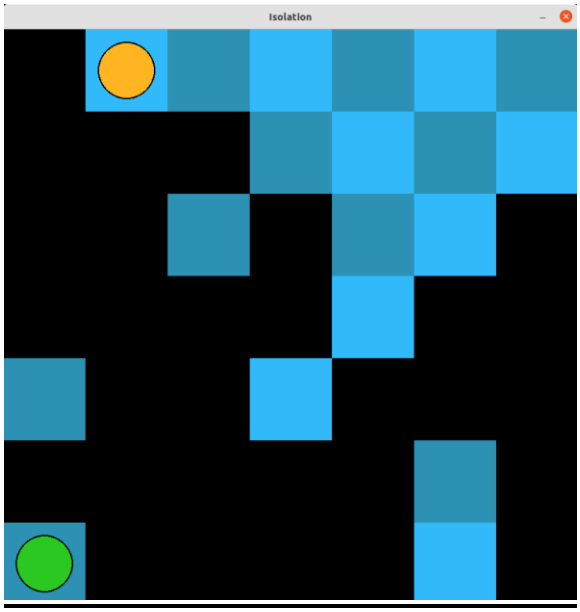
Dziećmi stanu określa się stany powstałe w wyniku przesunięcia aktualnego pionka na jedno z możliwych pól. Stan ma liczbę dzieci równą liczbie możliwych ruchów pionka. Implementacja zarówno heurystyk, jak i algorytmu znajduje się w pliku minimax.py.

Wizualizacja

Przebieg gry można obejrzeć po uruchomieniu programu. Jest to możliwe dzięki wykorzystaniu biblioteki pygame. Na ekranie wyświetla się okienko przedstawiającą planszę z niebieskimi polami. Pionki: zielony i pomarańczowy poruszają się i pozostawiają za sobą czarne kwadraty oznaczające pola wyłączone z gry.



Rysunek 1. Przykładowy układ początkowy na planszy 4x4



Rysunek 2. Układ końcowy na planszy 7x7

Przeprowadzanie prób

Dla każdej heurystyki i każdego trybu gry przeprowadziłam oddzielną serię prób. Polegały na stukrotnym uruchomieniu gry o danych parametrach i zwróceniu proporcji zwycięstw pionka pierwszego do zwycięstw pionka drugiego. Zastosowałam również różne wartości głębokości drzewa. Za każdym razem współrzędne punktów były losowane na nowo. Wszystkie gry odbywały się na planszy 4x4.

Obserwacje

Tryb gry:

I. Minimax vs minimax

Tryb	Głębokość	Gracz 1.	Gracz 2.	Heurystyka
1	1	56	44	Podstawowa
1	3	68	32	
1	5	60	40	
1	1	51	49	Różnicowa
1	3	54	46	
1	5	63	37	
1	1	53	47	Defensywna
1	3	64	36	
1	5	58	42	
1	1	55	45	Ofensywna
1	3	61	39	
1	5	61	39	

Tabela 1. Liczba zwycięstw dla prób przeprowadzonych dla gier minimax vs. Minimax

W większości przypadków wyniki są rozłożone równomiernie między gracza pierwszego i drugiego. Można jednak zauważyć delikatny przechył na stronę gracza pierwszego, co

znaczy, że algorytm nie daje jednakowych szans obu pionkom. Faworyzacja jednego z nich może świadczyć o istnieniu strategii wygrywającej dla niektórych ułożeń początkowych na planszy. Każda z heurystyk zachowuje się w podobny sposób.

II. Random vs random

Kiedy pionki poruszały się na losowe dostępne pola na planszy, wyniki rozkładały się równomiernie: ok. 50% zwycięstw pionka rozpoczynającego grę, 50% drugiego. Nie przeprowadzałam prób w zależności od głębokości ani heurystyki, gdyż nie mają one znaczenia dla wyniku takiej gry. Obserwacje opieram na kilkunastu próbach przeprowadzonych podczas testowania gry.

III. i IV. Random vs minimax oraz minimax vs random

Przewaga algorytmu minimax nad algorytmem losowym jest znacząca. Dla każdego rodzaju heurystyki różnica między zwycięstwami zaimplementowanego algorytmu wyróżnia się, nawet jeżeli głębokość drzewa jest niewielka. Oczywiście wraz ze wzrostem wartości głębokości rośnie skuteczność.

Tryb	Głębokość	Gracz 1.	Gracz 2.	Heurystyka
3	1	69	31	Podstawowa
3	3	89	11	
3	5	97	3	
3	1	88	12	Różnicowa
3	3	94	6	
3	5	98	2	
3	1	77	23	Defensywna
3	3	93	7	
3	5	100	0	
3	1	96	4	Ofensywna
3	3	96	4	
3	5	98	2	

Tabela 2. Próby przeprowadzone dla gier minimax - gracz 1. vs random - gracz 2.

Tryb	Głębokość	Gracz 1.	Gracz 2.	Heurystyka
4	1	29	71	Podstawowa
4	3	13	87	
4	5	3	97	
4	1	11	89	Różnicowa
4	3	6	94	
4	5	4	96	
4	1	18	82	Defensywna
4	3	5	95	
4	5	2	98	
4	1	14	86	Ofensywna
4	3	10	90	
4	5	1	99	

Tabela 3. . Próby przeprowadzone dla gier random - gracz 1. vs minimax- gracz 2.

Analizując tabele wyników można dojść do wniosku, że najlepiej sprawdza się heurystyka ofensywna, która nawet dla małej głębokości drzewa daje fenomenalne wyniki – rzędu 90% zwycięstw, a dla większej – nawet 99%. Porównywalne wyniki osiągają heurystyki defensywna i różnicowa – średnie wartości są nieznacznie mniejsze niż dla heurystyki ofensywnej. Zwróćmy uwagę jednak, że przy głębokości drzewa równej 5, heurystyce defensywnej udało się osiągnąć (!) 100% zwycięstw. Najgorzej sprawdza się heurystyka podstawowa, co nie jest zaskoczeniem, ponieważ w żaden sposób nie porównuje swoich predykcji do przeciwnika. Zwłaszcza widoczne jest to dla niewielkiej głębokości drzewa, ponieważ wtedy algorytm działa niemalże jak losowy. Dla większych wartości różnice między innymi heurystykami w pewnym stopniu się zacierają.
Należy zauważyć, że wszystkie próby podlegają marginesowi błędu, ponieważ pola początkowe za każdym razem generowane były na nowo. Pomimo stosunku liczby prób do liczby możliwych ułożeń, nie zawsze punkty mogły zostać rozrzucone równomiernie. Dodatkowo z niewielkich rozmiarów planszy wynikają niewielkie możliwości ruchów pionków. Nietrudno w takim przypadku o ułatwienie gry jednemu z nich, z czego może wynikać zaobserwowana faworyzacja gracza pierwszego.