**# 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""" #  
  
  
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 “