 # Artificial and Computational Intelligence Assignment 5
#Use Bi-directional search strategy for the given Assignment.
#Things to follow
#1. Use appropriate data structures to represent the graph and the path using python libraries
#2. Provide proper documentation
#3. Find the path and print it
#Coding begins here

#1. Define the agent environment in the following block
#PEAS environment, Initial data structures to define the graph and variable declarations

- PEAS Environment

Agent Type	Performance Measure	Environment	Actuators	Sensors
Ship Pilot	Safe, fast, comfortable trip, Shortest path, maximize profits	Sea, other traffic, Depth of the sea, Dockyard, citizen ,Weather Condition	Steering, accelerator, display	Camera, Weather and visibility Sensors. Compass, speedometer, odometer, accelerometer, engine sensors, keyboard

- We used **Queue** Data Structure to define the graph and variable declarations

#2. Define a formula that Checks for existence of path
#Function for checking for the path

```
def bidirectional_dijkstra(G, S, T)
```

Where,
G: Graph
S: Source node from where shortest path to be find
T: Target node till where shortest path to be find

We have used Dijkstra's algorithm to get the optimal path. It follows the greedy approach to determine the shortest path from a weighted graph, where the weight of each edge is non-negative

Bidirectional Dijkstra algorithm:
- Alternate between forward traversal (from source node) and backward traversal (from destination node)
- Calculate distance for the forward traversal (distance_f(v))
- Calculate distance for backward traversal (distance_b(v))
- Stop when forward_queue and backward_queue are empty
- Once traversal end fine the minimum value of distance_f(v)+ distance_b(v) to get the short path
- Combine both paths to get the final shortest path in the graph

Complexity :
Since we have implemented bi-directional search time complexity reduced to half. Since the search happened from both the source and destination simultaneously.

(O(2*(n/2)^2))

#3. Implementation of bi-directional search technique for finding the path

#Code block 1

```
import heapq as hq
import networkx as nx
import matplotlib.pyplot as plt
import math
import time
import random as random


"""Class: Queue"""
"""Description: Creating node for heap queue that will be used for running
efficient Dijkstra Algorithm"""

class Queue:
    def __init__(self, v, p):  #V is node and p is Priority in a heap tree
        self.v = v
        self.p = p

    def __lt__(self, other):
        return self.p < other.p


"""  Function : bidirectional_dijkstra(G,S,T)
     Parameters:
       G: Graph
       S: Source node from where shortest path to be find
       T: Target node till where shortest path to be find
     Description:
       - Heapq data structures has been used to implement Dijkstra
algorithm
       - <object>.heappop() and <object>.heappush() methods used to pop and
push vertices from a graph
     Stopping Criteria:
     1. dist_S[startS[0].v] + dist_T[startT[0].v] >= v_dist['weight']
     2. len(startS) + len(goal_S) < len(startT) + len(goal_T)
     3. when a node is scanned in both directions"""
#

def bidirectional_dijkstra(G, S, T):

    startS = [Queue(S, 0.0)]   # Creating initial start node for forward
search using HeapQ and setting its value to 0.0
    startT = [Queue(T, 0.0)]   # Creating initial start node for forward
search using HeapQ and setting its value to 0.0

    goal_S = set()
    goal_T = set()
    visit_node = {S,T}
    pre_S = dict()
    pre_T = dict()
    dist_S = dict()                                                  #
Dictionary to store distance from source to target
    dist_T = dict()                                                  #
Dictionary to store distance from target to source

    v_dist = {'weight': math.inf}                                          #
```

```
Setting other vertex initial distance to inf
    node = {'weight': None}


    pre_S[S] = None
    pre_T[T] = None
    dist_S[S] = 0.0
    dist_T[T] = 0.0
    def update(v, weight,goal):
        if v in goal:
            distance = dist_T[v] + weight
            if v_dist['weight'] > distance:
                v_dist['weight'] = distance
                node['weight'] = v

    while startS and startT:
        if dist_S[startS[0].v] + dist_T[startT[0].v] >= v_dist['weight']:
            return visit_node,reverse_traversal(node['weight'], pre_S,
pre_T)

        if len(startS) + len(goal_S) < len(startT) + len(goal_T):
            C = hq.heappop(startS).v                #Pop the smallest item
off the heap, maintaining the heap invariant.
            goal_S.add(C)
#C is current node
            for fwd in G[C]:
                if fwd in goal_S:
                    continue
                visit_node.add(C)
                visit_node.add(fwd)
                cur_dist = dist_S[C] + G[C][fwd]['weight']
                if fwd not in dist_S or cur_dist < dist_S[fwd]:
                    dist_S[fwd] = cur_dist
                    pre_S[fwd] = C
                    hq.heappush(startS, Queue(fwd, cur_dist))
                    update(fwd, cur_dist, goal_T)
        else:
            C = hq.heappop(startT).v                # Pop the smallest item
off the heap, maintaining the heap invariant
            goal_T.add(C)
            for back in G[C]:
                if back in goal_T:
                    continue
                visit_node.add(C)
                visit_node.add(back)
                cur_dist = dist_T[C] + G[back][C]['weight']
                if back not in dist_T or cur_dist < dist_T[back]:
                    dist_T[back] = cur_dist
                    pre_T[back] = C
                    hq.heappush(startT, Queue(back, cur_dist))
                    update(back, cur_dist, goal_S)

    return []

""" Function : traversal(T,pred)
    Description: Accept two argument i.e A Node and Pred (Predecessor) list
of visited node in a graph
    Function returns path of forward traversal"""
#
```

```python
def traversal(T,pred):
    path = []
    while T:
        path.append(T)
        T = pred[T]
    return path[::-1]


""" Function : traversal(v,pre_S,pre_T)
    Description: Accept three argument i.e A Node and two Pred
(Predecessor) list of visited node in a graph
              Function returns path of traversal for bi-directional
dijkstra algorithm, as it combine path traversal
              of forward and backward traversal"""


def reverse_traversal(v, pre_S, pre_T):
    path = traversal(v, pre_S)
    v = pre_T[v]
    while v:
        path.append(v)
        v = pre_T[v]
    return path
    #

""" Function : distance(G,path)
    Description: Function take a graph and path as argument and return
total distance for that particular path in a graph#
"""
#


def distance(G, path):
    dist = 0.0
    tot_v = len(path) -1  #Total Number of Vertex minus 1
    for i in range(tot_v):
        dist += G[path[i]][path[i + 1]]['weight']
    return dist



""" Function: generate_graph()
    Methods Used:
#
       generate_graph() for generating random graphs
#
       circular_layout(G): arranging node in circular way
#
       get_edge_attributes(G, 'weight'): getting weights of each edge
#
       draw_networkx_edge_labels(): Plotting weights of each edge on graph
#
"""
#

def generate_graph():
    G = nx.Graph()
    G.add_nodes_from(['A', 'B', 'C', 'D', 'E', 'F', 'G'])
    G.add_edge('A', 'B', weight=110)
    G.add_edge('A', 'C', weight=132)
    G.add_edge('B', 'D', weight=159)
```

```
    G.add_edge('B', 'G', weight=59)
    G.add_edge('C', 'G', weight=120)
    G.add_edge('C', 'E', weight=89)
    G.add_edge('G', 'E', weight=102)
    G.add_edge('G', 'F', weight=92)
    G.add_edge('G', 'D', weight=108)
    G.add_edge('D', 'F', weight=98)
    G.add_edge('F', 'E', weight=68)

    pos = nx.circular_layout(G)
    nx.draw_networkx(G, pos, node_size=700)
    labels = nx.get_edge_attributes(G, 'weight')
    nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
    plt.savefig('graph.png')
    plt.show(G)
    return G


""" Main
     Initial Inputs:
           - S: Source node from where traversal to begin
           - T: Target node till where shortest path to be find

     Methods Called:
           - generate_random_graph(): generates the random graph of n node
and e edges, which is then converted in dict
             format using nx.to_dict_of_dicts(G)
           - time.perf_counter(): is used for calculating runtime of
algorithm
           - dijkstra(G,S,T): calling single directional dijkstra algorithm
to find shortest path
           - bidirectional_dijkstra(G,S,T): calling bidirectional dijkstra
algorithm to find shortest path

     Output Variable:
           - bi_path: return shortest path for bidirectional dijkstra
algorithm
           - bi_dist: return shortest path distance for bidirectional
dijkstra algorithm
           - visited_node: List of node visited to find the shortest path
"""

if __name__ == "__main__":
    S = input("Please enter source city: ")
    T = input("Please enter destination city: ")
    print()
    G=generate_graph()
    G_to_dict = nx.to_dict_of_dicts(G)
    visited_node,bi_path = bidirectional_dijkstra(G_to_dict, S, T)
    bi_dist = distance(G_to_dict, bi_path)
    print("Bi Directional Dijkstra path: ", bi_path)
    print("Bi Directional Dijkstra cost: ", bi_dist)

    print("Bi Directional Search number of vertices travelled to cover the
path : ", visited_node)
```

#4. Calling main function
#Function call to the bi-directional search technique

```
if __name__ == "__main__":
    S = input("Please enter source city: ")
    T = input("Please enter destination city: ")
    G_to_dict = nx.to_dict_of_dicts(generate_graph())
    bi_path = bidirectional_dijkstra(G_to_dict, S, T)
    bi_dist = distance(G_to_dict, bi_path)
    print("Bi Directional Dijkstra path: ", bi_path)
    print("Bi Directional Dijkstra cost: ", bi_dist)
```

#5. The agent should provide the following output

#5.1. Whether a path exists
#Function to find the existence of path

```
def distance(G, path):
    dist = 0.0
    tot_v = len(path) -1  #Total Number of Vertex minus 1
    for i in range(tot_v):
        dist += G[path[i]][path[i + 1]]['weight']
    return dist
```

#5.2. The path that covers required vertices in the graph
#Function that prints the path covering required vertices using bi-directional search

```
bi_path = bidirectional_dijkstra(G_to_dict, S, T)
print("Bi Directional Dijkstra path: ", bi_path)
```

#5.3. Print the total number of vertices (areas) visited by the agent in finding the path

```
visited_node,bi_path = bidirectional_dijkstra(G_to_dict, S, T)
print("Bi Directional Search number of vertices travelled to cover the path: ", visited_node)
```

#Execute code to print the number of vertices travelled to cover the path. (using bi-directional search)

Main Program File : " Assignment_05_dijkstra_bidirectional.py "