CS 2302 - Lab #6/Version A (Report)

December, 2018

Instructor Diego Aguirre

By: Alejandra Maciel (80631752)

Introduction

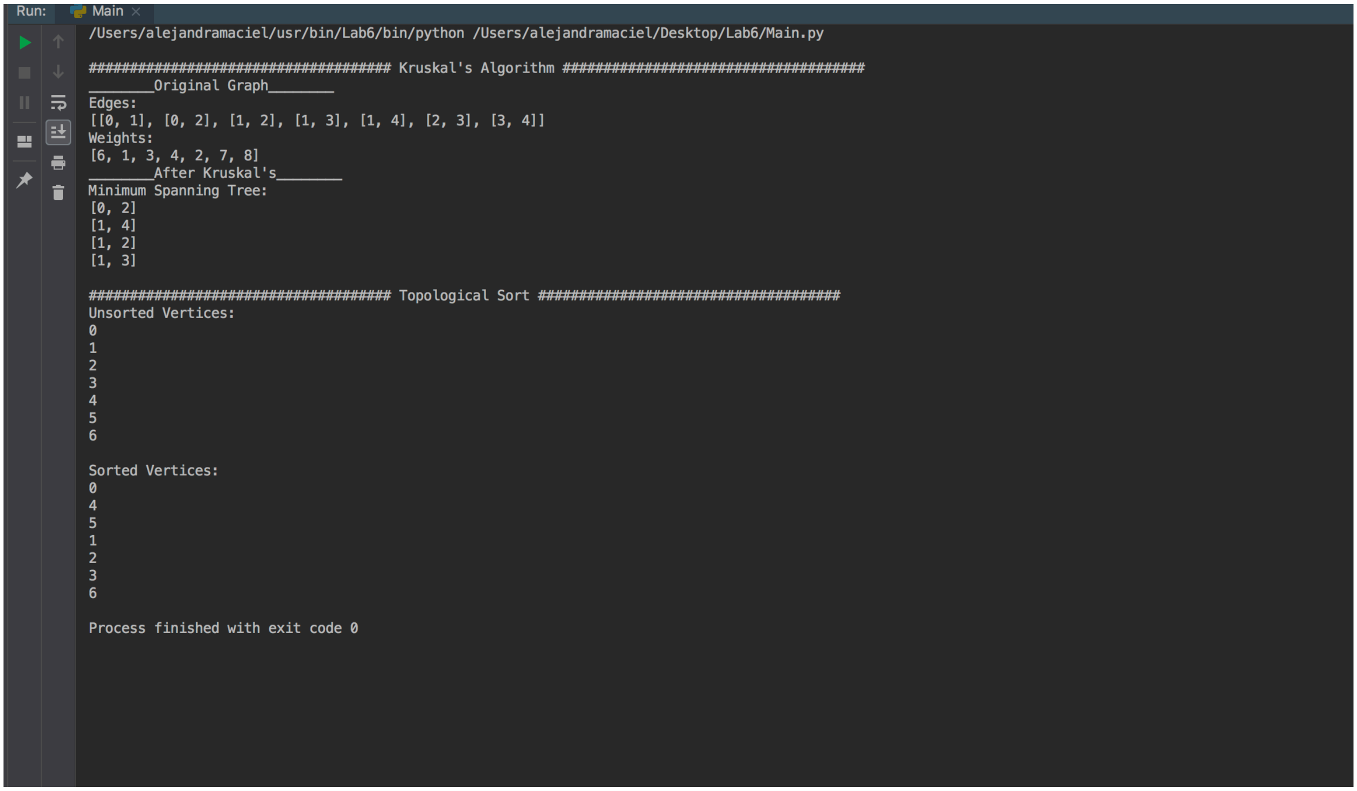
Lab 6 consisted on writing a code that would create a graph using adjacent list and implementing the Kruskal's method to find the minimum spanning tree possible and to sort the graph's vertices by implementing the Topological Sort.

Proposed solution, design, and implementation

To solve the assigned problem I decided to create three different classes; Graph AL, Graph and Queue. The Graph Al method is used to create the graph for the implementations. The Graph class contains all the methods needed to implement the Kruskals Algorithm to a graph. Finally, the Queue class contains all the actions needed to implement the topological sort. I also divided everything in 4 different files; Main, Adj_List_Graph, Kruskals, Topological. All of them are clearly explained by their name except by the Main file. The Main file imports the other 3 files to access their method in order to create the graphs and apply the algorithm and sorting method.

Experimental results

During the creation of the code there was several syntax and code errors, as well as several trial runs. But being that the professor had already given us the implementations during class it was not that hard. The most challenging part of the assignment was figuring out how to code the Kruskal's Algorithm, but after several research I understood how to make it work.



```
/Users/alejandramaciel/usr/bin/Lab6/bin/python /Users/alejandramaciel/Desktop/Lab6/Main.py

##################################### Kruskal's Algorithm #####################################
________Original Graph________
Edges:
[[0, 1], [0, 2], [1, 2], [1, 3], [1, 4], [2, 3], [3, 4]]
Weights:
[6, 1, 3, 4, 2, 7, 8]
________After Kruskal's________
Minimum Spanning Tree:
[0, 2]
[1, 4]
[1, 2]
[1, 3]

##################################### Topological Sort #####################################
Unsorted Vertices:
0
1
2
3
4
5
6

Sorted Vertices:
0
4
5
1
2
3
6

Process finished with exit code 0
```

Conclusions

Lab 6 as all the others was challenging but at the end I believe I got to understand how the Kruskal's Algorithm can be implemented as actual code. As well as how to implement the Topological sorting to a graph. Even though it was kind of confusing at first the program worked correctly at the end.

Appendix

TOPOLOGICAL FILE

```
#QUEUE CLASS
class Queue:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0, item)

    def dequeue(self):
        return self.items.pop()

    def size(self):
        return len(self.items)

#TOPOLOGICAL SORT METHOD
def topological(graph):
    #Get the number of vertices pointing to each vertex
    all_in_degrees = graph.compute_indegree_every_vertex()
    sort_result = []
    q = Queue()

    #Loop to find all the vertices with an indegree of 0 and enqueue them to the q list
    for i in range(len(all_in_degrees)):
        if all_in_degrees[i] == 0:
            q.enqueue(i)

    #While the q list is not empty dequeue each vertex and append it to the resultant list
    while not q.is_empty():
        u = q.dequeue()
        sort_result.append(u)

        #Loop to decrease by one the indegree of each vertex that the appended vertex point to
        for adj_vertex in graph.get_vertices_reachable_from(u):
            all_in_degrees[adj_vertex] -= 1
            #If the modified vertex's indegree becomes 0 then enqueue the vertex to the q list
            if all_in_degrees[adj_vertex] == 0:
                q.enqueue(adj_vertex)

    #If the length of the resultant list is different than the graph's number of vertices then return none
    if len(sort_result) != graph.get_num_vertices():
        return None

    #Else return the resultant list
    return sort_result
```

KRUSKALS FILE

```
class Graph:

    E = []
    W = []
    V = []

    def __init__(self, edge_list, weight):
        self.E.append(edge_list)
        self.W.append(weight)

    #Sort sets by weight
    def sort(self):
        if len(self.E) != len(self.W):
            return

        for i in range(1, len(self.W)):
            temp_weight = self.W[i]
            temp_edge = self.E[i]
            temp = i - 1

            while temp >= 0 and temp_weight < self.W[temp]:
                self.W[temp + 1] = self.W[temp]
                self.E[temp + 1] = self.E[temp]
                temp -= 1

            self.W[temp + 1] = temp_weight
            self.E[temp + 1] = temp_edge
```

```
#Create new sets
def build(self):
    for i in range(len(self.E)):
        for j in range(len(self.E[i])):
            if self.E[i][j] not in self.V:
                self.V.append(self.E[i][j])

    for k in range(len(self.V)):
        self.V[k] = [self.V[k]]

#Find sets
def find(self, vertex):
    for i in range(len(self.V)):
        for element in self.V[i]:
            if element == vertex:
                return i
    return None

#Combine sets
def union(self, vertex1, vertex2):
    index1 = self.find(vertex1)
    index2 = self.find(vertex2)
    for element in self.V[index2]:
        self.V[index1].append(element)
    self.V.pop(index2)

#Add edge with respective weight
def add(self, edge_list, weight):
    self.E.append(edge_list)
    self.W.append(weight)

#Kruskal's Algorithm
def kruskal(self):
    self.sort()
    self.build()
    count, i = 0, 0

    while len(self.V) > 1:

        if self.find(self.E[i][0]) != self.find(self.E[i][1]):
            print("[%d, %d]" % (self.E[i][0], self.E[i][1]))
            count += 1
            self.union(self.E[i][0], self.E[i][1])
            i += 1

    #Print Graph Method
    def print_graph(self):
        print("Edges:")
        print(self.E)
        print("Weights:")
        print(self.W)
```

ADJ_LIST_GRAPH FILE

```
#Node class for the Adjacent List Graph
class GraphALNode:
    def __init__(self, item, weight, next):
        self.item = item
        self.weight = weight
        self.next = next

#Adjacent List Graph Class
class GraphAL:

    def __init__(self, initial_num_vertices, is_directed):
        self.adj_list = [None] * initial_num_vertices
        self.is_directed = is_directed

    def is_valid_vertex(self, u):
        return 0 <= u < len(self.adj_list)

    def add_vertex(self):
        self.adj_list.append(None)

        return len(self.adj_list) - 1

    def add_edge(self, src, dest, weight = 1.0):
        if not self.is_valid_vertex(src) or not self.is_valid_vertex(dest):
            return

        self.adj_list[src] = GraphALNode(dest, weight, self.adj_list[src])

        if not self.is_directed:
            self.adj_list[dest] = GraphALNode(src, weight, self.adj_list[dest])


  def remove_edge(self, src, dest):
        self.__remove_directed_edge(src, dest)

        if not self.is_directed:
            self.__remove_directed_edge(dest, src)


    def __remove_directed_edge(self, src, dest):
        if not self.is_valid_vertex(src) or not self.is_valid_vertex(dest):
            return

        if self.adj_list[src] is None:
            return

        if self.adj_list[src].item == dest:
            self.adj_list[src] = self.adj_list[src].next
        else:
            prev = self.adj_list[src]
            cur = self.adj_list[src].next

            while cur is not None:
                if cur.item == dest:
                    prev.next = cur.next
                    return

                prev = prev.next
                cur = cur.next

        return len(self.adj_list)


    def get_num_vertices(self):
        return len(self.adj_list)


    def get_vertices_reachable_from(self, src):
        reachable_vertices = set()

        temp = self.adj_list[src]

        while temp is not None:
            reachable_vertices.add(temp.item)
            temp = temp.next
        return reachable_vertices


    def get_vertices_that_point_to(self, dest):
        vertices = set()

        for i in range(len(self.adj_list)):
            temp = self.adj_list[i]

            while temp is not None:
                if temp.item == dest:
                    vertices.add(i)
                    break

                temp = temp.next

        return vertices


    def get_vertex_in_degree(self, v):
        if not self.is_valid_vertex(v):
            return

        in_degree_count = 0

        for i in range(len(self.adj_list)):
            temp = self.adj_list[i]

            while temp is not None:
                if temp.item == v:
                    in_degree_count += 1
                    break

                temp = temp.next

        return in_degree_count


    def compute_indegree_every_vertex(self):

        all_indegrees = []
        for i in range(len(self.adj_list)):
            all_indegrees.append(self.get_vertex_in_degree(i))
        return all_indegrees
```

MAIN FILE

```
from Adj_List_Graph import GraphAL
from Topological import topological
from Kruskals import Graph

#Main Method
def main():

    print("\n######################################## Kruskal's Algorithm ########################################")
    print("________Original Graph________")
    #Test Graph
    graph = Graph([0, 1], 6)
    graph.add([0, 2], 1)
    graph.add([1, 2], 3)
    graph.add([1, 3], 4)
    graph.add([1, 4], 2)
    graph.add([2, 3], 7)
    graph.add([3, 4], 8)

    graph.print_graph()

    print("________After Kruskal's________")

    print("Minimum Spanning Tree: ")

    #Graph after applying Kruskal's Algorithm
    graph.kruskal()

    print("\n######################################## Topological Sort ########################################")

    #Test Graph
    graph2 = GraphAL(7, True)

    graph2.add_edge(0, 1)
    graph2.add_edge(0, 4)
    graph2.add_edge(1, 2)
    graph2.add_edge(2, 3)
    graph2.add_edge(3, 6)
    graph2.add_edge(4, 1)
    graph2.add_edge(4, 5)
    graph2.add_edge(5, 1)
    graph2.add_edge(5, 3)
    graph2.add_edge(5, 6)

    print("Unsorted Vertices:")
    for i in range(len(graph2.adj_list)):
        print(i)

    #Topological sorted vertices
    top_sort = topological(graph2)
    print("\nSorted Vertices: ")
    for j in top_sort:
        print(j)

main()
```

Academic honesty certification: ***"I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class."***

Student's Signature