WOJSKOWA AKADEMIA TECHNICZNA

Sprawozdanie z przedmiotu:

Sztuczna Inteligencja

Prowadzący: mgr inż. Przemysław Czuba

Autor: Mateusz Jasiński WCY20IY2S1

Zaimplementowany algorytm A*

```
def run_astar(maze_game, current_point, visited_points):
    # TODO: Dla podstawowej postaci labiryntu wynik powinien być identyczny
jak w BFS.
    queue = [current_point]
    while queue:
        current_point = queue.pop()
        if not is_in_visited_points(current_point, visited_points):
            visited_points.append(current_point)
            if maze_game.get_current_point_value(current_point) == '*':
                return current_point
            else:
                neighbors = maze_game.get_neighbors(current_point)
                for neighbor in neighbors:
                    neighbor.set_parent(current_point)
                    neighbor.set_cost(determine_cost(current_point,
neighbor))
                    queue.append(neighbor)
                queue.sort(key=lambda x: x.cost, reverse=True)
    return "No path to the goal found"


# Determine cost based on the distance to root
def determine_cost(current_point, neighbor):
    #TODO
    return mp.get_path_length(neighbor) + mp.get_move_cost(current_point,
neighbor)
```

Algorytm A* działa następująco:

1. Tworzymy kolejkę.
2. Powtarzamy poniższe punkty, aż kolejka stanie się pusta:
    a. Przypisujemy wartość (punkt) na początku kolejki do zmiennej.
    b. Sprawdzamy czy obecnie rozważana wartość była odwiedzana, jeżeli nie była to:
        i. Dodajemy punkt do odwiedzonych punktów.
        ii. Jeżeli odwiedzany punkt jest celem to zwracamy jego wartość, a w przeciwnym wypadku tworzymy listę sąsiadów punktu i dla każdego z nich:
            - Ustawiamy atrybut „rodzic” sąsiada na obecny punkt.
            - Obliczamy i ustawiamy atrybut „koszt” dla sąsiada
            - Dodajemy sąsiada do kolejki.
        iii. Sortujemy kolejkę rosnąco względem atrybutu „koszt”.
3. Jeżeli punkt 2. zakończył się nie zwracając punktu celu, zwracamy stosowny komunikat.

Jedynym problem napotkanym przeze mnie podczas implementacji algorytmu A* było posortowanie kolejki deque, która nie posiada metody sort(). Rozwiązałem to zmieniając kolejkę deque na listę, która metodę sort() posiada, a zasada jej działania jest identyczna.

Porównanie A*, DFS i BFS

Wymiary mapy	A*			DFS			BFS		
	Czas [ms]	Droga	Koszt	Czas [ms]	Droga	Koszt	Czas [ms]	Droga	Koszt
5x5	1	10	34	1	10	34	1	10	34
10x10	4	16	44	2	30	114	2	16	44
25x25	30	47	139	7	207	939	14	47	139
50x50	281	168	644	93	654	3074	147	168	644
90x90	2077	251	899	1152	2751	13399	1432	251	899
250x250	134456	990	3958	-	-	-	95193	990	3958

Algorytm A* wykonuje się znacznie wolniej od algorytmów DFS i BFS, jest to spowodowane dodatkowymi obliczeniami i sortowaniem listy.
Algorytm A* zwraca wyniki identyczne do BFS, ale potrzebuje ona zdecydowanie mniej pamięci, ze względu na mniejszą liczbę przeszukiwanych wartości

Sposoby wyznaczania wartości heurystycznych:

- Metoda wzrostu:
  Operator podejmuje działanie w stosunku do stanu aktualnego, aby otrzymać z niego kolejny stan.
  1. Wygeneruj stan początkowy (CS)
  2. Jeżeli CS jest stanem końcowym to zwróć „Sukces” i zakończ.
  3. Wybierz operator, który nie był jeszcze używany dla stanu CS i wygeneruj przy jego pomocy stan NS.
  4. Jeżeli:
     - NS jest stanem końcowym to zwróć „Sukces” i zakończ.
     - NS nie jest stanem końcowym, ale jest lepszy od CS to CS=NS.
  5. Powróć do 3.

  Strategia ta jest stosowana do zagadnień ciągłych oraz dyskretnych.

- Metoda najszybszego wzrostu:
  1. Wygeneruj stan początkowy (CS).
  2. Jeśli CS jest stanem końcowym to zwróć „Sukces” i zakończ.
  3. Dla <u>każdego</u> operatora, który nie był jeszcze używany wygeneruj stan NS.
  4. Jeżeli:
     - NS jest stanem końcowym to zwróć „Sukces” i zakończ.
     - NS nie jest stanem końcowym, ale jest lepszy od CS to CS=NS.
  5. Powróć do 3.

W pierwszej metodzie przechodzi się ze stanu bieżącego do pierwszego stanu, dla którego wartość funkcji oceniającej okazała się lepsza.

W drugiej metodzie sprawdza się dla każdego stanu wszystkie operatory i wybiera się ten, który daje najlepszy nowy stan.

Algorytm Minimax

Algorytm Minimax polega na przewidywaniu n ruchów do przodu i wybieranie na tej podstawie optymalnego ruchu.
Liczba ruchów „do przodu”, które ten algorytm wykonuje nazywana jest „Search depth”, czyli dosłownie głębokość przeszukiwania. Wartość tego parametru drastycznie zmienia czas, który algorytm potrzebuje do podjęcia decyzji.
Na przykładzie gry w „czwórki”, zauważyłem, że wraz ze wzrostem parametru „Search depth” wzrasta czas potrzebny na podjęcie decyzji oraz pamięć do tego potrzebna. Dzieje się tak gdyż algorytm musi stworzyć wiele scenariuszy możliwych ruchów, co zajmuje mnóstwo czasu i pamięci, np. dla parametru „Search depth” równego 7 początkowe ruchy AI zajmowały powyżej 15 sekund.
Warte wspomnienia jest to, że po każdym kolejnym ruchu czas potrzebny na podjęcie decyzji spada, a dzieje się to z powodu malejącej liczby możliwych ruchów, czyli scenariuszy, które trzeba przewidzieć.

Algorytm Alfa-Beta pruning

Algorytm przeszukujący, redukujący liczbę węzłów, które muszą być rozważane w drzewach przeszukujących przez algorytm minimax. Posiada dwa parametry: alfa, beta.
Alfa przez maksymalizację ocen węzłów potomnych stwierdza, czy możemy zakończyć ocenę węzła potomnego. Beta ocenia węzeł przez minimalizację ocen węzłów potomnych. Początkowo alfa jest -∞, a beta +∞. W miarę przeszukiwania przedział (alfa; beta) staje się mniejszy i kiedy beta staje się mniejsze niż alfa, oznacza to, że obecna pozycja nie może być wynikiem najlepszej zagrywki przez obu graczy i wskutek tego nie ma potrzeby przeszukiwania głębiej.
Algorytm Alfa-Beta jest wykorzystywany w sztucznej inteligencji, w grach dwuosobowych, takich jak kółko i krzyk, szachy i „czwórki".
Stosowany jest on ze względu na o wiele krótszy czas wykonania niż algorytm minimax przy większych wartościach „Search depth".

Porównanie algorytmu Minimax i Alfa-Beta

Search depth	Tura	Czas wykonania Minimax [ms]	Czas wykonania Alfa-Beta [ms]
2	1	8	7
	2	8	7
	3	8	7
3	1	40	40
	2	39	37
	3	36	38
4	1	200	194
	2	196	188
	3	194	186
5	1	989	1036
	2	974	772
	3	905	389
6	1	4831	4788
	2	4644	2789
	3	4292	292
7	1	23942	923
	2	22303	3563
	3	18133	215
100	1	-	150
	2	-	149
	3	-	93

Z powyższej tabelki możemy wnioskować, że wraz ze wzrostem parametru „Search depth” algorytm Minimax potrzebuje znacznie więcej czasu na podjęcie decyzji. Natomiast czas wykonania algorytmu Alfa-Beta wzrasta do pewnego momentu, a następnie przestaje, bądź nawet maleje. Algorytm ten jest w stanie podjąć decyzję w rozsądnym czasie nawet przy wartości 100 parametru „Search depth”, gdzie algorytm Minimax potrzebowałby prawdopodobnie kilka godzin.
Algorytm Alfa-Beta oraz Minimax mają podobne poziomy „inteligencji”, ale Alfa-Beta ma przewagę z powodu kolosalnie mniejszego czasu wykonania dla większych wartości „Search depth”, a jako, że parametr ten przekłada się na słuszność podejmowanych decyzji (większa wartość parametru, oznacza bardziej „przemyślane” decyzje) uważam, że algorytm Alfa-Beta jest bardziej „inteligentny”, a przynajmniej „inteligentny” szybciej.