Kenneth Schnall
Project: Graph Connectivity
https://github.com/kas/Graph-Connectivity

Introduction

In order to better understand graph theory and algorithm complexity analysis, I was tasked with developing an application that tests the connectivity of undirected and directed simple graphs. We must test the application with four graphs: directed connected, directed disconnected, undirected connected, and undirected disconnected. This project verified my understanding of analyzing time complexities.

Problem

In this project I had to find a way to test undirected and directed graphs for connectivity. I decided to use either a Breadth-First Search or a Depth-First Search.

I needed to perform a time complexity analysis on both searches to find the most efficient one. I found this to be the most difficult part of the project.

I chose to implement the Breadth-First Search. I decided to use Python 3 as I had been using it to develop some of my personal projects (https://kensch.com). I knew how to perform a Breadth-First Search with pen and paper using nodes and neighbors, but not programmatically. I figured that I would start by letting the user type in an adjacency matrix row by row. I found the package NumPy:

"NumPy is the fundamental package for scientific computing with Python."

NumPy will let us turn strings into matrices. For example, the string "0 1; 1 0" represents the adjacency matrix:

0	1
1	0

Next I wanted to find a framework that would let me easily work with nodes and neighbors. I found the framework NetworkX:

"NetworkX is a Python language software package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks."

With NetworkX we are able to import a graph from NumPy and draw the graph on the screen with a GUI. Of course this isn't necessary for our project, but it is neat to see the graph drawn on the screen directly from an adjacency matrix. NetworkX gives us the methods nodes() and neighbors(), which returns a list of all nodes on the graph and returns a list of neighbors of a node, respectively. These will be crucial for the algorithms we implement.

Possible Algorithms

- Depth-First Search

```
set graph object;
set visited to empty dictionary;

procedure depthFirst(vertex)
  set s to empty stack;

  for each currentVertex in graph do
    set visited[currentVertex] to false;

  push vertex onto s;

  while s is not empty do
    set u to pop s;

    if not visited[u] then
      set visited[u] to True;
        for each neighbour neighbor of u do
          if not visited[neighbor] then
            push neighbor onto s;
```

- Breadth-First Search

```
set graph object;
set visited to empty dictionary;

procedure breadthFirst(graph, vertex)
  set q to queue initialized with vertex;

  while q is not empty do
    set node to pop q;

    if visited[node] then
      continue;

    set visited[node] to True;

    for each neighbor neighbor of node do
      if not visited[neighbor] then
        append neighbor to q;
```

Time Complexity Analyses

Our Breadth-First Search algorithm has the time complexity of **O(n^2)**. This is because of two for loops and one while loop in the algorithm. Our first for loop, which iterates through each vertex to set them as not being visited, has the time complexity of **O(n)** because it visits each node once. Next we have a while loop which also loops through each node, having the

time complexity **O(n)**. Inside the while loop there is a for loop which in the worst case will run 1 less than the number of nodes, **O(n-1)**. Therefore, when we multiply those complexities together we get **O(n^2-n)**. We add the total while loop complexity to the for loop complexity from earlier and get a total algorithm time complexity of **O(n^2-n+n)**. This simplifies to **O(n^2)**.

Our Depth-First Search algorithm also has a for loop in the beginning which initializes each node as being not visited. This also has the time complexity **O(n)**. Later there is a while loop which also visits each node in the stack, having the time complexity **O(n)**. Inside this while loop there is a for loop which visits each neighbor of the selected node. The statement in this loop has the time complexity **O(n-1)**. We multiply the nested loops time complexities and add the earlier for loop time complexity to get the total time complexity of **O(n^2-n+n)**. Simplifying, we get a total algorithm time complexity of **O(n^2)**.

We can now see that the two algorithms we selected have the same time complexity of **O(n^2)**.

Code

The code for Graph Connectivity can be found at https://github.com/kas/Graph-Connectivity. Nonetheless we will include the Python code below:

```
import numpy
import networkx
# import matplotlib.pyplot as plt
from collections import deque



def breadthFirst(vertex):
        q = deque([vertex])

        for node in graph.nodes():
                visitedNodes[node] = False # o(n)

        while len(q) > 0: # o(n)
                node = q.pop()

                if visitedNodes[node]:
                        continue

                visitedNodes[node] = True

                for neighbor in graph.neighbors(node):
                        if (not visitedNodes[neighbor]):
                                q.appendleft(neighbor) # o(n^2-n)
```

```
def depthFirst(vertex):
        S = []

        for node in graph.nodes():
                visitedNodes[node] = False # o(n)

        S.append(vertex)

        while (S): # o(n)
                u = S.pop()

                if (not visitedNodes[u]):
                        visitedNodes[u] = True

                        for neighbor in graph.neighbors(u):
                                if (not visitedNodes[neighbor]):
                                        S.append(neighbor) # o(n^2-n)



isConnectedGraph = True
visitedNodes = {}



print("Enter each row of adjacency matrix using ; to denote a new row:")
userInput = input()

adjacencyMatrix = numpy.matrix(userInput)

print(adjacencyMatrix)
graph = networkx.from_numpy_matrix(adjacencyMatrix,
create_using=networkx.MultiDiGraph())
graph.edges(data=True)



# CODE TO DRAW GRAPH WITH MATPLOTLIB.PYPLOT
# pos = networkx.shell_layout(graph)
# networkx.draw(graph)
# plt.show()



# depthFirst(graph.nodes()[0])
```

```
breadthFirst(graph.nodes()[0])

for key, value in visitedNodes.items():
        if (value == False):
                isConnectedGraph = False;

print(isConnectedGraph)
```

I/O

I have included a file in the GitHub repository named graphinputs.txt that contains adjacency matrices and looks like this:

undirected connected
0 1 1 0 0 0;1 0 0 0 0 1;1 0 0 1 1 0;0 0 1 0 0 0;0 0 1 0 0 0;0 1 0 0 0 0

undirected disconnected
0 1 1 0 0 0;1 0 0 0 0 0;1 0 0 1 1 0;0 0 1 0 0 0;0 0 1 0 0 0;0 0 0 0 0 0

directed connected
0 1 0 0 0 0;0 0 1 0 0 0;0 0 0 0 0 1;1 0 0 0 0 0;0 0 0 1 0 0;0 0 0 0 1 0

directed disconnected
0 0 0 0 0 0;1 0 1 0 0 0;0 0 0 0 0 0;1 0 0 0 0 0;0 0 0 1 0 0;0 0 1 0 1 0

Each adjacency matrix is labeled by what type of graph it represents. I copied each adjacency matrix into my application and here is the output:

Undirected Connected Output:
[[0 1 1 0 0 0]
 [1 0 0 0 0 1]
 [1 0 0 1 1 0]
 [0 0 1 0 0 0]
 [0 0 1 0 0 0]
 [0 1 0 0 0 0]]
True

Undirected Disconnected Output:
[[0 1 1 0 0 0]
 [1 0 0 0 0 0]
 [1 0 0 1 1 0]
 [0 0 1 0 0 0]
 [0 0 1 0 0 0]
 [0 0 0 0 0 0]]

False

Directed Connected Output:

```
[[0 1 0 0 0 0]
 [0 0 1 0 0 0]
 [0 0 0 0 0 1]
 [1 0 0 0 0 0]
 [0 0 0 1 0 0]
 [0 0 0 0 1 0]]
```

True

Directed Disconnected Output:

```
[[0 0 0 0 0 0]
 [1 0 1 0 0 0]
 [0 0 0 0 0 0]
 [1 0 0 0 0 0]
 [0 0 0 1 0 0]
 [0 0 1 0 1 0]]
```

False