Jose Escobar

80536060

CS2302 Lab 6 Report

Instructor: Diego Aguirre

**Introduction**

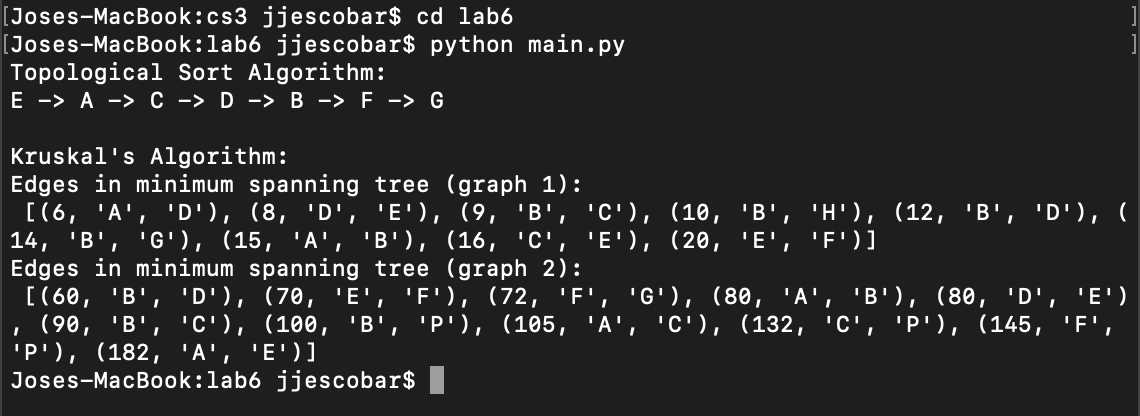
This lab required the implementation of Kruskal’s minimum spanning tree algorithm and Topological method to sort it. In order to make it possible, Disjoint Set Forest and Graph functions were also required.

**Proposed Solution**

For this lab, I thought that in order to make it more essay to read and effective only 2 files were needed, ‘Node’ and ‘main’ files. The ‘Node’ file contains the Graph and Disjoint Set Forest methods as well as Kruskal’s minimum spanning tree and Topological sort implementation with all their respective nodes. The ‘main’ file imports all methods from the ‘Node’ file and creates two graphs to test the minimum spanning and sort methods and their efficiency.

**Experimental Results**

Because we were able to use the instructor implementations and with the help of zybooks, the lab seemed as an easy one but it was a little more challenging than expected. The implementation of Kruskal’s and Topological methods took several reads to being able to successfully compile the results with the Graphs and DSF methods provided.



**Conclusion**

This lab helped me to have a better concept and understanding of Kruskal’s minimum spanning tree, as well as strength even more the knowledge of uses that Graph and Disjoint Set Forest data structures uses. Also, even though for Topological sort I already had a clearer idea of how it works, it helps a lot to see it working implemented in a lab.

**Appendix**

**NODE FILE:**

'''

Jose Escobar

UTEP ID 80536060

CS3 Lab 6: Kruskal's and Topological sort implementation

'''

class Vertex:

def \_\_init\_\_(self, label):

self.label = label

class Graph:

def \_\_init\_\_(self):

self.adjacency\_list = {}

self.edge\_weights = {}

def add\_vertex(self, new\_vertex):

self.adjacency\_list[new\_vertex] = []

def add\_directed\_edge(self, from\_vertex, to\_vertex, weight=1.0):

self.edge\_weights[(from\_vertex, to\_vertex)] = weight

self.adjacency\_list[from\_vertex].append(to\_vertex)

def add\_undirected\_edge(self, vertex\_a, vertex\_b, weight=1.0):

self.add\_directed\_edge(vertex\_a, vertex\_b, weight)

self.add\_directed\_edge(vertex\_b, vertex\_a, weight)

def get\_vertex(self, vertex\_label):

for vertex in self.adjacency\_list:

if vertex.label == vertex\_label:

return vertex

return None

def disjoint\_set(graph): #DSF creator

sets[graph] = graph

vertices[graph] = 0

def find(graph): #Find subsets on graph

if sets[graph] != graph:

sets[graph] = find(sets[graph])

return sets[graph]

def union(s1, s2): #Union of sets

ra = find(s1)

rb = find(s2)

if ra != rb:

if vertices[ra] > vertices[rb] or vertices[ra] < vertices[rb]:

sets[rb] = ra

if vertices[ra] == vertices[rb]: # If vertices are the same then increment both vertices

vertices[ra] += 1

vertices[rb] += 1

# The Topological sort function is from ZyBook

def get\_incoming\_edge\_count(edge\_list, vertex):

count = 0

for (from\_vertex, to\_vertex) in edge\_list:

if to\_vertex is vertex:

count += 1

return count

def topological\_sort(graph):

result\_list = []

e = []

for vertex in graph.adjacency\_list.keys():

if get\_incoming\_edge\_count(graph.edge\_weights.keys(), vertex) == 0:

e.append(vertex)

remaining\_edges = set(graph.edge\_weights.keys()) # starts with all edges

while len(e) is not 0:

curr\_vertex = e.pop() # select next vertex

result\_list.append(curr\_vertex)

outgoing\_edges = []

# remove current vertex outgoing edges from remaining edges

for to\_vertex in graph.adjacency\_list[curr\_vertex]:

outgoing\_edge = (curr\_vertex, to\_vertex)

if outgoing\_edge in remaining\_edges:

outgoing\_edges.append(outgoing\_edge)

remaining\_edges.remove(outgoing\_edge)

# check if removing outgoing edges creates new vertices with no incoming edges

for (from\_vertex, to\_vertex) in outgoing\_edges:

in\_count = get\_incoming\_edge\_count(remaining\_edges, to\_vertex)

if in\_count == 0:

e.append(to\_vertex)

return result\_list

sets = {} # Create an empty set dict

vertices = {} # Create an empty vertex dict

#Kruskal's minimum spanning tree algorithm

def kruskal(graph):

global dsf, tree\_edges

for i in graph['vertex']: #Minimum spanning tree on vertices

disjoint\_set(i)

tree\_edges = list(graph['edges'])

tree\_edges.sort()

dsf = set()

for j in tree\_edges: #Minimum spanning tree of the edges

w, vertex\_s1, vertex\_s2 = j # Weight, vertex\_1, vertex\_2

if find(vertex\_s1) != find(vertex\_s2):

union(vertex\_s1, vertex\_s2)

dsf.add(j)

return sorted(dsf)

**MAIN FILE**

'''

Jose Escobar

UTEP ID 80536060

CS3 Lab 6: Kruskal's and Topological sort implementation

'''

import Node

from Node import Graph, Vertex

g = Graph()

#Topological sort test

vertex\_A = Vertex('A')

vertex\_B = Vertex('B')

vertex\_C = Vertex('C')

vertex\_D = Vertex('D')

vertex\_E = Vertex('E')

vertex\_F = Vertex('F')

vertex\_G = Vertex('G')

g.add\_vertex(vertex\_A)

g.add\_vertex(vertex\_B)

g.add\_vertex(vertex\_C)

g.add\_vertex(vertex\_D)

g.add\_vertex(vertex\_E)

g.add\_vertex(vertex\_F)

g.add\_vertex(vertex\_G)

g.add\_directed\_edge(vertex\_A, vertex\_B)

g.add\_directed\_edge(vertex\_A, vertex\_C)

g.add\_directed\_edge(vertex\_B, vertex\_F)

g.add\_directed\_edge(vertex\_C, vertex\_D)

g.add\_directed\_edge(vertex\_D, vertex\_F)

g.add\_directed\_edge(vertex\_E, vertex\_F)

g.add\_directed\_edge(vertex\_E, vertex\_G)

g.add\_directed\_edge(vertex\_F, vertex\_G)

result\_list = Node.topological\_sort(g)

print("Topological Sort Algorithm: ")

first = True

for vertex in result\_list:

if first:

first = False

else:

print(' -> ', end='')

print(vertex.label, end='')

print ("\n")

#Kruskal's minimum spanning tree test

print("Kruskal's Algorithm: ")

graph2 = dict(vertex=['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H'],

edges={(15, 'A', 'B'),(6, 'A', 'D'),(9, 'B', 'C'),(12, 'B', 'D'),(14, 'B', 'G'),(10, 'B', 'H'),(16, 'C', 'E'),(8, 'D', 'E'),(20, 'E', 'F')})

graph3 = dict(vertex=['A', 'B', 'C', 'D', 'E', 'F', 'G', 'P'],

edges={(80, 'A', 'B'),(105, 'A', 'C'),(182, 'A', 'E'),(90, 'B', 'C'),(60, 'B', 'D'),(100, 'B', 'P'),(132, 'C', 'P'),(80, 'D', 'E'),(70, 'E', 'F'),(72, 'F', 'G'),(145, 'F', 'P'),(180, 'G', 'P')})

print("Edges in minimum spanning tree (graph 1): ", "\n", (Node.kruskal(graph2)))

print("Edges in minimum spanning tree (graph 2): ", "\n", (Node.kruskal(graph3)))

**Academic Honest 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.

Jose Escobar