1. Given a directed graph G of n vertices and m edges, let s be a vertex of G.

a. Design an O(m+n) time algorithm to determine whether the following is true: there exists a path from s to v in G for every vertex v of G. (10 points)

- Algorithm Description

run a BFS algorithm on the graph. At the end find if a node wasnt found return false otherwise return true.

- Psudocode

// G = adjacency list

// s = start vertex

// n = number of vertices

// m = number of edges

def is\_path\_from\_s\_to\_all\_nodes(G, s, n, m) {

color = []

distance = []

prev = []

for v in G {

color[v] = "white"

distance[v] = null

prev[v] = null

}

queue = Queue()

queue.enqueue(s)

color[s] = "blue"

distance[s] = 0

while not queue.isEmpty() {

u = queue.dequeue()

for v in u {

if color[v] == "white" {

color[v] = "blue"

queue.enqueue(v)

prev[v] = u

distance[v] = distance[u] + 1

}

}

}

for i in distance { if i == null return false }

return true

}

- Correctness

The problem is almost asking to directly run a BFS algorithm. The algorithm written is an adaptation of the one written in class.

- Time Analysis

The time complexity would be the same as a BFS algorithm thus O(n+m)

b. Design an O(m+n) time algorithm to determine whether the following is true: there exists a path from v to s in G for every vertex v of G. (10 points)

- Algorithm Description

I have adapted the algorithm given in part a for this problem. Now I start at every node and try to find a path to s. If a node is in a path to get to s then I mark it as found.

- Psudocode

// G = adjacency list

// s = start vertex

// n = number of vertices

// m = number of edges

def is\_path\_from\_all\_nodes\_to\_s(G, s, n, m) {

found = []

color = []

distance = []

prev = []

for v in G {

color[v] = "white"

distance[v] = null

prev[v] = null

found[v] = false

}

for w in G {

if not found[w] {

queue = Queue()

queue.enqueue(w)

color[w] = "blue"

distance[w] = 0

while not queue.isEmpty() {

u = queue.dequeue()

if u == s {

mark all the nodes in the path as found in the found list

found[w] = true

break

}

for v in u {

if color[v] == "white" {

color[v] = "blue"

queue.enqueue(v)

prev[v] = u

distance[v] = distance[u] + 1

}

}

}

}

}

for f in found { if not f { return false } }

return true

}

- Correctness

I have run through examples on my whiteboard and it works.

- Time Analysis

I ask if a node has already been found to connect to s and skip doing unnecessary work if that is the case.Thus the time complexity would be O(n+m) because I travel on all the edges and nodes.

Notes:

- The input is the adjacency list for G. This means that all information needed in your algorithm must be computed from the adjacency lists of G. For example if you want to convert G to a new graph G-prime, then you must compute the adjacency lists of G-prime using the adjacency lists of G.

- Here is an application of your algorithm for (a) and (b). We say that a directed graph G is strongly connected if for every pair of vertices u and v, there exists a path from u to v and there also exists a path from v to u in G. An interesting observation is that G is strongly connected if and only if there exists a ptah in G from s to v and there is also a path from v to s for every vertex v of G (you may think about how to prove this observation). In light of the observation, we can determine whether G is strongly connected in O(m+n) time by using your algorithms for the above two questions (a) and (b).

1. Given a directed-acyclic-graph (DAG) G of n vertices and m edges, let s and t be two vertices of G. there might be multiple different paths (not necessarily shortest paths) from s to t (e.g., see Fig. 1 for an example). Design an O(m + n) time algorithm to compute the number of different paths in G from s to t. (20 points)

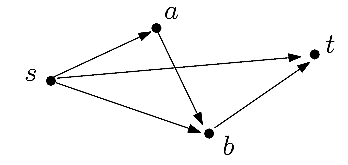


Figure 1: There are three different paths from s to t: s -> t, s -> b -> t, and s -> a -> b -> t.

- Algorithm Description

A recursive algorithm will be used to traverse the graph and find all the paths. once a paths end is found then the head of that path will be popped off and another path will be looked for. This will continue until all the paths are found.

- Psudocode

// s = start node

// t = end node

// n = number of nodes

// m = number of edges

def getNumPaths(s, t, n, m) {

visited = [false] \* n

paths = []

getPaths(s, t, visited, [], paths)

return len(paths)

}

def getPaths(v, t, visited, path, paths) {

visited[v] = true

path.add(v)

if v == t {

paths.add(path)

}

else {

for u in v {

if not visited[u] {

getPaths(u, t, visited, path, paths)

}

}

}

path.pop()

visited[v] = false

}

- Correctness

I have ran through examples on my whiteboard for a while and the algorithm works.

- Time Analysis

All edges and nodes will be visited thus the time complexity is O(n+m)

1. Given a directed graph G of n vertices and m edges, each edge (u, v) has a weight w(u, v), which can be positive, zero, or negative. The bottleneck-weight of any path in G is defined to be the largest weight of all edges in the path. Let s and t be two vertices of G. A minimum bottleneck-weight path from s to t is a path with the smallest bottleneck-weight among all paths from s to t in G. Refer to Figure 2 for an example. Modify Dikstra's algorithm to compute minimum bottleneck-weight paths from s to all other vertices of G. Your algorithm does not have to output all paths but only need to compute the correct predecessor information for all vertices (which forms a minimum bottleneck-weight path tree with s as the root), as we did in class for Dijkstra's algorithm. Your algorithm is required to have the same time complexity as Dijkstra's algorithm, i.e., O((n+m)log(n)) time. (20 points)

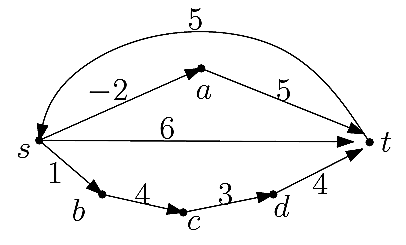


Figure 2: Here is a minimum bottleneck-weight path from s to t: s, b, c, d, t, whose bottleneck-weight is 4.

- Algorithm Description

4 lists will be maintained. The distance from s to all other nodes called d, the predecessor of the node called prev, and a Boolean list where a true means the node has already been visited, this is to prevent cycles, called visited, max\_path\_values that will hold the max value for a path once t is found.

Dijkstra's algorithm is augmented and ran with s as the starting node and t as the node to find. While the algorithm is running the 4 lists are filled with the required information.

At the end the max\_path\_values list can be examined to see what the minimum bottleneck-weight is.

- Psudocode

def dijkstra(G, s, t) {

d = []

prev = []

max\_path\_values = []

visited = []

for v in G {

d[v] = null

prev[v] = null

visited = false

}

d[s] = 0

heap = minHeap() // using d[v] as keys

while not heap.isEmpty() {

u = heap.pop()

if not visited[u] {

visited[u] = true

for v in u {

if v == t {

find the largest value in the path using the prev list and store the result into max\_path\_values

}

if d[v] > d[u] + w(u,v) {

d[v] = d[u] + w(u, v)

prev[v] = u

heap.decreseKey(v, d[u])

heap.heapify()

}

}

}

}

min = null

for m in max\_path\_values {

if min > m { min = m }

}

return min

}

- Correctness

I have ran through many examples on my whiteboard and the algorithm works with any graph I can think of.

- Time Analysis

Because the algorithm is an adaptation of Djistras algorithm the time complexity would be the same. In my algorithm only extra data and if statements have been added thus there is no change to the time complexity O((n+m)log(n)).

1. Let G be an undirected connected graph of n vertices and m edges. Suppose each edge of G has a color of either blue or red. Design an algorithm to find a spanning tree T of G such that T has as few red edges as possible. Your algorithm should run in O((n+m)log(n)). (10 points)

- Algorithm Description

two min priority queues are used. One to keep track of the blue edges and the other the red. Only when the blue queue is empty is an edge from the red queue used. The mst is filled out with edges and nodes until there are no more nodes left. Prims algorithm is augmented with this idea.

- Psudocode

// n = number of vertices

// m = number of edges

// G = adjacency list of weighted edges

// s = start node index

Example of using G

for v in G:

for u in v:

u.color // the color from v to u

u.weight // the weight from v to u

End example

def get\_mst\_least\_red(G, n, m, s = 0) {

visited = [false] \* n

mst = Tree // stores the nodes and edges (v, u)

blue\_pq = min priority queue // priority on u.weight

red\_pq = min priority queue // priority on u.weight

v = G[s] // get the starting node

visited[v] = true

for u in v { // visit v's neighbors

if u.color == "red" { red\_pq.add((v, u)) }

else { blue\_pq.add((v, u)) }

}

while (not blue\_pq.isEmpty() or not red\_pq.isEmpty) {

// pull from blue if there is data and red if blue is empty

v, u = blue\_pq.pop() or red\_pq.pop()

if vistited[u] { continue }

mst.add((v, u))

vistied[u] = true

for w in u {

if w.color == "red" { red\_pq.add((u, w)) }

else { blue\_pq.add((u, w)) }

}

}

if mst.size == n { return mst }

else {return null}

}

- Correctness

I have ran through many examples on my whiteboard and everything seems in-order

- Time Analysis

The time would be the same as Prims algorithm since Prims algorithm is at the heart of the algorithm. Only extra stored data and checks are added. O((n+m)log(n))