Richard Hayes Crowley

07/25/2021

CSC\_242\_Lab\_015

**SOURCE CODE**

**Flight\_scheduler.py**

*import* db.directed\_graph\_db *as* db

*from* simple\_term\_menu *import* TerminalMenu

*from* Collections.LinkedDirectedGraph *import* LinkedDirectedGraph

airports = ["ATL", "NYC", "LA", "SF", "SEA", "CHI", "MIA", "DEN"]

def buildRoute(*g*: LinkedDirectedGraph):

fromVertex = airports[TerminalMenu(

airports, *title*="\nBuild route from...").show()]

toVertex = airports[TerminalMenu(

airports, *title*="\nBuild route to...").show()]

distance = int(

input(f"\nWhat is the distance from {fromVertex} to {toVertex}? Integers only: "))

edge = g.addEdge(fromVertex, toVertex, distance)

print(f"\nRoute added: {edge}")

def findShortestPaths(*g*: LinkedDirectedGraph):

fromVertex = airports[TerminalMenu(

airports, *title*="\nFind the shortest route from...").show()]

*return* g.shortestPaths(g.getVertex(fromVertex))

def main():

global conn

*# create a database connection*

conn = db.createConnection("./db/graphs.db")

*# create table*

*if* conn is not None:

*# create contacts table*

db.createTable(conn)

*else*:

print("Error! cannot create the database connection.")

*with* conn:

def readTable():

*return* db.readTable(conn)

def saveGraph(*graphLabel*, *graphData*):

*try*:

db.insertIntoTable(conn, [graphLabel, graphData])

*except*:

print("Something went wrong trying to save your graph to the database!")

def updateGraph(*graphLabel*, *graphData*, *graphId*):

*try*:

db.updateTable(conn, [graphLabel, graphData, graphId])

*except*:

print(

"Something went wrong trying to update your graph in the database!")

def deleteGraph(*graphId*):

*try*:

db.deleteFromTable(conn, graphId)

*except*:

print(

"Something went wrong trying to delete your graph from the database!")

def viewGraph(*id*, *name*, *routeGraph*):

print(f"\nViewing {name}")

*while* True:

entry = TerminalMenu(["View Flight Graph", "Build Routes",

"Find Shortest Paths", "Save Graph", "Exit"], *title*=f"\nWhat would you like to do with {name}?").show()

*if* entry == 0:

routeGraph.printGraph()

*elif* entry == 1:

buildRoute(routeGraph)

*elif* entry == 2:

shortestPaths = findShortestPaths(routeGraph)

print("\n")

print(shortestPaths["title"])

print(shortestPaths["table"])

*elif* entry == 3:

graphJson = routeGraph.saveJson()

updateGraph(name, graphJson, id)

*else*:

*break*

print("\nWelcome to the flight scheduler, a CLI for building graphs of flight routes and their shortest paths.\n")

currentGraph = LinkedDirectedGraph()

*while* True:

entry = TerminalMenu(["Build new flight graph", "Load flight graph", "Delete flight graph",

"Exit"], *title*="\nWhat would you like to do?").show()

*if* entry == 0:

name = input("Please enter a name for your graph: ")

newRouteGraph = LinkedDirectedGraph(

airports, *edgeLabel*="Flight miles")

graphJson = newRouteGraph.saveJson()

saveGraph(name, graphJson)

*elif* entry == 1:

table = readTable()

graphList = [i[1] *for* i *in* table]

selection = TerminalMenu(

graphList, *title*="\nWhich graph would you like to view?").show()

*for* i *in* table:

*if* i[1] == graphList[selection]:

currentGraph.loadJson(i[2])

viewGraph(*id*=i[0], *name*=i[1], *routeGraph*=currentGraph)

*elif* entry == 2:

table = readTable()

graphList = [i[1] *for* i *in* table]

selection = TerminalMenu(

graphList, *title*="\nWhich graph would you like to delete?").show()

*for* i *in* table:

*if* i[1] == graphList[selection]:

deleteGraph(i[0])

*else*:

print("goodbye!")

exit()

*if* \_\_name\_\_ == '\_\_main\_\_':

main()

**directed\_graph\_db.py**

*import* sqlite3

def createConnection(*dbFile*):

conn = None

*try*:

conn = sqlite3.connect(*dbFile*)

*except*:

print(

f"Something went wrong trying to connect to the db file {*dbFile*}")

*return* conn

def createTable(*conn*):

cur = *conn*.cursor()

cur.execute('''CREATE TABLE IF NOT EXISTS graphs

(id integer PRIMARY KEY, graphLabel text NOT NULL, graphData json)''')

def readTable(*conn*):

cur = *conn*.cursor()

*# table is an iterable of rows*

table = cur.execute('SELECT \* FROM graphs')

*return* [row *for* row *in* table]

def insertIntoTable(*conn*, *data*):

params = (*data*[0], *data*[1])

sql = '''INSERT INTO graphs(graphLabel,graphData)

VALUES (?,?)'''

cur = *conn*.cursor()

cur.execute(sql, params)

*conn*.commit()

print(f"{params[0]} added to DB")

def updateTable(*conn*, *data*):

''' params are label, json data, ID'''

params = (*data*[0], *data*[1], *data*[2])

sql = f''' UPDATE graphs

SET graphLabel = ? ,

graphData = ?

WHERE id = ?'''

cur = *conn*.cursor()

cur.execute(sql, params)

*conn*.commit()

print(f"{params[0]} updated in DB")

def deleteFromTable(*conn*, *graphId*: int):

*# remember the tuple gotcha, gotta have that comma!*

*# PS second argument in cur.execute must be tuple type*

params = (*graphId*,)

sql = ''' DELETE FROM graphs

WHERE id = ?'''

cur = *conn*.cursor()

cur.execute(sql, params)

*conn*.commit()

print(f"Entry with ID: {*graphId*} removed from DB")

*from* tabulate *import* tabulate

*from* dataclasses *import* dataclass

*import* jsons

*from* Collections.AbstractCollection *import* AbstractCollection

*from* Collections.LinkedVertex *import* LinkedVertex

*from* Collections.LinkedEdge *import* LinkedEdge

class LinkedDirectedGraph(AbstractCollection):

def \_\_init\_\_(*self*, *sourceCollection*=None, *edgeList*=None, *edgeLabel*=None) -> None:

*self*.edgeCount = 0

*self*.toEdges = []

*self*.vertices: dict(LinkedVertex) = dict()

*self*.edgeLabel = *edgeLabel*

*# init using abstract collection, will use linkedGraph.add moethod on each item in sourceCollection*

AbstractCollection.\_\_init\_\_(*self*, *sourceCollection*)

*if* *edgeList*:

*for* edge *in* *edgeList*:

*self*.addEdge(edge[0], edge[1], edge[2])

def \_\_str\_\_(*self*):

*return* f"{[ (f'Source Vertex: {v.getLabel()}',[f'{edge}' *for* edge *in* v.edgeList] )*for* v *in* *self*.vertices.values()]}"

def \_\_contains\_\_(*self*, *label*):

*return* *self*.containsVertex(*label*)

def clear(*self*):

*self*.\_\_init\_\_()

def loadJson(*self*, *jsonObj*):

'''

Loads properties from JSON string

'''

jsonDict = jsons.loads(*jsonObj*)

vertexList = jsonDict["vertexList"]

edgeList = jsonDict["edgeList"]

edgeLabel = jsonDict["edgeLabel"]

*self*.\_\_init\_\_(vertexList,

edgeList, edgeLabel)

*self*.edgeCount = jsonDict["edgeCount"]

*self*.toEdges = jsonDict["toEdges"]

def saveJson(*self*):

'''

Converts properties into JSON format and then returns the JSON string

'''

edgeList = []

vertexList = []

*for* v *in* *self*.vertices.values():

vLabel = v.getLabel()

vertexList.append(vLabel)

*for* e *in* v.getEdgeList():

otherVertex = e.getOtherVertex(v).getLabel()

distance = e.getWeight()

edgeList.append([vLabel, otherVertex, distance])

jsonDict = {

"edgeCount": *self*.edgeCount,

"toEdges": *self*.toEdges,

"edgeList": edgeList,

"vertexList": vertexList,

"edgeLabel": *self*.edgeLabel

}

results = jsons.dumps(jsonDict, *indent*=4)

*return* results

def clearEdgeMarks(*self*):

*for* vertex *in* *self*.vertices.values():

*for* e *in* vertex.edgeList:

e.clearMark()

def clearVertexMarks(*self*):

*for* vertex *in* *self*.vertices.values():

vertex.clearMark()

def isEmpty(*self*):

*return* len(*self*.vertices) == 0

def sizeEdges(*self*):

*return* *self*.edgeCount

def sizeVertices(*self*):

*return* *self*.size

*# Function to print adjacency list representation of a graph*

def printGraph(*self*):

*for* vertex *in* *self*.vertices.values():

print(vertex, *end*=": ")

*# print current vertex and all its neighboring vertices*

*for* neighborVertex *in* vertex.neighboringVertices():

edgeWeight = vertex.getEdgeTo(neighborVertex).getWeight()

print(

f"({vertex} —> {neighborVertex}, {f'{*self*.edgeLabel}: ' *if* *self*.edgeLabel *else* ''}{edgeWeight}) ", *end*="")

print()

def add(*self*, *label*: str):

*self*.addVertex(*label*)

def containsVertex(*self*, *label*) -> bool:

*return* *label* in *self*.vertices

def getVertex(*self*, *label*):

*if* *self*.containsVertex(*label*):

*return* *self*.vertices[*label*]

*raise* KeyError("That vertex doesn't exist!")

def addVertex(*self*, *label*):

''' adds a vertex with the given label to the graph'''

*self*.vertices[*label*] = LinkedVertex(*label*)

*self*.size += 1

def removeVertex(*self*, *label*: str) -> bool:

'''returns True if vertex was removed, false otherwise'''

removedVertex: LinkedVertex = *self*.vertices.pop(*label*, None)

*if* removedVertex is None:

*return* False

*# examine all other vertices to remove edges*

*# directed at the removed vertex*

*for* vertex *in* *self*.getVertices():

*if* vertex.removeEdgeTo(removedVertex):

*self*.edgeCount -= 1

*# examine al edges from the removed vertex to others*

*for* \_ *in* removedVertex.incidentEdges():

*self*.edgeCount -= 1

*self*.size -= 1

*return* True

def containsEdge(*self*, *edge*: LinkedEdge) -> bool:

*for* v *in* *self*.vertices.values():

*if* *edge* in v.edgeList:

*return* True

*return* False

def addEdge(*self*, *fromLabel*: str, *toLabel*: str, *weight*: int) -> None:

'''connects the vertices with and edge with the given weight'''

fromVertex = *self*.getVertex(*fromLabel*)

toVertex = *self*.getVertex(*toLabel*)

edge = fromVertex.addEdgeTo(toVertex, *weight*)

*self*.toEdges.append(*toLabel*)

*self*.edgeCount += 1

*return* edge

def getEdge(*self*, *fromLabel*: str, *toLabel*: str) -> LinkedEdge or None:

''' returns the edge containing the two vertices, or None if no edge exists'''

fromVertex = *self*.getVertex(*fromLabel*)

toVertex = *self*.getVertex(*toLabel*)

*return* fromVertex.getEdgeTo(toVertex)

def removeEdge(*self*, *fromLabel*: str, *toLabel*: str):

''' returns true if edge remove, or else false'''

fromVertex = *self*.getVertex(*fromLabel*)

toVertex = *self*.getVertex(*toLabel*)

edgeRemovedFlag = fromVertex.removeEdgeTo(toVertex)

*if* edgeRemovedFlag:

*self*.edgeCount -= 1

*self*.toEdges.remove(*toLabel*)

*return* edgeRemovedFlag

def edges(*self*):

edges = set()

*for* v *in* *self*.vertices.values():

*for* edge *in* v.edgeList:

edges.add(edge)

*return* iter(edges)

def getVertices(*self*):

*return* iter(*self*.vertices.values())

def incidentEdges(*self*, *label*):

*return* *self*.vertices[*label*].incidentEdges()

def neighboringVertices(*self*, *label*):

*return* *self*.vertices[*label*].neighboringVertices()

def shortestPaths(*self*, *sourceVertex*: LinkedVertex) -> list:

''' runs Djikstra's algorithm to find shortest path from source vertex to every vertex in graph'''

results = []

*# included object is a in process memo for marking whether a vertex's minimal distance from the source vertex has been included in the results grid yet or not*

included = {}

INFINITY = float("inf")

def addWithInfinity(*a*, *b*):

*if* *a* == INFINITY or *b* == INFINITY:

*return* INFINITY

*else*:

*return* *a* + *b*

def init(*results*, *included*):

*# for each vertex*

*for* idx, vertex *in* enumerate(*self*.vertices.values()):

*# get edge from source vertex to current vertex, if there is one*

edge: LinkedEdge = *sourceVertex*.getEdgeTo(vertex)

*# append row in results grid*

*# order of columns is: index, vertexLabel, distance from source vertex, parent vertex*

*results*.append([idx, vertex.getLabel(), 0, None])

*# source vertex gets 0 distance and no parent, this is the root of our tree*

*if* vertex == *sourceVertex*:

*results*[idx][2] = 0

*results*[idx][3] = None

*# mark as included*

*included*.update({vertex.getLabel(): True})

*# if there is an edge from source to current vertex, add distance to its row in the results grid, mark as unincluded, set parent to source vertex*

*elif* edge:

*results*[idx][2] = edge.getWeight()

*results*[idx][3] = *sourceVertex*.getLabel()

*# marking as unincluded, since we want to iterate through this vertex to see if we can use it to build a route*

*included*.update({vertex.getLabel(): False})

*else*:

*# else, no direct edge from source to current vertex, mark distance as INFINITY and parent to None*

*results*[idx][2] = INFINITY

*results*[idx][3] = None

*included*.update({vertex.getLabel(): False})

*return* {"results": *results*, "included": *included*}

def compute(*results*, *included*):

*# while not all vertices are included in the list*

*while* False in *included*.values():

*# iterate through our results rows*

*for* row *in* *results*:

label = row[1]

weight = row[2]

isIncluded: bool = *included*[label]

*# if it's not included in the list and has a weight, include it*

*if* not isIncluded and type(weight) is int:

*included*[label] = True

*else*:

*# else, iterate through the results again, find each other vertex that is not included*

*# and whether they have an edge to the current vertext*

*for* row2 *in* *results*:

label2 = row2[1]

weight2 = row2[2]

*if* label2 == label:

*continue*

isIncluded2: bool = *included*[label2]

fromVertex = *self*.vertices[label]

toVertex = *self*.vertices[label2]

*if* isIncluded2 == False:

edge: LinkedEdge = fromVertex.getEdgeTo(

toVertex)

*if* edge:

newDistance = addWithInfinity(

weight, edge.getWeight())

*# if distance not set and or new distance is less than the old distance*

*if* weight2 is INFINITY or newDistance < weight2:

*# set new distance in results grid*

row2[2] = newDistance

*# set parent label in results row*

row2[3] = label

*elif* label2 not in *self*.toEdges:

row2[2] = 0

*return* *results*

initialize = init(results, included)

computed = compute(initialize["results"], initialize["included"])

title = f"Set of shortest paths for origin {*sourceVertex*.getLabel()}"

table = tabulate(computed, *headers*=[

"Index", "Vertex", "Distance", "Parent"], *tablefmt*="github", *numalign*="left")

*return* {"title": title, "table": table, "computed": computed}

*# edgeList = [["C", "A", 3], ["A", "C", 4], ["B", "C", 2],*

*# ["A", "B", 5], ["C", "B", 8], ["C", "D", 4]]*

*# g = LinkedDirectedGraph(["A", "B", "C", "D", "E"],*

*# edgeList, edgeLabel="Distance")*

*# saveJson = g.saveJson()*

*# loadJson = g.loadJson(saveJson)*

**/\* LinkedVertex.py and LinkedEdge.py are the same as in my previous lab \*/**

**OUTPUT**

**Please see attached video!**