AIML LAB 6

SAHIL BONDRE: U18CO021

Question

Implement A* algorithm and AO* algorithm in python without using existing libraries. Consider n cities and generate distance between each pair of city with the help of random function. You can provide range in random number generating function. Input/Output of the code:

When user enters 2 cities, program should provide the shorter path between the two cities using both algorithms separately.

Compare the complexity of both algorithms for same set of input.

Code

There several files involved in making up these algorithms work:

city.py: Defines the city model

```
from typing import Dict


class City:
    def __init__(self, x: int, y: int, name: str):
        self.x = x
        self.y = y
        self.name = name
        self.neighbours: Dict[City, float] = {}

    def __sub__(self, other):
        """Manhattea Distance"""
        return abs(self.x - other.x + self.y - other.y)

    def __hash__(self):
        return f"{self.x} {self.y}".__hash__()
```

```
    def __str__(self):
        return f"{self.name}{{{self.x} {self.y}}}"

    def print(self):
        return f"{self} => " + \
            " ".join(
                [f"({str(city)}, {self.neighbours[city]})" for city in
self.neighbours])

    def connect(self, city, distance: int):
        self.neighbours[city] = distance
        city.neighbours[self] = distance
```

comparator.py: Used to store the cost path and heuristic value to visit the cities

```
from city import City


class Comparator:
    def __init__(self, current: City, start_cost: int, end_cost: int):
        self.city = current
        self.start_cost = start_cost
        self.end_cost = end_cost

    def cost(self):
        return self.start_cost + self.end_cost

    def __lt__(self, other):
        return (self.cost()) - other.cost()
```

utils.py: defines utility functions for random numbers and connecting cities randomly

```
from city import City
import random
from typing import List


def random_number(start: int, end: int) -> int:
    return random.randint(start, end)
```

```
def connect_city_random(a: City, b: City):
    md = b - a
    d = random_number(md, 2 * md)
    a.connect(b, d)


def print_answer(title: str, arr: List):
    print(f"Solution for {title}: {arr}")
```

a_star.py: Wrapper class for the A* algorithm

```
from city import City
from comparator import Comparator
from heapq import heappush, heappop
from typing import Dict, List


class AStar:
    @classmethod
    def solve(self, start: City, end: City) -> List[str]:
        visited: Dict[City, List[str]] = {start: [start.name]}
        q: List[Comparator] = []
        heappush(q, Comparator(start, 0, end - start))

        while q:
            hc = heappop(q)
            city = hc.city
            start_cost = hc.start_cost
            path = visited[city]

            if city == end:
                return path

            for neighbour in city.neighbours:
                if neighbour not in visited and neighbour not in q:
                    heappush(q, Comparator(
                        neighbour, start_cost + 1, end - neighbour))
                    new_path = list.copy(path)
                    new_path.append(neighbour.name)
                    visited[neighbour] = new_path
```

ao_star.py: Wrapper class for AO* Algorithm

```
from typing import Dict, List
from city import City
import math


class AOStar:
    @classmethod
    def solve(self, cities: Dict[str, City], start: City, end: City):
        visited1: Dict[City, List[str]] = {start: [start.name]}
        visited2: Dict[City, List[str]] = {start: [start.name]}
        or_res = self._solve_or(visited1, start, end)

        cities_list = list(cities.keys())
        and_res = self._solve_and(
            visited2, cities[cities_list[0]], cities[cities_list[1]],
end)
        if len(and_res) == 0 or len(and_res) > len(or_res):
            return or_res
        return and_res

    @classmethod
    def _solve_or(self, visited: Dict[City, List[str]], start: City,
end: City) -> List[str]:
        if start == end:
            return visited[start]

        path = visited[start]
        res = []
        min_length = math.inf
        for neighbour in start.neighbours:
            if neighbour not in visited:
                new_path = list.copy(path)
                new_path.append(neighbour.name)
                visited[neighbour] = new_path
                temp = self._solve_or(visited, neighbour, end)
                if len(temp) < min_length and len(temp) != 0:
                    res = temp
                    min_length = len(res)

        return res

    @classmethod
```

```
    def _solve_and(self, visited: Dict[City, List[str]], start1: City,
start2: City, end: City) -> List[str]:
        first = self._solve_or(visited, start1, end)
        second = self._solve_or(visited, start2, end)

        if len(first) < len(second):
            return first
        return second
```

index.py: Main file to call all the above defined algorithms

```
from city import City
from utils import connect_city_random, print_answer
from a_star import AStar
from ao_star import AOStar

cities = {
    "a": City(1, 2, "a"),
    "b": City(1, 7, "b"),
    "c": City(2, 2, "c"),
    "d": City(3, 5, "d"),
    "e": City(7, 3, "e"),
    "f": City(9, 9, "f"),
}

connect_city_random(cities["a"], cities["b"])
connect_city_random(cities["a"], cities["d"])
connect_city_random(cities["b"], cities["c"])
connect_city_random(cities["a"], cities["c"])
connect_city_random(cities["d"], cities["c"])
connect_city_random(cities["d"], cities["e"])
connect_city_random(cities["f"], cities["e"])


for k in cities:
    print(f"{k}: {cities[k].print()}")

startCity = input("Enter starting city: ")
endCity = input("Enter ending city: ")

print_answer("A*", AStar.solve(cities[startCity], cities[endCity]))
print_answer("AO*", AOStar.solve(cities, cities[startCity],
cities[endCity]))
```

```
godcrampy@acer:~/code/college-notes/aiml-lab/lab-6$ python3 index.py
a: a{1 2} => (b{1 7}, 6) (d{3 5}, 10) (c{2 2}, 1)
b: b{1 7} => (a{1 2}, 6) (c{2 2}, 8)
c: c{2 2} => (b{1 7}, 8) (a{1 2}, 1) (d{3 5}, 8)
d: d{3 5} => (a{1 2}, 10) (c{2 2}, 8) (e{7 3}, 2)
e: e{7 3} => (d{3 5}, 2) (f{9 9}, 9)
f: f{9 9} => (e{7 3}, 9)
Enter starting city: a
Enter ending city: b
Solution for A*: ['a', 'b']
Solution for AO*: ['a', 'b']
godcrampy@acer:~/code/college-notes/aiml-lab/lab-6$
```

```
godcrampy@acer:~/code/college-notes/aiml-lab/lab-6$ python3 index.py
a: a{1 2} => (b{1 7}, 10) (d{3 5}, 7) (c{2 2}, 2)
b: b{1 7} => (a{1 2}, 10) (c{2 2}, 8)
c: c{2 2} => (b{1 7}, 8) (a{1 2}, 2) (d{3 5}, 6)
d: d{3 5} => (a{1 2}, 7) (c{2 2}, 6) (e{7 3}, 4)
e: e{7 3} => (d{3 5}, 4) (f{9 9}, 11)
f: f{9 9} => (e{7 3}, 11)
Enter starting city: a
Enter ending city: e
Solution for A*: ['a', 'd', 'e']
Solution for AO*: ['a', 'b', 'c', 'd', 'e']
godcrampy@acer:~/code/college-notes/aiml-lab/lab-6$
```

Note: AO* is not optimal and hence finds the longer path in some cases as compared to the A* algorithm

Comparison

A* algorithm :

1. It is used in pathfinding and graph traversal, also in the process of plotting an efficiently directed path between a number of points called nodes.
2. a* traverse the tree in depth and keep moving and adding up the total cost of reaching the cost from the current state to the goal state and add it to the cost of reaching the current state.

AO* algorithm :

1. AO* follow a similar procedure but there are constraints traversing specific paths.
2. When those paths are traversed, cost of all the paths which originate from the preceding node are added till that level, where you find the goal state regardless of the fact whether they take you to the goal state or not.