Paweł Jędrzejczyk 224819

Podstawy sztucznej inteligencji
Laboratorium 1
Rozwiązywanie problemów
poprzez przeszukiwanie przestrzeni stanów

Definicja problemu

Zdefiniowany jest zbiór n miast, każde miasto posiada współrzędne x i y, określające jego położenie na płaszczyźnie. Zakładamy, że każde dwa miasta są połączone drogą, której przebycie wiąże się z poniesieniem kosztu, oraz że koszt podróży po danej trasie w obie strony jest taka sama. Za koszt pokonania drogi uznajemy odległość pomiędzy miastami na płaszczyźnie. Tak zdefiniowany problem możemy przedstawić w postaci pełnego grafu ważonego, którego węzłami są miasta, a krawędziami drogi pomiędzy miastami.
Komiwojażer zaczyna podróż w dowolnym z miast. Jego celem jest odwiedzenie każdego z miast dokładnie jeden raz oraz powrót do miasta początkowego, przy możliwie najmniejszym koszcie.
Jako stan rozumiemy częściową lub całkowitą ścieżkę – w postaci uporządkowanej listy kolejno odwiedzanych miast wraz z przypisaną wartością funkcji oceny stanu. Przestrzenią dopuszczalnych rozwiązań jest dla nas drzewo, w którego korzeniu znajduje się miasto początkowe, a stany potomne tworzone są poprzez przejście jednej z dostępnych krawędzi grafu.
Należy rozwiązać problem
- metodą pełnego przeszukania przestrzeni stanów – w głąb oraz wszerz.
- metodą zachłanną
- metodą A*

Rozwiązanie i opis

Aby móc zrealizować implementację algorytmów wykorzystywanych w rozwiązywaniu problemu komiwojażera niezbędna jest jakaś reprezentacja odwiedzanych węzłów. W tym celu do reprezentowania miast stworzona została klasa City_node. Poza nazwą i współrzędnymi kartezjańskimi zawiera ona kilka atrybutów niezbędnych dla działania zaimplementowanych algorytmów.

```
class City_node:
    def __init__(self, name, x, y):
        self.name = name
        self.x = x
        self.y = y
        self.connected_nodes = []
        self.path = []
        self.total_estimation = 0
        self.cost = 0
```

Poza funkcjami realizującymi algorytmy wymagane dla rozwiązania problemów zaimplementowane zostało również kilka funkcji pomocniczych

```
def calculate_distance(node1: City_node, node2: City_node):
    return math.sqrt(pow(node1.x - node2.x, 2) + pow(node1.y - node2.y, 2))
```

calculate_distance do obliczania odległości pomiędzy dwoma punktami na płaszczyźnie

```
def generate_cities(n: int):
    cities_list = []
    for i in range(0, n):
        cities_list.append(City_node("City_" + str(i + 1), random.randrange(0,
10), random.randrange(0, 11)))
    return cities_list
```

generate_cities do generowana listy miast (oraz samych miast) o współrzędnych z przedziału <0, 10 >. Liczba miast determinowana jest przekazywaną do funkcji wartością.

Przeszukiwanie wszerz

```
def bfs(root: City_node, graph: list):
    root_cpy = City_node(root.name, root.x, root.y)
    nodes_to_visit = [root_cpy]
    best_path = []
    best_cost = -1
    while len(nodes_to_visit) > 0:
        current_node = nodes_to_visit.pop(0)
        current_node.path.append(current_node.name)
        if len(current_node.path) == len(graph):
            current_node.path.append(root.name)
            current_node.cost += calculate_distance(current_node, root)
        if len(current_node.path) == len(graph) + 1:
            if best_cost < 0 or current_node.cost < best_cost:
                best_cost = current_node.cost
                best_path = current_node.path
        else:
            for child in graph:
                if child.name not in current_node.path:
                    new_node = City_node(child.name, child.x, child.y)
                    new_node.cost = current_node.cost +
calculate_distance(current_node, new_node)
                    new_node.path = list(current_node.path)
                    nodes_to_visit.append(new_node)
    return [best_path, best_cost]
```

Przeszukiwanie w głąb

```
def dfs(root: City_node, graph: list):
    root_cpy = City_node(root.name, root.x, root.y)
    nodes_to_visit = [root_cpy]
    best_path = []
    best_cost = -1
    while len(nodes_to_visit) > 0:
        current_node = nodes_to_visit.pop(0)  # 0
        current_node.path.append(current_node.name)
        if len(current_node.path) == len(graph):
            current_node.path.append(root.name)
            current_node.cost += calculate_distance(current_node, root)
        if len(current_node.path) == len(graph) + 1:
            if best_cost < 0 or current_node.cost < best_cost:
                best_cost = current_node.cost
                best_path = current_node.path
        else:
            temp_nodes_to_visit = []
            for child in graph:
                if child.name not in current_node.path:
                    new_node = City_node(child.name, child.x, child.y)
                    new_node.cost = current_node.cost +
calculate_distance(current_node, new_node)
```

```
                    new_node.path = list(current_node.path)
                    temp_nodes_to_visit.append(new_node)
            nodes_to_visit = temp_nodes_to_visit + nodes_to_visit
    return [best_path, best_cost]
```

Metoda zachłanna

```
def greedy(root: City_node, graph: list):
    root_cpy = City_node(root.name, root.x, root.y)
    nodes_to_visit = [root_cpy]
    best_path = []
    best_cost = -1
    while len(nodes_to_visit) > 0:
        current_node = nodes_to_visit.pop(0)  # 0
        current_node.path.append(current_node.name)
        if len(current_node.path) == len(graph):
            current_node.path.append(root.name)
            current_node.cost += calculate_distance(current_node, root)
        if len(current_node.path) == len(graph) + 1:

            if best_cost < 0 or current_node.cost < best_cost:
                best_cost = current_node.cost
                best_path = current_node.path
        else:
            temp_nodes_to_visit = []
            for child in graph:
                if child.name not in current_node.path:
                    new_node = City_node(child.name, child.x, child.y)
                    new_node.cost = current_node.cost +
calculate_distance(current_node, new_node)
                    new_node.path = list(current_node.path)
                    temp_nodes_to_visit.append(new_node)
            newlist = sorted(temp_nodes_to_visit, key=lambda x: x.cost,
reverse=False)
            nodes_to_visit.append(newlist[0])
    return [best_path, best_cost]
```

*A**

```
def a_star(root: City_node, graph: list):
    root_cpy = City_node(root.name, root.x, root.y)
    nodes_to_visit = [root_cpy]
    best_path = []
    best_cost = -1
    while len(nodes_to_visit) > 0:
        nodes_to_visit.sort(key=lambda x: x.total_estimation)
        current_node = nodes_to_visit.pop(0)  # 0
        current_node.path.append(current_node.name)
        if len(current_node.path) >= len(graph):
            if len(current_node.path) == len(graph) + 1:
                best_cost = current_node.cost
                best_path = current_node.path
                break
            new_node = City_node(root.name, root.x, root.y)
            new_node.cost = current_node.cost + calculate_distance(current_node,
new_node)
            new_node.path = list(current_node.path)
            new_node.total_estimation = new_node.cost +
calculate_distance(new_node, root)
            nodes_to_visit.append(new_node)
```

```
        else:
            temp_nodes_to_visit = []
            for child in graph:
                if (child.name not in current_node.path):
                    new_node = City_node(child.name, child.x, child.y)
                    new_node.cost = current_node.cost +
calculate_distance(current_node, new_node)
                    new_node.path = list(current_node.path)
                    new_node.total_estimation = new_node.cost +
calculate_distance(new_node, root)
                    temp_nodes_to_visit.append(new_node)
            nodes_to_visit = temp_nodes_to_visit + nodes_to_visit
    return [best_path, best_cost]
```

Wyniki

Działanie przedstawionych powyżej implementacji algorytmów zostało przetestowane na zestawach generowanych miast dla liczności <5, 11>.

```
for i in range(5, 12):
    graph = generate_cities(i)

    start_time = datetime.datetime.now()
    bfs_res = bfs(graph[0], graph)
    end_time = datetime.datetime.now()
    bfs_execution_time = (end_time - start_time).total_seconds() * 1000

    start_time = datetime.datetime.now()
    dfs_res = dfs(graph[0], graph)
    end_time = datetime.datetime.now()
    dfs_execution_time = (end_time - start_time).total_seconds() * 1000

    start_time = datetime.datetime.now()
    greedy_res = greedy(graph[0], graph)
    end_time = datetime.datetime.now()
    z_execution_time = (end_time - start_time).total_seconds() * 1000

    start_time = datetime.datetime.now()
    a_star_res = a_star(graph[0], graph)
    end_time = datetime.datetime.now()
    a_execution_time = (end_time - start_time).total_seconds() * 1000

    print("for " + str(i) + " nodes:"
          + "\n\tbfs time:    {0}ms".format(round(bfs_execution_time, 1)) +
"\tcost:{0} path: {1}".format(
        round(bfs_res[1], 3), bfs_res[0])
          + "\n\tdfs time:    {0}ms".format(round(dfs_execution_time, 1)) +
"\tcost:{0} path: {1}".format(
        round(dfs_res[1], 3), dfs_res[0])
          + "\n\tgreedy time: {0}ms".format(round(z_execution_time, 1)) +
"\tcost:{0} path: {1}".format(
        round(greedy_res[1], 3), greedy_res[0])
          + "\n\ta* time:     {0}ms".format(round(a_execution_time, 1)) +
"\tcost:{0} path: {1}".format(
        round(a_star_res[1], 3), a_star_res[0]))
```

Dla każdego algorytmu zaprezentowane na wyjściu zostały: czas działania algorytmu, całkowity koszt najkrótszej drogi oraz sama najkrótsza droga.

Przykładowy output

```
        bfs time:    0.0ms      cost:20.002 path: ['City_1', 'City_4', 'City_2', 'City_3', 'City_5',
'City_1']
        dfs time:    0.0ms      cost:20.002 path: ['City_1', 'City_4', 'City_2', 'City_3', 'City_5',
'City_1']
        greedy time: 0.0ms      cost:21.986 path: ['City_1', 'City_4', 'City_3', 'City_2', 'City_5',
'City_1']
        a* time:     1.0ms      cost:20.002 path: ['City_1', 'City_5', 'City_3', 'City_2', 'City_4',
'City_1']
for 6 nodes:
        bfs time:    0.0ms      cost:25.312 path: ['City_1', 'City_2', 'City_5', 'City_6', 'City_3',
'City_4', 'City_1']
        dfs time:    1.0ms      cost:25.312 path: ['City_1', 'City_2', 'City_5', 'City_6', 'City_3',
'City_4', 'City_1']
        greedy time: 0.0ms      cost:26.245 path: ['City_1', 'City_4', 'City_2', 'City_5', 'City_6',
'City_3', 'City_1']
        a* time:     2.0ms      cost:25.312 path: ['City_1', 'City_4', 'City_3', 'City_6', 'City_5',
'City_2', 'City_1']
for 7 nodes:
        bfs time:    4.0ms      cost:26.485 path: ['City_1', 'City_3', 'City_5', 'City_6', 'City_2',
'City_7', 'City_4', 'City_1']
        dfs time:    5.0ms      cost:26.485 path: ['City_1', 'City_3', 'City_5', 'City_6', 'City_2',
'City_7', 'City_4', 'City_1']
        greedy time: 0.0ms      cost:29.16 path: ['City_1', 'City_3', 'City_4', 'City_7', 'City_2',
'City_5', 'City_6', 'City_1']
        a* time:     10.9ms     cost:26.485 path: ['City_1', 'City_3', 'City_5', 'City_6', 'City_2',
'City_7', 'City_4', 'City_1']
for 8 nodes:
        bfs time:    41.9ms     cost:23.317 path: ['City_1', 'City_7', 'City_3', 'City_2', 'City_5',
'City_6', 'City_4', 'City_8', 'City_1']
        dfs time:    33.9ms     cost:23.317 path: ['City_1', 'City_7', 'City_3', 'City_2', 'City_5',
'City_6', 'City_4', 'City_8', 'City_1']
        greedy time: 0.0ms      cost:26.448 path: ['City_1', 'City_8', 'City_7', 'City_4', 'City_5',
'City_6', 'City_2', 'City_3', 'City_1']
        a* time:     72.8ms     cost:23.317 path: ['City_1', 'City_8', 'City_4', 'City_6', 'City_5',
'City_2', 'City_3', 'City_7', 'City_1']
for 9 nodes:
        bfs time:    829.6ms    cost:25.252 path: ['City_1', 'City_3', 'City_2', 'City_8', 'City_5',
'City_9', 'City_6', 'City_4', 'City_7', 'City_1']
        dfs time:    282.7ms    cost:25.252 path: ['City_1', 'City_3', 'City_2', 'City_8', 'City_5',
'City_9', 'City_6', 'City_4', 'City_7', 'City_1']
        greedy time: 0.0ms      cost:35.087 path: ['City_1', 'City_3', 'City_2', 'City_5', 'City_9',
'City_6', 'City_4', 'City_7', 'City_8', 'City_1']
        a* time:     428.8ms    cost:25.252 path: ['City_1', 'City_3', 'City_2', 'City_8', 'City_5',
'City_9', 'City_6', 'City_4', 'City_7', 'City_1']
for 10 nodes:
        bfs time:    139615.1ms cost:28.855 path: ['City_1', 'City_7', 'City_5', 'City_6', 'City_4',
'City_8', 'City_3', 'City_2', 'City_9', 'City_10', 'City_1']
        dfs time:    2644.5ms   cost:28.855 path: ['City_1', 'City_7', 'City_5', 'City_6', 'City_4',
'City_8', 'City_3', 'City_2', 'City_9', 'City_10', 'City_1']
        greedy time: 0.0ms      cost:30.638 path: ['City_1', 'City_9', 'City_2', 'City_10', 'City_3',
'City_8', 'City_4', 'City_6', 'City_5', 'City_7', 'City_1']
        a* time:     22324.5ms  cost:28.855 path: ['City_1', 'City_10', 'City_9', 'City_2', 'City_3',
'City_8', 'City_4', 'City_6', 'City_5', 'City_7', 'City_1']
```

Wnioski

Algorytmy pełnego przeszukiwania drzewa w głąb i wszerz ze względu na swoją naturę sprawdzają wszystkie możliwe ścieżki. Dzięki temu zawsze znajdują najlepszą możliwą ścieżkę, niestety sprawia to, że wraz ze wzrostem ilości węzłów czas ich działania znacznie rośnie. Zupełnie inaczej sytuacja się ma w przypadku algorytmu zachłannego. Dzięki temu, że zawsze wybiera najlepszą możliwą opcję w danym kroku i nie sprawdza ścieżek alternatywnych działa niesamowicie szybko. Niestety ceną za to jest nieoptymalne rozwiązanie (prawie zawsze).
Złotym środkiem jest algorytm A* który zawsze znajduje najlepsze rozwiązanie i tylko w najgorszym przypadku jest tak wolny jak przeszukiwanie DFS lub BFS.

Jak można zaobserwować na przykładowych wynikach działaniu zaimplementowanego rozwiązania, algorytm zachłanny zawsze jest najszybszy, wyprzedzając inne algorytmy o rzędy wielkości, dając często akceptowalnie dobre lub nawet takie same rezultaty (niezawarte w przykładowych rezultatach). Rozbieżności w czasach działania algorytmów BFS i DFS prawdopodobnie mają związek z wielkością tablicy stanów pośrednich w trakcie działania algorytmów (lub innym niezidentyfikowanym problemem implementacyjnym). Algorytm DFS jest nie tylko szybszy w przypadku zaprezentowanych implementacji, ale jednocześnie z natury mniej złożony pamięciowo.
Algorytm A*, który powinien być złotym środkiem sprawdził się niestety przeciętnie. Najprawdopodobniej wynika to ze sposobu implementacji. Mimo odnajdywania najlepszej możliwej ścieżki (a przynajmniej jednej z nich, co można zauważyć na prezentacji wyników działania – taki sam koszt, inna ścieżka) działa prawie zawsze najwolniej (czasami ustępując miejsca algorytmowi BFS). Czynnikiem mocno wpływającym na czas działania algorytmu jest zawarte w nim sortowanie.