Artificial and Computational Intelligence Assignment 1

Double-click (or enter) to edit

Problem solving by Uninformed & Informed Search

List only the BITS (Name) of active contributors in this assignment:

1. Sughosh P Dixit - 2021fa04058
2. Pulkit Khandelwal - 2021fc04905
3. Pooja Jain - 2021fc04203
4. Akash Goel - 2021fc04277

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 environment in the following block

List the PEAS decription of the problem here in this markdown block

1. **Performance Measure**: Performance measure is the unit to define the success of an agent. Performance varies with agents based on their different precepts.The agent here is the search agent which performs the searching algorithms (DFS and RBFS) to find the best possible path in terms of time taken and ticket cost which must be optimized based on priority.
2. **Environment**: Environment is the surrounding of an agent at every instant. It keeps changing with time if the agent is set in motion. The type of environment that can be observed is deterministic and stochastic. It is able to determine the best possible path based on the given initial state and goal state. By statistical normalization the environment is stochastic which is able to consider parameters (Time Taken and Ticket Cost in this case) on equal grounds and compute.

3. **Actuator**: An actuator is a part of the agent that delivers the output of action to the environment. The actuator is in the form of goalTest method which decides whether to terminate the search operation or continue searching.
4. **Sensor**: Sensors are the receptive parts of an agent that takes in the input for the agent. Sensors are in this case the visited data structure which sense whether a given node of the graph is visited or not.

Design the agent as PSA Agent(Problem Solving Agent) Clear Initial data structures to define the graph and variable declarations is expected IMPORTANT: Write distinct code block as below

```
#Code Block : Set the matrix for transition & cost (as relevant for the given problem)
#importing required libraries
from sklearn.preprocessing import StandardScaler,MinMaxScaler
import pandas as pd
import bisect
import collections
import collections.abc
import heapq
import operator
import os.path
import random
import math
import functools
from itertools import chain, combinations

#Below is the dictionary which denotes the ticket cost of each node
GRAPH_Ticket = {
            'TU': {'JO': 20, 'SY': 15, 'IN': 40},
            'JO': {'SA': 16, 'SY': 12, 'IQ': 27, 'TU': 20},
            'SA': {'UAE': 32, 'IQ': 44, 'JO': 16},
            'UAE': {'SA': 32, 'KW': 50},
            'KW': {'UAE': 50, 'IQ': 30, 'IN': 60},
            'SY': {'TU': 15, 'JO': 12,'IQ': 27},
            'IQ': {'SY': 10, 'JO': 27,'KW': 30,'SA': 44},
            'IN': {'TU': 40, 'KW': 60}
        }

#Below is the dictionary whcih denotes the time taken to traverse each node
graph_time = {
            'TU': {'JO': 2, 'SY': 1, 'IN': 3},
            'JO': {'SA': 2, 'SY': 1, 'IQ': 2, 'TU': 2},
            'SA': {'UAE': 2, 'IQ': 3, 'JO': 2},
            'UAE': {'SA': 2, 'KW': 2},
            'KW': {'UAE': 2, 'IQ': 3, 'IN': 3},
            'SY': {'TU': 1, 'JO': 1,'IQ': 2},
            'IQ': {'SY': 2, 'JO': 3,'KW': 3,'SA': 3},
            'IN': {'TU': 3, 'KW': 3}
}

#Given the problem statement the priority of the time taken is double the priority of tick
#and have the cost function as 2*t+c which when minimized will be the optimal cost
data_ticket =pd.DataFrame(GRAPH_Ticket)
```

```
scaler = MinMaxScaler()
model = scaler.fit(data_ticket)
scaled_data_ticket = model.transform(data_ticket)

data_time =pd.DataFrame(graph_time)
scaler = MinMaxScaler()
model = scaler.fit(data_time)
scaled_data_time = model.transform(data_time)
#below heuristic matrix has vlaues computed based on the above defined cost function
heuristic_matrix = scaled_data_ticket + 2*(scaled_data_time)

#Graph with cost matrix with the normalized data of Ticket Cost and Time Taken
input_cost_matrix = {
     'TU': {'JO': 1.2, 'SY': 0.5, 'IN': 2.6},
     'JO': {'SA': 1.12, 'SY': 0.04, 'IQ': 1.54, 'TU': 1.2},
     'SA': {'UAE': 1.44, 'IQ': 2.68, 'JO': 1.12},
     'UAE': {'SA': 1.44, 'KW': 1.8},
     'KW': {'UAE': 1.8, 'IQ': 2.4, 'IN': 3},
     'SY': {'TU': 0.5, 'JO': 0.04,'IQ': 1.54},
     'IQ': {'SY': 1.54, 'JO': 1.54,'KW': 2.4,'SA': 2.68},
     'IN': {'TU': 2.6, 'KW': 3}
}
print(input_cost_matrix)

#Creation of Undirected Graph depicting the locations of all the countries
def UndirectedGraph(graph_dict=None):
    return Graph(graph_dict = graph_dict, directed=False)


input_cost_Undirected_graph = UndirectedGraph(
    {'TU': {'JO': 1.2, 'SY': 0.5, 'IN': 2.6},
     'JO': {'SA': 1.12, 'SY': 0.04, 'IQ': 1.54},
     'SA': {'UAE': 1.44, 'IQ': 2.68, },
     'UAE': {'KW': 1.8},
     'KW': {'IQ': 2.4, 'IN': 3},
     'SY': {'IQ': 1.54},
     'IQ': {'KW': 2.4}})

#Below is the locations of all the countris on a 2d Graph
input_cost_Undirected_graph.locations = dict(
    TU=(20, 20), IN=(40, 20), KW=(60, 15),
    SY=(50, 10), IQ=(30, 15), JO=(20, 10),
    UAE=(15, 7), SA=(25, 7))

     {'TU': {'JO': 1.2, 'SY': 0.5, 'IN': 2.6}, 'JO': {'SA': 1.12, 'SY': 0.04, 'IQ': 1.54,
```

2. Depth First Search

```
#Clase node
class Node_DFS:
```

```
    def __init__(self,state,parent=None,action=None,path_cost=0):
        self.state=state
        self.parent=parent
        self.action=action
        self.path_cost=path_cost

    def childNode(self,gp,action):
        childState=gp.result(self.state,action)
        path_cost_to_childNode = gp.pathCost(self.path_cost,self.state,action,childState)
        return Node_DFS(childState,self,action,path_cost_to_childNode)

    def expand(self,gp):
        return [self.childNode(gp,action) for action in gp.actions(self.state)]

#Travel Agent class
class TravelAgent:

    def __init__(self,initial,goal,graph):
        self.initial=initial
        self.goal=goal
        self.graph=graph

    def actions(self,state):
        return list(input_cost_matrix[state].keys())

    def result(self,state,action):
        return action
    #function to handle goal test (Must handle dynamic inputs).
    def goalTest(self,state):
        return state == self.goal

    def pathCost(self,cost_so_far, fromState,action,toState):
        return cost_so_far + input_cost_matrix[fromState][toState]

#Depth First Search implementation using stack approach
def DepthFirstSearch(gp,index, route):
    frontier=[]
    initialNode=Node_DFS(gp.initial)
    frontier.append(initialNode)
    explored=set()

    while frontier:
        print('Frontier: ', [node.state for node in frontier])
        if len(frontier) == 0 : return 'Failure' #If Initial state is the goal state

        node= frontier.pop(index)
        print('Pop : ', node.state)
        route.append(node.state)

        if gp.goalTest(node.state): return node #Checking if goal state is achieved

        explored.add(node.state)

        for child in node.expand(gp):
            print('Chld Node: ',child.state)
```

```
            if child.state not in explored and child not in frontier:
                frontier.append(child)

        print('==================================')

    return None
```

3. Recursive Best First Search

```
def is_in(elt, seq):
    """Similar to (elt in seq), but compares with 'is', not '=='."""
    return any(x is elt for x in seq)

def distance(a, b):
    """The distance between two (x, y) points."""
    xA, yA = a
    xB, yB = b
    return math.hypot((xA - xB), (yA - yB))

infinity = float('inf')

class Graph: # For undirected graphs only
    def __init__(self, graph_dict=None, directed=True):
        self.graph_dict = graph_dict or {}
        self.directed = directed

    def get(self, a, b=None):
        links = self.graph_dict.setdefault(a, {})
        if b is None:
            return links
        else:
            return links.get(b)

class Problem(object):
    def __init__(self, initial, goal=None):
       self.initial = initial
       self.goal = goal

    def actions(self, state):
         raise NotImplementedError

    def result(self, state, action):
        raise NotImplementedError
    #function to handle goal test (Must handle dynamic inputs).
    def goal_test(self, state):
        if isinstance(self.goal, list):
            return is_in(state, self.goal)
        else:
            return state == self.goal
```

```
    def path_cost(self, c, state1, action, state2):
        return c + 1

    def value(self, state):
        raise NotImplementedError

class GraphProblem(Problem):
    def __init__(self, initial, goal, graph):
        Problem.__init__(self, initial, goal)
        self.graph = graph

    def actions(self, A):
        return list(self.graph.get(A).keys())

    def result(self, state, action):
        return action

    def path_cost(self, cost_so_far, A, action, B):
        return cost_so_far + (self.graph.get(A, B) or infinity)

    def h(self, node):
        locs = getattr(self.graph, 'locations', None)
        if locs:
            if type(node) is str:
                return int(distance(locs[node], locs[self.goal]))
            return int(distance(locs[node.state], locs[self.goal])) ##this line works
        else:
            return infinity

class Node_RBFS:
    def __init__(self, state, parent=None, action=None, path_cost=0):
        self.state = state
        self.parent = parent
        self.action = action
        self.path_cost = path_cost
        self.f=0 #extra variable to represent total cost
        self.depth = 0
        if parent:
            self.depth = parent.depth + 1

    def __repr__(self):
        return "<Node {}>".format(self.state)

    def expand(self, problem):
        return [self.child_node(problem, action)
                for action in problem.actions(self.state)]

    def child_node(self, problem, action): # to make node object of each child
        next_state = problem.result(self.state, action)
        new_cost = problem.path_cost(self.path_cost, self.state,action, next_state)
        next_node = Node_RBFS(next_state, self, action,new_cost )
        return next_node

    def solution(self): # extracts the path of solution
        return [node.state for node in self.path()]
```

```
    def path(self): # extracts the path of any node starting from current to source
        node, path_back = self, []
        while node:
            path_back.append(node)
            node = node.parent
        return list(reversed(path_back)) # order changed to show from source to current

#Forming an undirected graph using a dictionary data structure
def UndirectedGraph(graph_dict=None):
    return Graph(graph_dict = graph_dict, directed=False)


#Expanding to child nodes
def mymax(childf,nodef, child,node):
    if childf>=nodef:
        print("node=", node.state, ", child=", child.state,
              ", node f=",nodef, " childf = ", childf, " assigning child's f" )
        return childf
    else:
        print("node=", node.state, ", child=", child.state,
              ", node f=",nodef, " childf = ", childf, " assigning node's  f <----" )
        return nodef

def RecursiveBFS(problem) :
    startnode = Node_RBFS(problem.initial)
    startnode.f = problem.h(problem.initial)
    return RBFS(problem, startnode,infinity)

#implementation of recursive best search algorithm
def RBFS(problem, node,f_limit) :
    print("\nIn RBFS Function with node ", node.state, " with node's f value = ", node.f ,
    if problem.goal_test(node.state) :
        return [node, None]
    successors = []
    for child in node.expand(problem):
        gval = child.path_cost
        hval = problem.h(child)
        child.f = mymax(gval+hval , node.f,child, node)
        successors.append(child)
    print("\n Got following successors for  ",node.state, ":", successors)
    if len(successors) == 0 :
        return [None, infinity]
    while True:
        best  = lowest_fvalue_node(successors)
        if best.f > f_limit :
            return [None, best.f]
        alternative = second_lowest_fvalue(successors, best.f)
        x = RBFS(problem, best, min(f_limit, alternative))
        result = x[0]
        print("updating f value of best node ", best.state, " from ", best.f , " to ", x[1
        best.f = x[1]
        if result != None :
            return [result, None]
```

```
#The fvalue here denotes the distance of each path with the other which are defined on the
#and the minimum cost path with start state and goal state are considered
def lowest_fvalue_node(nodelist):
    min_fval = nodelist[0].f
    min_fval_node_index=0
    for n in range(1,len(nodelist)):
        if nodelist[n].f < min_fval :
            min_fval_node_index = n
            min_fval = nodelist[n].f
    return nodelist[min_fval_node_index]
#Computing the second min value
def second_lowest_fvalue(nodelist,lowest_f):
    secondmin_fval = infinity
    for n in range(0,len(nodelist)):
        if nodelist[n].f > lowest_f and nodelist[n].f < secondmin_fval :
            secondmin_fval = nodelist[n].f
    return secondmin_fval
```

DYNAMIC INPUT

IMPORTANT : Dynamic Input must be got in this section. Display the possible states to choose from: This is applicable for all the relevent problems as mentioned in the question.

```
#Code Block : Function & call to get inputs (start/end state)
start_state = input('Enter the start state from the nodes given')
end_state = input('Enter the goal state from the nodes given')


decision=input('select which search you want to implement:\n1.DFS-Depth First Report\n2.RB
if(decision=='1'):
  gp=TravelAgent(start_state,end_state,input_cost_matrix)
  route=[]
  print ( " Result of Uninformed Search: DFS " )
  print('====================================')
  print("\n\nSolving for",start_state," to ",end_state,"....")
  node=DepthFirstSearch(gp,-1,route)
  print('====================================')
  print('Path:', (', '.join(route)))
  print('Cost of the travel: ', node.path_cost)
elif(decision=='2'):
  print ( " Result of Informed Search: RBFS " )
  print('====================================')
  print("\n\nSolving for",start_state," to ",end_state,"....")
  gp1 = GraphProblem(start_state, end_state, input_cost_Undirected_graph)
  resultnode = RecursiveBFS(gp1)
  if(resultnode[0] != None ):
      print('====================================')
      print("Path taken :" , resultnode[0].path())
      print("Path Cost :" , resultnode[0].path_cost)
else:
  print('Invalid Choice')
```

```
Enter the start state from the nodes givenJO
Enter the goal state from the nodes givenKW
select which search you want to implement:
1.DFS-Depth First Report
2.RBFS-Recursive Best First Search
2
 Result of Informed Search: RBFS
==================================


Solving for JO  to  KW ....

In RBFS Function with node  JO  with node's f value =  40  and f-limit =  inf
node= JO , child= SA , node f= 40  childf =  36.12  assigning node's  f <----
node= JO , child= SY , node f= 40  childf =  11.04  assigning node's  f <----
node= JO , child= IQ , node f= 40  childf =  31.54  assigning node's  f <----

 Got following successors for   JO : [<Node SA>, <Node SY>, <Node IQ>]

In RBFS Function with node  SA  with node's f value =  40  and f-limit =  inf
node= SA , child= UAE , node f= 40  childf =  47.56  assigning child's f
node= SA , child= IQ , node f= 40  childf =  33.8  assigning node's  f <----

 Got following successors for   SA : [<Node UAE>, <Node IQ>]

In RBFS Function with node  IQ  with node's f value =  40  and f-limit =  47.56
node= IQ , child= KW , node f= 40  childf =  6.2  assigning node's  f <----

 Got following successors for   IQ : [<Node KW>]

In RBFS Function with node  KW  with node's f value =  40  and f-limit =  47.56
updating f value of best node  KW  from  40  to  None
updating f value of best node  IQ  from  40  to  None
updating f value of best node  SA  from  40  to  None
==================================
Path taken : [<Node JO>, <Node SA>, <Node IQ>, <Node KW>]
Path Cost : 6.2
```

K### 4. Calling the search algorithms (For bidirectional search in below sections first part can be used as per Hint provided. Under second section other combinations as per Hint or your choice of 2 algorithms can be called .As an analyst suggest suitable approximation in the comparitive analysis section)

```
#Invoke algorithm 1 (Should Print the solution, path, cost etc., (As mentioned in the prob
start_state = input('Enter the start state from the nodes given')
end_state = input('Enter the goal state from the nodes given')

gp=TravelAgent(start_state,end_state,input_cost_matrix)
route=[]
print ( " Result of Uninformed Search: DFS " )
print('==================================')
node=DepthFirstSearch(gp,-1,route)
print('==================================')
print('Path taken :', (', '.join(route)))
print('Path Cost : ', node.path_cost)
```

```
Enter the start state from the nodes givenUAE
Enter the goal state from the nodes givenTU
 Result of Uninformed Search: DFS
==================================
Frontier:  ['UAE']
Pop :  UAE
Chld Node:  SA
Chld Node:  KW
==================================
Frontier:  ['SA', 'KW']
Pop :  KW
Chld Node:  UAE
Chld Node:  IQ
Chld Node:  IN
==================================
Frontier:  ['SA', 'IQ', 'IN']
Pop :  IN
Chld Node:  TU
Chld Node:  KW
==================================
Frontier:  ['SA', 'IQ', 'TU']
Pop :  TU
==================================
Path taken : UAE, KW, IN, TU
Path Cost :  7.4
```

```
#Invoke algorithm 2 (Should Print the solution, path, cost etc., (As mentioned in the prob
start_state = input('Enter the start state from the nodes given')
end_state = input('Enter the goal state from the nodes given')

print ( " Result of Informed Search: RBFS " )
print('==================================')
print("\n\nSolving for",start_state," to ",end_state,"....")
gp1 = GraphProblem(start_state, end_state, input_cost_Undirected_graph)
resultnode = RecursiveBFS(gp1)
if(resultnode[0] != None ):
    print('==================================')
    print("Path taken :" , resultnode[0].path())
    print("Path Cost :" , resultnode[0].path_cost)
```

```
 Enter the start state from the nodes givenJO
 Enter the goal state from the nodes givenKW
  Result of Informed Search: RBFS
 ==================================


 Solving for JO  to  KW ....

 In RBFS Function with node  JO  with node's f value =  40  and f-limit =  inf
 node= JO , child= SA , node f= 40  childf =  36.12  assigning node's  f <----
 node= JO , child= SY , node f= 40  childf =  11.04  assigning node's  f <----
 node= JO , child= IQ , node f= 40  childf =  31.54  assigning node's  f <----

  Got following successors for   JO : [<Node SA>, <Node SY>, <Node IQ>]

 In RBFS Function with node  SA  with node's f value =  40  and f-limit =  inf
 node= SA , child= UAE , node f= 40  childf =  47.56  assigning child's f
```

```
node= SA , child= IQ , node f= 40  childf =  33.8  assigning node's  f <----

 Got following successors for   SA : [<Node UAE>, <Node IQ>]

In RBFS Function with node  IQ  with node's f value =  40  and f-limit =  47.56
node= IQ , child= KW , node f= 40  childf =  6.2  assigning node's  f <----

 Got following successors for   IQ : [<Node KW>]

In RBFS Function with node  KW  with node's f value =  40  and f-limit =  47.56
updating f value of best node  KW  from  40  to  None
updating f value of best node  IQ  from  40  to  None
updating f value of best node  SA  from  40  to  None
==================================
Path taken : [<Node JO>, <Node SA>, <Node IQ>, <Node KW>]
Path Cost : 6.2
```

5. Comparitive Analysis

```
#Code Block : Print the Time & Space complexity of algorithm 1
import time
start_state = input('Enter the start state from the nodes given')
end_state = input('Enter the goal state from the nodes given')
times=[]
start_time = time.time()
gp=TravelAgent(start_state,end_state,input_cost_matrix)
route=[]
print ( " Result of Uninformed Search: DFS " )
print('==================================')
node=DepthFirstSearch(gp,-1,route)
print('==================================')
print('Path taken :', (', '.join(route)))
print('Path Cost : ', node.path_cost)
elapsed_time = time.time() - start_time
times.append(elapsed_time)
print('time taken:',times)
print('Worst Case Time Complexity of DFS is O(V+E)\n')
print('Space Complexity of DFS is O(V)\n')
print('Note: Here E denotes the no of edges and V denoted no of vertices\n')
```

```
Enter the start state from the nodes givenTU
Enter the goal state from the nodes givenKW
 Result of Uninformed Search: DFS
==================================
Frontier:  ['TU']
Pop :  TU
Chld Node:  JO
Chld Node:  SY
Chld Node:  IN
==================================
Frontier:  ['JO', 'SY', 'IN']
Pop :  IN
Chld Node:  TU
Chld Node:  KW
==================================
```

```
     Frontier:  ['JO', 'SY', 'KW']
     Pop :  KW
     ==================================
     Path taken : TU, IN, KW
     Path Cost :  5.6
     time taken: [0.005609035491943359]
     Worst Case Time Complexity of DFS is O(V+E)

     Space Complexity of DFS is O(V)

     Note: Here E denotes the no of edges and V denoted no of vertices



#Code Block : Print the Time & Space complexity of algorithm 2
import time
start_state = input('Enter the start state from the nodes given')
end_state = input('Enter the goal state from the nodes given')
times=[]
start_time = time.time()
print ( " Result of Informed Search: RBFS " )
print('==================================')
print("\n\nSolving for",start_state," to ",end_state,"....")
gp1 = GraphProblem(start_state, end_state, input_cost_Undirected_graph)
resultnode = RecursiveBFS(gp1)
if(resultnode[0] != None ):
    print('==================================')
    print("Path taken :" , resultnode[0].path())
    print("Path Cost :" , resultnode[0].path_cost)
elapsed_time = time.time() - start_time
times.append(elapsed_time)
print('time taken',times)
print('Worst Case Time Complexity of RBFS is O(nlogn)\n')
print('Space Complexity of RBFS is O(b m)\n')
print('Note: Here n is the no of nodes, b is the branching factor and m is the maximum dep

     Enter the start state from the nodes givenTU
     Enter the goal state from the nodes givenKW
      Result of Informed Search: RBFS
     ==================================


     Solving for TU  to  KW ....

     In RBFS Function with node  TU  with node's f value =  40  and f-limit =  inf
     node= TU , child= JO , node f= 40  childf =  41.2  assigning child's f
     node= TU , child= SY , node f= 40  childf =  11.5  assigning node's  f <----
     node= TU , child= IN , node f= 40  childf =  22.6  assigning node's  f <----

      Got following successors for   TU : [<Node JO>, <Node SY>, <Node IN>]

     In RBFS Function with node  SY  with node's f value =  40  and f-limit =  41.2
     node= SY , child= IQ , node f= 40  childf =  32.04  assigning node's  f <----

      Got following successors for   SY : [<Node IQ>]

     In RBFS Function with node  IQ  with node's f value =  40  and f-limit =  41.2
```

```
node= IQ , child= KW , node f= 40  childf =  4.4399999999999995  assigning node's  f

 Got following successors for   IQ : [<Node KW>]

In RBFS Function with node  KW  with node's f value =  40  and f-limit =  41.2
updating f value of best node  KW  from  40  to  None
updating f value of best node  IQ  from  40  to  None
updating f value of best node  SY  from  40  to  None
==================================
Path taken : [<Node TU>, <Node SY>, <Node IQ>, <Node KW>]
Path Cost : 4.4399999999999995
time taken [0.011774301528930664]
Worst Case Time Complexity of RBFS is O(nlogn)

Space Complexity of RBFS is O(b m)

Note: Here n is the no of nodes, b is the branching factor and m is the maximum depth
```

6. Provide your comparitive analysis or findings in no more than 3 lines in below section

Comparison : Depth first search which can be categorized as an Uniformed Search algorithm will always fetch the shortest path with a lower time complexity, but cannot ensure if the cost of the path is optimal, whereas Recursive Best First Search which is a Informed type of searching algorithm, although having a larger time complexity will ensure that it recursively compute and fetch the optimal path.___
