**Lab 6: Graphs and Shortest Path Algorithms**

**Task 1: Graphs**

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges. The various terms and functionalities associated with a graph is described in this lab practice. Following are the basic operations we perform on graphs.

* Create a graph
* Display graph vertices
* Display graph edges
* Add a vertex
* Add an edge

A graph can be easily presented using Python dictionaries. In fact, we are using the adjacency list representation discussed in the lecture slides. However, we represent the vertices as the keys of the dictionary for a quicker search, and all the connected vertices with a key as the corresponded value in the dictionary. For example, the key and its value like ‘a’:[‘b’,’c’] means the edges (a,b) and (a,c).

A graph is formally represented by a set of vertices (***V***), and a set of edges (***E***). Of course, we use lists rather than sets to represent ***V*** and ***E*** in our Python code. For example, a simple graph can be represented as follows, and your first task is to convert them into a graph dictionary.

V = [‘a’, ‘b’, ‘c’]

E = [(‘a’,’b’), (‘b’,’c’), (‘c’,’a’), (‘c’,’b’)]

Now, use the above V and E as the testing example to finish the initialisation of a Graph class.

class Graph:

def \_\_init\_\_(self, V=None, E=None):

self.gdict = {}

if V != None and E != None:

# add each from V as a new key with an empty list

for v in V:

self.gdict[v] = []

# update the dictionary by each pair from E

for sv, ev in E:

if sv in self.gdict:

self.gdict[sv].append(ev)

Then, if you use the following code to test, you should see the output like that

g = Graph([‘a’,’b’,’c’], [(‘a’, ‘b’), (‘b’, ‘c’), (‘c’, ‘a’), (‘c’,’b’)] )

print(g.gdict)

output: {'a': ['b'], 'b': ['c'], 'c': ['a', 'b']}

Note that by default we are considering directed graphs. For undirected graphs, you can add a new state parameter such as directed in the initialisation. For example,

def \_\_init\_\_(self, V=None, E=None, directed=True)

self.gdict = {}

self.directed = directed

if V != None and E != None:

# add each from V as a new key with an empty list

# update the dictionary by each pair from E

#if directed is True, add the edges once

# else add the edges twice.

If directed is false, you can add an edge to the graph dictionary twice by different orders in a pair. Now, modify your code based on the above pseudocode to tackle with undirected graphs and use the following code to test your program.

g = Graph([‘a’,’b’,’c’], [(‘a’, ‘b’), (‘b’, ‘c’), (‘c’, ‘a’)], False )

print(g.gdict)

output: {'a': ['b', ‘c’], 'b': [‘a’, 'c'], 'c': ['b',’a’]}

To display all vertices is easy as it simply displays all keys in the graph dictionary. You can add the following code into your Graph class.

def getVertices(self):

return list(self.gdict.keys())

To display all edges, you need to iterate all the items in the dictionary, and pair keys with the vertices in their values. Please finish the following code.

def getEdges(self):

edges = []

for key, value in self.gdict.items():

# pair key with each element in the value

# add each pair into edges

return edges

We also need another two functions to add extra vertices and edges. In fact, these functions have been included in the initialisation function already. Please finish the following two functions:

def addVertices(self, vertices):

def addEdges(self, edges):

where vertices and edges are two lists of vertices and edges respectively.

**Task 2: Weighted Graphs**

Edges may be weighted to show that there is a cost to go from one vertex to another. For example, in a road map, cities are connected by edges (or roads), the weight on the edge might represent the distance between the two cities or traffic status.

We still use our dictionary to store the weight for an edge. For example, 'a': [('b', 8), (‘c’, 1)] means the weight of (a, b) is 8, and the weight of (a, c) is 1. Of course, we have other approaches to represent edges and their weights.

Now you create a new class called GraphWeight based on the Graph class in Task 1 to make it compatible with weights. Also, you add a new function getEdgesWithWeights() to display all edges with their weights, and getEdges() is for the edges only.

You can also use the following simple example to test all the functions.

g = GraphWeight(["a","b","c"], [("a", "b", 1), ("b", "c", 2), ("c", "a", 3)])

**Task 3: Graph Search or Traversal**

Breadth-first or depth-first traversals for a graph are similar to the traversals of a tree. The major difference is that a graph may contain cycles, so we may come back to the same vertex again. Therefore, we may use a Boolean array or other data types to record the visited vertices.

Note that we are talking about the order of visiting all connected vertices, and it doesn’t mean that the adjacent vertices in the order must be connected. Finding a path of a graph is discussed in Task 4. Also, the algorithms introduced in the lecture are for all connected vertices, and it doesn’t directly work for the isolated vertices in undirected graphs or unreachable vertices in directed graphs. Certainly, the following algorithms can be enhanced to deal with searching all vertices in a graph.

For detailed demonstrations, you can visit the on-line tutorials, [DFS](https://www.tutorialspoint.com/data_structures_algorithms/depth_first_traversal.htm) and [BFS](https://www.tutorialspoint.com/data_structures_algorithms/breadth_first_traversal.htm). The main idea for DFS is to use a stack to store the visited vertices, and the vertex in the stack will be popped out if it has no unvisited adjacent vertices. However, BFS uses a queue to store all unvisited adjacent vertices. Please use the GraphWeight class and the following pseudocode to finish the functions DFS and BFS.

def depthFirstSearch(self, start, goal):

current = start

visited = []

toVisit = Stack()

toVisit.push(current)

while toVisit is not empty:

current = toVisit.pop()

if current == goal:

add the current into the visited list

break

else:

# each adjacent vertices of the current vertex

for v in [v for v, weight in self.gdict[current]]:

the **unvisited** vertex can be pushed into the stack

if current not be visited, add it into the visited list

return visited

def breadthFirstSearch(self, start, goal):

current = start

visited = []

toVisit = Queue()

toVisit.enqueue(current)

while toVisit is not empty:

current = toVisit.dequeue()

if current == goal:

add the current into the visited list

break

else:

# each adjacent vertices of the current vertex

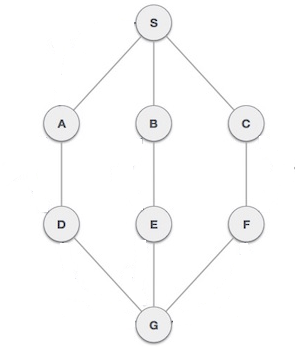
for v in [v for v, weight in self.gdict[current]]:

the **unvisited** vertex can be pushed into the stack

if current not be visited, add it into the visited list

return visited

Please use the following code to build a graph like the diagram, and test the output as shown.



g = GraphWeight(['s','a','b','c','d','e','f','g'],

[('s','a', 0), ('s','b', 0),('s','c', 0),

('a','d', 0),('b','e', 0),('c','f', 0),

('d','g', 0),('e','g', 0),('f','g', 0)],

False)

print(g.depthFirstSearch('s', 'a'))

output: ['s', 'c', 'f', 'g', 'e', 'b', 'd', 'a']

print(g.breadthFirstSearch('s', 'g'))

output: ['s', 'a', 'b', 'c', 'd', 'e', 'f', 'g']

**Task 4: Shortest Paths**

A path of a graph is a sequence of vertices in which any two consecutive vertices are adjacent and any vertex can be visited once. Here, we’ll not work on new algorithms for finding a path. Instead, we are going to use DFS and BFS to get a path. For example, the above output ['s', 'c', 'f', 'g', 'e', 'b', 'd', 'a'] isn’t a path because ‘b’ and ‘d’ are not adjacent. However, we can easily select a path from the output from DFS.

The idea is we start from the end of the list and delete non-adjacent vertices. For example, we start from ‘a’, and (‘d’, ‘a’) is an edge. We move to ‘d’, but there is no (‘b’, ‘d’), so ‘b’ should be deleted. Again, there is no (‘e’, ‘d’), so ‘e’ is removed. We apply this approach repeatedly until the first vertex in the list, and we can get a path.

Can you translate this idea into a function to find a path between any two vertices?

The shortest paths that we are talking about are not measured by the distance. In fact, we are searching a path with a minimal weight, and Dijkstra’s shortest path algorithm is a simple but efficient one. Dijkstra’s algorithm can be found in the lecture slide, and you should know how to work out the shortest path manually for a small graph. Based on the algorithm, there are many different implementations.

There are few notes you should pay attention to. The shortest path must have a start node, and it’s the shortest path from the start node to any other nodes, rather than between any nodes. You can also work out the complete shortest path tree no matter what the destination is, or just calculate the necessary shortest path based on the end node. The two different goals will determine how we terminate the program. Clearly, the complete shortest path tree will terminate only if all nodes have been visited already, the shortest path tree with an end node terminates as long as the end node has been visited.

The following program is for the shortest path tree with an end node.

def dijkstraSP(self, start, end):

#check wheter start and end are valid nodes

allnodes = list(self.gdict.keys())

if start not in allnodes or end not in allnodes:

return None

infinite = 100000

# build a table to record the total weight and its predecessor to get the weight

# also, this table records all unvisited nodes, visited nodes are removed from this table

table = {}

for node in allnodes:

table[node] = (node, infinite)

#record the sequence of visited nodes

edges = []

table[start] = (start, 0)

current = start

while current != end: #find the goal

# update total weight for all adjacent nodes of current

for v, w in self.gdict[current]:

if v in table: # not visited yet

# calculate node's total weight

totalweight = table[current][1] + w

if totalweight < table[v][1]:

#update weight and previous node

table[v] = (current, totalweight)

#add the visited edge into the sequence

#edges.append((table[current][0], current))

# delete current node from table to denote it's been visited

table.pop(current)

# get the unvisited nodes from the table

unvisited = list(table.items())

# terminate if all visited already

if len(unvisited) == 0:

return None

#sort the unvisited by its total weight

unvisited.sort(key = lambda x:x[1][1])

# pick up the first one or smalles one

current = unvisited[0][0]

# add the visited edge

edges.append((table[current][0], current, table[current][1]))

return edges

Note that the dictionary variable ‘table’ plays a key role in this program. E.g., table[2]=(1, 10) means from node 1 to node 2 can get the minimal weight 10. Also, visited nodes can removed from the table, and the empty means all nodes have been visited.

Also, this program returns a partial or complete shortest path tree only. You must work out further to calculate the shortest path from start to end and the related weight. Can you do it yourself?