Daniel Oliveros A02093272
April 12, 2018

Assignment 6: Graph Algorithms I

Problem #1 Strong Graphs
The classes used for this algorithm are:

```
class Node:
    def __init__(self, val, color = "white"):
        self.val = val
        self.color = color

class Graph:
    def __init__(self, nodes, edges, inEdges = None):
        self.nodes = nodes
        self.edges = edges
        self.inEdges = self.makeIn()
    #builds adjacency lists with incoming edges based on the one made from
      ↪ outgoing edges
    def makeIn(self):
        IN = [[] for i in range (len(self.edges))]
        for index in range(len(self.edges)):
            for out in self.edges[index]:
                IN[out].append(index)
        return IN
```

(a) Paths from s to every v

(a) **Algorithm Description:** This algorithm is pretty simple. Performing a BFS from s takes us to every node s can reach. So, by performing a BFS and then checking if all nodes were reached we can determine if there is a path from s to every other node in the Graph.

(b) **Pseudocode:**

```
def BFSA(G, sI):
    s = G.nodes[sI]
    s.color = 'blue'
    Q = queue.Queue(len(G.nodes))
    Q.put(s)
    while(not Q.empty()):#O(n+m), all edges are visited once and all
        ↪ nodes are added once
        u = Q.get()
        for node in G.edges[u.val]:
            if(G.nodes[node].color == 'white'):
                G.nodes[node].color = 'blue'
                Q.put(G.nodes[node])
        u.color = 'red'
    works = True
    #if any nodes are not red by the end of BFS, then there isn't a
        ↪ path from s to all nodes
    for node in G.nodes:#O(n)
        if node.color != 'red':
            works = False
    return works
```

(c) **Correctness:** This algorithm is based on the idea that if there is a path from s to a node u, and that node has an edge going to another node v, then there must also be a path from s to v that goes through u, and so on for all nodes in the adjacency list of v. This is one of the basis of the pillars of Graph theory, and it's a reliable observation to be based off of.

It also relies on the observation that if a node was not reached by the algorithm, then its color cannot be red. This is because all nodes become red when they are processed by the queue, so no node that was added to it can be any color other than red by the time the loop ends.

(d) **Time Analysis:** This algorithm works the exact same way as BFS does. At most, it checks each node once, which has a runtime of $O(m)$, and it goes through each node once as well, which has a complexity of $O(n)$. The check at the end where I go through all the nodes and check if any of them are not red takes $O(n)$ operations. So, the total complexity of this algorithm is simply $O(n+m)$

(b) Paths from every v to s

(a) **Algorithm Description:** This algorithm follows the same approach as the previous one, except it uses a set of incoming edges instead of outgoing ones. This means we can reach every node with a path to s by performing this search, as we continuously look through the list of nodes that can reach the nodes that reach our node s

(b) **Pseudocode:**

```
def BFSB(G, sI):
    s = G.nodes[sI]
    s.color = 'blue'
    Q = queue.Queue(len(G.nodes))
    Q.put(s)
    while(not Q.empty()):#O(n+m) all incoming edges are visited once
        u = Q.get()
        for node in G.inEdges[u.val]:#use inEdges adjacency list
            if(G.nodes[node].color == 'white'):
                G.nodes[node].color = 'blue'
                Q.put(G.nodes[node])
        u.color = 'red'
    works = True
     #if any nodes are not red by the end of BFS, then there isn't a
        ↪ path from s to all nodes
    for node in G.nodes:#O(n)
        if node.color != 'red':
            works = False
    return works
```

(c) **Correctness:** This algorithm relies on similar ideas than the one for (a). The main difference is that it is based on the idea that if there's a path from a node u to s, and there is also a path from another node v to u, then there must be a path from v to s that goes through u. This observation is correct, and so it is a good basis to build an algorithm from.

(d) **Time Analysis:** This algorithm works the exact same way as the one in (a), the only difference is that it requires a list of incoming edges to be created based on the outgoing edges list. This process, however, also has a runtime of $O(n+m)$, since it traverses the list of outgoing ones in a manner that makes it so it only visits each edge once. So, the total runtime for this algorithm is also $O(n+m)$

(c) **Proof:** *G is strongly connected if and only if there exists a path in G from s to v and there is also a path from v to s for all vertices v of G.*
To prove this statement, we have to find a proof of how each statement entails the other one. So:

a. **G is strongly connected, therefore, there exists a path in G from s to v and there is also a path from v to s for all vertices v of G.**
Assuming G is strongly connected, then based on the definition of a strongly connected graph, for every pair of vertices u and v, there exists a path from u to v and vice versa. Since these paths exist for every pair of nodes, then we know for a fact it exists for s and every other node.

b. **There exists a path in G from s to v and there is also a path from v to s for all vertices v of G, therefore, G is strongly connected.**

For this proof, we need to explain how having a path to and from s and every other node entails that there's also a path from any of these nodes to one another. Since we have a path from any node v to s, and we have a path from s to every other node, then we know we are guaranteed to have a path from any of these nodes to one another through s, since we know we can get to s from any node, and we can get to any node from s.

Here is some testing code for this algorithm:

```
#create nodes
numNodes = 4
nodes = [Node(i) for i in range(numNodes)]
#create adjacency lists
edges = [None]*numNodes
edges[0] = [1,2]
edges[1] = [3]
edges[2] = [1,3]
edges[3] = [0]
#create Graph
G = Graph(nodes, edges)
i = 0 #index for starting node
isStrong = BFSA(G, i) and BFSB(G, i)
if(isStrong):
    print("G is stronk")
else:
    print("G is weak")
```

Problem #2 Number of shortest paths from s to t

1. **Algorithm Description:** For this algorithm, I augmented the nodes a little bit so they would store the number of paths that lead to them. This allows me to determine the number of paths to a node based on the nodes pointing at it, the counts stored within them, and the lengths of the paths from s to these nodes. Based on some criteria about the length of the path that reaches it, we can determine the number of paths to a node to be equal to the sum of paths that take us to the nodes that point to it.

2. **Pseudocode:**

```
import queue
class Node:
    def __init__(self, val):
        self.val = val
        self.pre = None
        self.d = 0
        self.color = 'white'
        self.count = 0
class Graph:
    def __init__(self, nodes, edges):
        self.nodes = nodes
        self.edges = edges
def problem2(G, sI, tI):
    #initialize distances to values greater than what's possible
    for node in G.nodes:
        node.d = len(G.nodes)+1
    #set up first node
    s = G.nodes[sI]
    s.d = 0
    s.count = 1
    s.color = 'blue'
    #set up queue
    Q = queue.Queue(len(G.nodes))
    Q.put(s)
    while(not Q.empty()):#O(n+m) go through all nodes and edges like in
       ↪ BFS
        u = Q.get()
        for node in G.edges[u.val]:
            #when a node is reached for the first time, initialize its
               ↪ count to be the count of the first node that visits it
            if(G.nodes[node].color == 'white'):
                G.nodes[node].d = u.d+1
                G.nodes[node].pre = u
                G.nodes[node].color = 'blue'
                G.nodes[node].count = u.count
                Q.put(G.nodes[node])
            #when a node is reached and it's still blue, check if the node
               ↪  that visited is closer to s than it and add its count
               ↪ if that's the case
            elif(G.nodes[node].color == 'blue' and u.d+1 == G.nodes[node].
               ↪ d):
                G.nodes[node].count += u.count
        u.color = 'red'
    return (G.nodes[tI].count)

#testing code
numNodes = 6
nodes = [Node(i) for i in range(numNodes)]
```

```
edges = [None]*numNodes
edges[0] = [1, 3]
edges[1] = [0, 2]
edges[2] = [1, 3, 5]
edges[3] = [0, 2, 4]
edges[4] = [3, 5]
edges[5] = [2, 4]

G = Graph(nodes, edges)
print(problem2(G, 0, 5))
```

3. **Correctness:** This algorithm relies in the idea that if we can get to a node from multiple nodes, and these other nodes are all at the same distance from s, which is equal to the distance from s to that node minus one, then the number of paths from s to that node is equal to the sum of the number of paths to each node that meets this criteria. This allows us to calculate the number of paths by traversing the graph with a similar approach to one used in a BFS, except we've modified it to allow for the necessary computations to take place.

4. **Time Analysis:** This algorithm traverses the graph in a manner similar to BFS, with some minor extra comparisons taking place in ways that don't increase the runtime. It has the same runtime as BFS, as it only touches each edge at most once and at most it accesses each node's list. Its runtime is therefore $O(n + m)$

Problem #3 Total number of paths from s to t

1. **Algorithm Description:** This algorithm tests all incoming nodes recursively in order to determine how many paths are coming from s to that given node. When a node has gone through this function, it adds its count to the count of all the nodes it paths to. Because of this, we use colors to tell apart the nodes that have been visited vs the ones that haven't. Those that have been visited increase the count of their adjacent nodes on their own, so there's no reason to check its count if it's been visited.

2. **Pseudocode:**

```
class Node:
    def __init__(self, val):
        self.val = val
        self.pre = None
        self.color = 'white'
        self.count = 0

class Graph:
    def __init__(self, nodes, edges):
        self.nodes = nodes
        self.edges = edges
        self.inEdges = self.makeIn()
    def makeIn(self):
        IN = [[] for i in range (len(self.edges))]
        for index in range(len(self.edges)):
            for out in self.edges[index]:
                IN[out].append(index)
        return IN

def problem3(G, sI, tI):
    t = G.nodes[tI]
    t.color = 'blue' #make color blue
    for nodeI in range (len(G.inEdges[tI])):#look at incoming edges
        if G.nodes[G.inEdges[tI][nodeI]] == G.nodes[sI]:
            t.count += 1
        #if a node has not been evaluated, evaluate it
        elif G.nodes[G.inEdges[tI][nodeI]].color == 'white':
            problem3(G, sI, G.inEdges[tI][nodeI])
    #once a node is evaluated, increase the count of all nodes it points
        ↪ to by its own count
    for node in G.edges[tI]:
        G.nodes[node].count += t.count
#testing code
numNodes = 4

nodes = [Node(i) for i in range(numNodes)]
edges = [None]*numNodes
edges[0] = [1, 2, 3]
edges[1] = [2]
edges[2] = [3]
edges[3] = []
sI, tI = 0, 3
G = Graph(nodes, edges)
problem3(G, sI, tI)

print(G.nodes[tI].count)
```

3. **Correctness:** This algorithm relies on similar principles as the one for 2 does. It can't take some things for granted that 1 does, and because of this it has to check for the nodes that reach a given

node instead of allowing the paths coming from s at their own pace.

4. **Time Analysis:** This algorithm only evaluates each node once, this is because when a node is evaluated its color is changed to blue, and a node is only evaluated when it's white. When nodes are evaluated, their incoming and outgoing adjacency lists are checked. Each of these list checks takes m operations, and going through all the nodes takes n. The runtime for this algorithm is, therefore, $O(n + m)$

Problem #4 Path with least weight and less than k hops

1. **Algorithm Description:** For this algorithm, we evaluate all paths coming from s that have a number of hops of at most k. Since we cannot make assumptions out of what the optimal path could be, we evaluate all of them and when t is reached compare the weights of the paths that reached it.

2. **Pseudocode:**

```
import queue
from collections import namedtuple


class Node:
    def __init__(self, val):
        self.val = val

class Graph:
    def __init__(self, nodes, edges):
        self.nodes = nodes
        self.edges = edges

def BFS(G, sI, tI, k):
    element = namedtuple('element', 'node, hops, weight')
    s = element(G.nodes[sI], 0, 0)
    Q = queue.Queue(len(G.nodes))
    Q.put(s)
    bestPath = None
    while(not Q.empty()):
        #go through all combinations of hops and weights as long as hops
           ↪ <= k
        u = Q.get()
        if(u.hops>k):
            # when a node is popped and hops>k, don't do anything with it
            continue
        if(u.hops <= k and u.node == G.nodes[tI]):
            #only care about checking weights when node t is reached
            if(bestPath==None):
                bestPath = [u.hops, u.weight]
            elif(bestPath[1] > u.weight):
                bestPath = [u.hops, u.weight]
        #add all weighted from our node to the queue
        for edge in G.edges[u.node.val]:
            toAdd = element(G.nodes[edge[0]], u.hops+1, u.weight + edge
                ↪ [1])
            Q.put(toAdd)

    return bestPath

#testing code
numNodes = 5
maxHops = 3

nodes = [Node(i) for i in range(numNodes)]

edges = []
for i in range(numNodes):
    edges.append([])

edges[0] = [(1,1),(3,4),(4,5)]
```

```
edges[1] = [(2,1)]
edges[2] = [(4,1)]
edges[3] = []

sI, tI = 0, 4
G = Graph(nodes, edges)

print(BFS(G, sI, tI, maxHops))
```

3. **Correctness:** How optimal a path is cannot be determined only based on the weight of it and the number of hops that it took to get there. We, therefore, cannot ignore paths based on either of these values, and instead have to evaluate all paths available as long as their number of hops is less than k

4. **Time Analysis:** The number of paths created when going through this process is at most $k(n+m)$, this is because the maximum number of paths of length k found in it is $n+m$. The total runtime for this algorithm is, therefore, $O(k(n+m))$

People I worked with for this assignment:

- Andrew Aposhion
- Chris Darais
- Erik Davtyan