Fernando Marquez

CS 2302 TTR 10:30- 12:00

Lab 6

Implementation of Kruskals Algorithm and Topological Sort

The objective of the lab was to implement kruskals algorithm and topological sort while checking for a cycle using disjoint set forest. Using kruskals algorithm were going to get the minimum spanning tree of the graph. For topological sort were going to sort vertices from a graph.

I was able to implement the data structures by first creating sets for disjoint set forest. After I created utility functions, find and union which would eventually be used for kruskals algorithm. For topological sort I got 2 functions from zybooks. The incoming edge count which checks to see if a vertex matches any other vertex in the graph. While the loop iterates through the conditional statements, the counter is incremented every time theres a vertex that matches another vertex. Next I implemented topological sort which would iterate though the conditional statements and return the result of the list. For kruskals algorithm I created 2 dictionary variables that would be used in the helper functions. I also created global statements to help with the outer scope variables being called. When the function is called it will iterate through kruskals function trying to find the minimum spanning tree in the vertices and edges. When it runs through the vertices conditionals it will sort the tree edges and weight according to minimum cost. For the edges conditional we will add an edge to the dsf according to the weight, the first vertex and the second vertex. Once its done iterating it will return the sorted graph. From zybooks I took the file, graph in order to add edges, vertexes and directed edges to the test file.

Experimental Results:

Topological sort

Minimal spanning tree Graph 1

Minimal spanning tree Graph 2

Result:

Topological Sort Algorithm from ZyBooks:

E -> A -> C -> D -> B -> F -> G

Kruskals Algorithm:

Minimal Spanning Tree of Graph 1 from ZyBooks:

[(6, 'A', 'D'), (8, 'D', 'E'), (9, 'B', 'C'), (10, 'B', 'H'), (12, 'B', 'D'), (14, 'B', 'G'), (15, 'A', 'B'), (16, 'C', 'E'), (20, 'E', 'F')]

Minimal Spanning Tree of Graph 2 from ZyBooks:

[(60, 'B', 'D'), (70, 'E', 'F'), (72, 'F', 'G'), (80, 'A', 'B'), (80, 'D', 'E'), (90, 'B', 'C'), (100, 'B', 'P'), (105, 'A', 'C'), (132, 'C', 'P'), (145, 'F', 'P'), (182, 'A', 'E')]

Time it took to run through program: 0.0003120899200439453

I learned how to implement kruskals and topological data structures. I also learned how to implement those data structures using disjoint set forest.

Appendix:

# Disjoint Set Forest taken from GraphAm

def disjoint\_set(graph): # Create sets for dsf

sets[graph] = graph

vertices[graph] = 0

def find(graph): # Util function to help find subsets of graph

if sets[graph] != graph:

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

return sets[graph]

def union(set1, set2): # Util function to combine both sets

ra = find(set1)

rb = find(set2)

if ra != rb: # Iterates through conditionals to combine sets unless both sets have repeated elements

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 # Return sorted topological list

sets = {} # Create an empty set dict

vertices = {} # Create an empty vertex dict

# Kruskals Algorithm Implementation

def kruskal(graph):

# Global statements to help with the outer scope variable calls

global dsf, tree\_edges

# Find the minimum spanning tree of the vertices

for i in graph['vertex']: # Vertex called in main

# Use Util functions to help find min spanning tree

disjoint\_set(i)

tree\_edges = list(graph['edges']) # Edges called in main

tree\_edges.sort() # sort edges

dsf = set()

# Find the minimum spanning tree of the edges

for j in tree\_edges:

w, vertex\_set1, vertex\_set2 = j # Weight, vertex\_1, vertex\_2

if find(vertex\_set1) != find(vertex\_set2):

union(vertex\_set1, vertex\_set2)

dsf.add(j) # Add edge to dsf

return sorted(dsf) # Return sorted graph

“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.”