**Aleks Itskovich**

**Assignment #3**

**Question 1**

**a)**

PrintCC(G)

*# initialize the graph*

for each v in G.V

v.visited = false

v.parent = NIL

*# initialize global variable to count components*

numComponents = 0

*# loop through each vertex, carrying out* *DFS whenever we encounter an unvisited node*

for each v in V

if v.visited = false

numComponents += 1

print “Component ” + numComponents + “:”

DFS-visit-with-print (v)

print “Total number of connected components: ” + numComponents

DFS-visit-with-print(u)

u.visited = true

print (u)

for each v in Adj[u]

if v.visited = false

v.parent = u

DFS-visit(u)

**b)**

**Find-Cycles-BFS(G)**

Q = new Queue()

V = G.V *# get all vertices in the graph*

*# loop over all vertices to handle unconnected graphs*

for each v in V

if v.visited = false

*# begin BFS*

v.visited = true

ENQUEUE(Q, v)

while Q is not empty

u = DEQUEUE(Q)

for t in Adj[u]

if (t.visited = true and t != u.parent)

*# CYCLE FOUND*

return true

else if (t.visited = false)

t.visited = true

t.parent = u

ENQUEUE(t)

*# NO CYCLES FOUND*

return false

**Runtime Justification**

This algorithm is just a modification of BFS.

We have an outer loop that searches through all the indices. That has complexity O(V). The outer loop might perform constant work (if v is visited), or carry out BFS on a subset of vertices and edges. Overall, however, BFS is carried out over all V vertices and E edges with no overlap. Overall, each vertex is placed in the queue exactly once and the for loop over the edges is executed exactly twice.

Therefore the overall runtime remains O(V + E).

**c)**

Example of an input graph that above may find different cycles on the same graph in the solution above:

Chart, polygon

Description automatically generated

If we begin searching at vertex C and process E first, the solution above would find cycle E-C-D.

If we begin search from vertex C and process B first, the solution above would find cycle B-C-D.

**Shortest-Cycle(G)**

shortestCycle = ∞

for v in G.V:

*# initialize all vertices for BFS*

for v2 in G.V:

v2.visited = false

v2.parent = null

distance = 0

*# perform BFS*

Q = new Queue()

v.visited = true

ENQUEUE(Q, v)

while Q is not empty:

u = DEQUEUE(Q)

for t in Adj[u]:

if (t.visited = true and t != u.parent)

*# FOUND CYCLE*

shortestCycle = min(shortestCycle, u.distance + t.distance + 1)

else if (t.visited = false)

t.visited = true

t.parent = u

t.distance = u.distance + 1

ENQUEUE(t)

if (shortestCycle = ∞)

*# no cycles found*

return -1

else

return shortestCycle

**Runtime Justification:**

In this algorithm we have to consider all cycles lengths possible from any vertex in the graph. That is, we have to look for cycles where each vertex takes a turn being the starting vertex. To look for cycles we use BFS. That means we are running BFS once per node. The runtime complexity of BFS is . Performing that work V times leaves with a runtime complexity of .

**d)**

**makeStronglyConnected(G)**

*# run algorithm from class to get an array of all strongly-connected components*

sccs = findStronglyConnectedComponents(G)

*# take any node from the first SCC and combine it with all the other SCCs in the graph*

firstScc = sccs[0]

u = firstScc[0]

*# for any node in all the other SCCs, connect it to this first node*

for i = 1 to V.G.size

t = sccs[i][0]

*# add bidirectional edge (2 connections) between u and t*

t.parent = u

u.parent = t

* The idea of this algorithm is to find all the existing strongly-connected components and link them all together.
* *Step 1:* use algorithm from class to find all strongly-connected components and store the result in an array of arrays. Finding all SCCs in a graph takes time .
* *Step 2:* select the first SSC from the output, and select any node from that set. Call this node *u*. This step takes constant time.
* *Step 3:* loop over all the remaining SCCs. For each SCC, select any node and make a bidirectional connection with node *u*. This takes time .
* Once step 3 is complete, we have a bidirectional connection from the first SCC to all other SCC in the graph. By definition, each node within each SCC can be reached from all other nodes in the same SCC. With these new edges, each SCC can be reached from all other SCCs.
* For *k* strongly-connected components, we add *at most* 2(k-1) new directed edges, which is less than *2k*.
* Therefore, the overall runtime of the algorithm is which simplifies to .

**e)**

*# global list*

L = newList()

**PrintSorts(G)**

M = newList()

For each v in G.V:

if v.visited = false and v.indegree = 0

append(M, v)

while M is not empty:

*# process current source vertex and append to L*

u = remove(M)

append(L, u)

u.visited = true

*# if all nodes have been added to L, a topological sort has been found: print*

if (L.size = G.V.size)

print(L)

*# process the current vertex’s neighbors, decrementing their indegree*

for each t in Adj[u]:

t.indegree = t.indegree – 1

*# recursively call this algorithm with the previous changes having been made*

PrintSorts(G)

# *undo previous changes to backtrack repeat the search for other possible sorts*

u.visited = false

for each t in Adj[u]

t.indegree = t.indegree + 1

remove(L, u)

**2.**

**a)**

Step 0: Starting Graph Step 2: Remove edge with weight 10

**A picture containing text, sky, wire, line

Description automatically generatedChart, radar chart

Description automatically generated**

Step 3: Remove edge with weight 9 Step 4: Remove edge with weight 8

**Chart

Description automatically generatedA picture containing text, sky, line

Description automatically generated**

Step 5: Remove edge with weight 7 Step 6: Remove edge with weight 6

A picture containing text, sky, line

Description automatically generatedA picture containing text, sky, line, day

Description automatically generated

Step 7: Examine edge with weight 5. Removing would disconnect the graph, so do not remove.

Step 8: Remove edge with weight 4 Step 9: Remove edge with weight 4

A picture containing text, sky, skiing, line

Description automatically generatedChart

Description automatically generated

Step 10: Examine rightmost edge with weight 3. Removing would disconnect the graph, so do not remove.

Step 11: Remove leftmost edge with weight 3.

A picture containing text, sky, day

Description automatically generated

All other edges would disconnect the graph if they were removed; thus this is the MST.

The total of the weights is: 5+ 2 + 3 + 2 + 1 + 2 + 1 = 16

* Kruskal’s algorithm would produce the same MST.
* Prim’s algorithm would also produce the same MST.

**MST(G)**

*# loop over all edges*

edges = newList()

for e in G.E:

append(edges, e)

*# sort edges in descending order, e.g. using quicksort*

sort(edges)

*# determine all bridges in the*

cutEdges = newList()

for e in G.E:

remove(G, e)

numVertices = DFS(G)

add(G, e)

if (numVertices != G.V.size)

*# this is a cut edge*

append(cutEdges, e)

*# loop through edges descending by weight, removing weights only if they are not cut edges*

for e in edges:

cutEdge = find(cutEdges, e)

if (cutEdge == nil):

remove(G, e)

**Runtime Justification:**

The first step is to loop through and sort all the edges in the graph. Using a sorting algorithm like Quicksort, the time complexity of this step is .

The second step is to find all the cut edges. We have to loop through each edge, remove it from the Graph, run DFS on the resulting graph to determine the number of reachable vertices), and add it back into the graph. The time complexity of DFS is ; doing this E times leaves us with a runtime of . Because the number of edges is *at minimum* V and can be much larger, in the worst-case this can be simplified to .

Finally, we have to loop through the edges, perform a look up for each edge to check if it is in an array of cut edges, and remove the edge from the graph if it is not a cut edge. In the worst-case the number of cut edges is equal to the number of total edges, so this step takes roughly time.

The overall runtime of this algorithm is therefore .

**c)**

* Model this problem as two graphs with the same set of vertices representing bus stops. In graph G1, edges represent red bus lines. In graph G2, edges represent blue bus lines.
* We want to know if we can get from vertex X to vertex Y using exactly one transfer. First, we can run a modified version of BFS in G1 and G2 where X is the source node that returns all visited nodes. This gives us all the bus stops we can reach if we started on the red bus line or the on blue bus line.
* If either DFS result can get us to Y, we can return true.
* Otherwise, we need to run DFS on G1T and G2T where Y is the source node. This gives us all the vertices from which it is possible to reach Y using either the red or blue lines.
* Finally, we just need to cross-reference the vertices from the first DFS results with the vertices from the second DFS results. If there is at least one vertex in common, then we know we can get from X to Y using a single transfer. Note: we have to run this comparison twice: once in case we started on a red line and switched to blue and vice-versa.

**BusRoute(G, X, Y)**

redLineVerticesFromX = BFS(G1)

for each v in redLineVerticesFromX:

*# if we can reach Y along just the red line*

if v = Y return true

blueLineVerticesFromX = BFS(G2)

for each v in blueLineVerticesFromX:

*# if we can reach Y along just the blue line*

if v = Y return true

G1\_T = transpose(G1)

redLineVerticesToY = BFS(G1\_T)

for each v1 in blueLineVerticesFromX:

for each v2 in redLineVerticesToY:

*# can reach Y by transferring from blue to red*

if v1 = v2 return true

G2\_T = transpose(G2)

blueLineVerticesToY = BFS(G2\_T)

for each v3 in redLineVerticesFromX:

for each v4 in blueLineVerticesToY:

*# can reach Y by transferring from red to blue*

if v3 = v4 return true

BFS(G)

*# visited nodes*

visited = newList()

*# initialize graph*

for v in G

v.visited = false

v.parent = nil

Q = new Queue()

s.visited = true

ENQUEUE(Q, s)

while Q is not empty:

u = DEQUEUE(Q)

u.visited = true

append(visited, u)

for v in Adj[u]

if v.visited = false

ENQUEUE(Q, v)

return visited

**Runtime Justification**

* The time it takes to run DFS is
* The time it takes to loop through all red and blue vertices is in the worst case because there are *n* vertices; if red connects half the vertices and blue connects to other half, then we have an outer loop of with an inner loop of *n/2* which is still .
* Therefore the overall runtime is .

**d)**

* If we update Dijkstra’s algorithm to update to the *longest* path instead of the *shortest* path the algorithm would *not* work for any graph.
* Here is a counter example:

A picture containing text, clock

Description automatically generated

* Say in this graph we begin with Node 1. We look at (A, B) and (A, C), updating their max lengths accordingly:

A picture containing text, clock, vector graphics

Description automatically generated

* The outcome of the algorithm will depend on which node we examine next: B or C. Both have equal weight. Let’s say we process Node C next. We examine (C, D). The updated graph will now look like:

Diagram

Description automatically generated with medium confidence

* Nodes A and C are now visited. Now we examine Node B. The distance from B to D is 1 + 1 = 2, which is less than the distance between A-C-D. Therefore we do not update. However, the distance from B-C is now updated because it is larger:

Diagram

Description automatically generated with medium confidence

* Now nodes A, B, and C have all been visited. Finally, we examine Node D which is a sink node, so we do nothing. The algorithm is now complete.
* It is apparent from running this algorithm that we have failed to find the longest path to Node D. The path is currently set to A-C-D, which has distance 3. However, the *actual* longest path is A-B-C-D, which has distance 4.
* Therefore, running Dijkstra’s with the longest-path modification failed for this graph. It did not find the true longest path for Node D.
* It is clear that the *order* in which we processed nodes of equal weight mattered. Because we processed Node C first, Node D was updated with Node C’s longest distance *at that point* in time, which was 1. It was not updated with Node C’s *true* longest distance, which is 2.
* The issue is therefore that *longest path updates don’t propagate through the graph*. That is, a given node can only update the longest path for all of its current neighbors, not it’s neighbor’s neighbors and so on. However, Dijkstra’s algorithm only updates the distances for the nodes in the queue and does not reconsider a node once it marks it as visited even is a longer path exists.
* This reasoning is actually quite similar to why Dijkstra’s algorithm fails for negative weights.

**e)**

* The runtime for Dijkstra’s algorithm is given by . In this case we know we have *n* points on the plane, therefore . Furthermore, each point is connected to every other point (“every pair of points has an edge between them). Therefore this is a complete graph and the number of edges is given by which simplifies asymptotically to . That is, .
* Therefore, using the above reasoning, the runtime on this graph model is

**SSSP()**

*# initialize vertices*

for each v in G.V:

v.distance =

v.visited = false

*# initialize source vertex*

s.distance = 0

s.visited = true

*# keep track of visited vertices versus total number available*

totalNumVertices = G.V.size

totalVerticesVisited = 1

while (totalVerticesVisited != totalNumVertices)

minDistance =

u = nil

for each v in G.V

if v.visited = false and v.distance < minDistance

minDistance = v.distance

u = v

for each t in Adj[u]

t.distance = min(t.distance, u.distance + w[u, t)

u.visited = true

totalVerticesVisited += 1

* In this algorithm we have a while loop running over all vertices V, and an inner for-loop that also loops through each vertex looking for the vertex with the current minimum distance. Overall this gives us a runtime of ). Because we have n vertices, the runtime is therefore
* This is a better runtime than the algorithm using a priority queue.