Wprowadzenie do sztucznej inteligencji: Lista 1

Radosław Wojtczak

numer indeksu: 254607

0.1 Problematyka

Problematyką zadania było zastosowanie algorytmu A* (A-gwiazdka) do rozwiązania popularnej gry "Piętnastka". Piętnastka to układanka zbudowana z pudełka, w którym znajduje się 15 kwadratowych klocków o jednakowych rozmiarach ułożonych w kwadrat 4×4 i ponumerowanych od 1 do 15. Jedno miejsce jest puste i umożliwia przesuwanie sąsiednich klocków względem siebie. Oczywiście to pojęcie rozszerza się na więcej wymiarów, co również zostało wzięte pod uwagę w stworzonym przeze mnie programie. Dodatkowym wymaganiem było użycie dwóch róznych funkcji heurystycznych.

0.2 Specyfikacja implementacji

Program został napisany w pythonie z wykorzystaniem dwóch klas:

1. Klasa "Fifteen" zajmująca się przeprowadzaniem rozgrywki, wykonywaniem wszystkich niezbędnch obliczeń oraz przedstawianiem niezbędnych komunikatów. Plansza jest przedstawiona przy pomocy tablicy jednowymiarowej (początkowo chciałem wykonać zadanie przy pomocy tablicy dwuwymiarowej, jednakże czas wykonywania programu, prawdopodobnie ze względu na błędy implementacyjne, był zdecydowanie za duży). Pusty klocek oznaczony jest ostatnim numerem (szerokość*wysokosc).

2. Klasa "Node" przedstawiająca pojedynczy stan planszy (w terminologii grafów reprezentuje węzeł, stąd nazwa). Zawiera takie informacje jak wartość funkcji heurystycznej dla danego stanu, numer ruchu oraz odwołanie do stanu, z którego do niego przybyliśmy (funkcjonalność wykorzystywana do przedstawienia drogi jaką trzeba pokonać od stanu startowego do ułożenia planszy)

Funkcjonalność programu została rozszerzona o rozwiązywanie stanu gry nawet w sytuacjach, gdy puste pole nie znajduje się w prawym dolnym rogu.
Ze względów objętościowych powyższy opis przedstawia jedynie zarys programu. Szczegółowy opis wszystkich specyfikacji został pominięty i zostanie przedstawiony w trakcie oddawania programu.

0.3 Generowanie permutacji

W programie za generowanie permutacji są odpowiedzialne dwie funkcje: *makePermutation()* i *checkPermutation()*. Pierwsza z nich kreuje plansze, druga natomiast dokonuje losowań aż do momentu, gdy plansza będzie rozwiązywalna.
Kiedy plansza jest rozwiązywalna?
INWERSJA- Inwersją w tym przypadku nazywamy parę liczb a_i, a_j taką, że $i < j$ jeżeli $a_i > aj$
i,j-indeksy w tabeli

- Jeśli szerokość planszy jest liczbą nieparzystą to liczba inwersji jest liczbą parzystą.

- Jeśli szerokość jest liczbą parzystą i pusty kafelek znajduje się w nieparzystym rzędzie licząc od dołu to liczba inwersji musi być liczbą parzystą

- Jeśli szerokość jest liczbą parzystą i pusty kafelek znajduje się w parzystym rzędzie licząc od dołu, to liczba inwersji musi być liczbą nieparzystą.

0.4 Użyte heurystyki

1. Heurystyka określona w programie jako *distance_heuristic()* to heurystyka, która dla każdego kafelka zwraca jego odległość względem spodziewanego końcowego położenia. Odległość oznacza sumę kafelków w pionie jak i w poziomie, jaką kafelek musi pokonać aby dotrzeć do celu. Nie bierzemy pod uwagę ruchów "na ukos", gdyż ów ruchy w tej grze są nielegalne. Wynikiem zwracanym przez powyższą funkcję jest wysumowana odległość.

2. Heurystyka określona w programie jako *linear_heuristic()* to lekko rozbudowana wersja poprzedniej heurystyki. Do wcześniej obiczonej wartości (otrzymanej przez funkcję *distance_heuristic()*) dodajemy jeszcze liczbę "liniowych konfliktów" występujących między kafelkami. Jako liniowy konflikt uznajemy sytuację, gdy dwa kafelki znajdują się w tym samym rzędzie (kolumnie) i miejsca docelowe znajdują się w tym samym rzędzie (kolumnie) jednak jeden z kafelków przeszkadza drugiemu w dojściu do celu. Do rozwiązania tego konfliku potrzeba dwóch ruchów (w pierwszym kafelek przeszkadzający musi ustąpić miejsca, w drugim rozpatrywany kafelek zajmuje swoje miejsce), także liczbę występujących konfliktów mnożymy razy 2. Ostatecznie otrzymujemy wzór

$$linear\_heuristic() = distance\_heuristic() + 2 * liczba\_konfliktw \quad (1)$$

W pliku znajduje się również implementacja heurystyki *misplaced_heuristic()*, jednakże ze względu na bardzo słabe wyniki nie będzie ona analizowana w dalszej części sprawozdania.

0.5 Wyniki i interpetacja

UWAGA: Plansze przedstawione w formie tablicy jednowymiarowej.
Kolejność: od góry do dołu, od lewej do prawej
Poniższe tabele przedstawiają wyniki dla wykonanych testów:

1. Liczba odwiedzonych stanów

2. Liczba wykonanych przesunięć

3. Czas

UWAGA: Wyniki otrzymane w tabeli 3 mogą nie oddawać rzeczywistej szybkości wykonywania się powyższych algorytmów. Może być to spowodowane miedzy innymi wykonywaniem innych czynności na komputerze w trakcie pracy algorytmu.
Interpretacja: Otrzymane wyniki pokazują, iż wiele zmiennych wpływa na wydajność działania programu. Przede wszystkim otrzymane wyniki w tabelach **3** i **1** pokazują, iż użyta heurstyka ma spore znaczenie i dobór odpowiedniej w znacznym stopniu może przyśpieszyć pracę naszego programu. Z rozpatrywanych heurystyk zauważamy, iż mimo mniejszej liczby odwiedzonych stanów

Plansza	*distance_ heuristic()*	*linear_ heuristic()*
[14, 5, 12, 3, 13, 1, 9, 11, 16, 7, 8, 4, 2, 10, 6, 15]	592 552	399 793
[1, 3, 16, 15, 5, 7, 11, 4, 9, 2, 8, 12, 13, 10, 14, 6]	11 273	8 025
[2, 5, 1, 3, 6, 8, 11, 12, 16, 9, 7, 14, 13, 10, 15, 4]	30 751	7 007
[13, 1, 2, 10, 5, 3, 11, 7, 9, 6, 16, 8, 14, 15, 4, 12]	87 411	15 520
[14, 1, 2, 8, 5, 6, 4, 3, 9, 7, 11, 10, 13, 12, 16, 15]	10 099	3 683
[8, 1, 7, 3, 13, 10, 6, 5, 2, 15, 14, 4, 9, 16, 11, 12]	143 005	40 519
[12, 14, 8, 9, 6, 4, 2, 10, 15, 16, 3, 11, 5, 13, 1, 7]	3 018 170	1 480 419
[14, 13, 2, 11, 4, 6, 5, 10, 3, 1, 16, 8, 7, 15, 9, 12]	1 906 763	552 249
[2, 10, 7, 3, 1, 5, 6, 9, 13, 14, 11, 4, 16, 15, 12, 8]	13 886	5 933
[5, 8, 6, 12, 10, 16, 13, 14, 3, 9, 1, 2, 4, 11, 15, 7]	507 118	231 267
[4, 2, 3, 7, 1, 6, 12, 8, 15, 9, 10, 11, 16, 5, 13, 14]	252 538	7 565
[16, 7, 1, 4, 3, 10, 11, 8, 5, 15, 14, 6, 2, 13, 9, 12]	55 523	28 150
Wartośćć średnia:	552 424	231 677

Table 1: Liczba odwiedzonych stanów

Plansza	*distance_ heuristic()*	*linear_ heuristic()*
[14, 5, 12, 3, 13, 1, 9, 11, 16, 7, 8, 4, 2, 10, 6, 15]	46	46
[1, 3, 16, 15, 5, 7, 11, 4, 9, 2, 8, 12, 13, 10, 14, 6]	30	32
[2, 5, 1, 3, 6, 8, 11, 12, 16, 9, 7, 14, 13, 10, 15, 4]	34	34
[13, 1, 2, 10, 5, 3, 11, 7, 9, 6, 16, 8, 14, 15, 4, 12]	36	36
[14, 1, 2, 8, 5, 6, 4, 3, 9, 7, 11, 10, 13, 12, 16, 15]	31	31
[8, 1, 7, 3, 13, 10, 6, 5, 2, 15, 14, 4, 9, 16, 11, 12]	40	40
[12, 14, 8, 9, 6, 4, 2, 10, 15, 16, 3, 11, 5, 13, 1, 7]	55	55
[14, 13, 2, 11, 4, 6, 5, 10, 3, 1, 16, 8, 7, 15, 9, 12]	52	52
[2, 10, 7, 3, 1, 5, 6, 9, 13, 14, 11, 4, 16, 15, 12, 8]	33	33
[5, 8, 6, 12, 10, 16, 13, 14, 3, 9, 1, 2, 4, 11, 15, 7]	52	52
[4, 2, 3, 7, 1, 6, 12, 8, 15, 9, 10, 11, 16, 5, 13, 14]	39	37
[16, 7, 1, 4, 3, 10, 11, 8, 5, 15, 14, 6, 2, 13, 9, 12]	38	38
Wartość średnia:	40.5	40.5

Table 2: Liczba kroków potrzebna do zwycięstwa

linear_ heuristic średnio potrzebuje więcej czasu na rozwiązanie układanki, jednakże trzeba wziąć poprawkę na to, iż inny sposób implementacyjny może wpłynąć na osiągi powyższych heurystyk.
Dodatkowo zauważamy, że średnie wartości potrzebnych ruchów do rozwiązania układanki są równe, bliskie liczbie 40. Biorąc pod uwagę, iż górna granica liczby ruchów dla układanki o wymiarach 4x4 to 80, stwierdzenie, iż powyższe heurystyki zostały odpowiednio zaimplementowane wydaje się być poprawne. Ponadto w żadnym pojedynczym układzie startowym liczba ruchów nie przekroczyła wyżej wspomnianej liczby 80.

Plansza	*distance_ heuristic()*	*linear_ heuristic()*
[14, 5, 12, 3, 13, 1, 9, 11, 16, 7, 8, 4, 2, 10, 6, 15]	46.89s	112.1s
[1, 3, 16, 15, 5, 7, 11, 4, 9, 2, 8, 12, 13, 10, 14, 6]	0.79s	2.10s
[2, 5, 1, 3, 6, 8, 11, 12, 16, 9, 7, 14, 13, 10, 15, 4]	2.15s	1.65s
[13, 1, 2, 10, 5, 3, 11, 7, 9, 6, 16, 8, 14, 15, 4, 12]	7.24s	3.81s
[14, 1, 2, 8, 5, 6, 4, 3, 9, 7, 11, 10, 13, 12, 16, 15]	0.71s	0.93s
[8, 1, 7, 3, 13, 10, 6, 5, 2, 15, 14, 4, 9, 16, 11, 12]	11.35s	11.01s
[12, 14, 8, 9, 6, 4, 2, 10, 15, 16, 3, 11, 5, 13, 1, 7]	255.0s	421.1s
[14, 13, 2, 11, 4, 6, 5, 10, 3, 1, 16, 8, 7, 15, 9, 12]	152.9s	149.5s
[2, 10, 7, 3, 1, 5, 6, 9, 13, 14, 11, 4, 16, 15, 12, 8]	0.96s	1.71s
[5, 8, 6, 12, 10, 16, 13, 14, 3, 9, 1, 2, 4, 11, 15, 7]	42.3s	62.6s
[4, 2, 3, 7, 1, 6, 12, 8, 15, 9, 10, 11, 16, 5, 13, 14]	20.7s	1,82s
[16, 7, 1, 4, 3, 10, 11, 8, 5, 15, 14, 6, 2, 13, 9, 12]	4.64s	7.43s
Wartość średnia:	45.5s	64.5s

Table 3: Czas potrzebny do ułożenia układanki