**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 G.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.key)

for each v in Adj[u]

if v.visited = false

DFS-visit(u)

**b)**

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

*# initialize all vertexes for BFS*

for each v in G.V:

v.visited = false

v.parent = nil

Q = new Queue()

*# loop over all vertices to handle unconnected graphs*

for each v in G.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 searching 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

v2.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.

* Reinitialization takes time overall, because we have to loop through all vertexes once for every vertex.
* To look for cycles we use BFS. That means we are running BFS once per vertex. The runtime complexity of BFS is . Performing that work V times leaves with a runtime complexity of .
* Overall, the runtime complexity is therefore which is just .

**d)**

**makeStronglyConnected(G)**

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

*# call DFS-visit with timestamps*

DFS-visit-with-timestamps(G)

*# create a new graph which is the result of reversing all edges in G called G\_T*

G\_T = transpose(G)

sortedVertexesDescFinishTime = sort G\_T.V in order of decreasing finish time

*# re-initialize G\_T vertexes for next DFS search*

for each v in sortedVertexesDescFinishTime:

v.visited = false

v.parent = nil

sccs = newList()

k = 0

for each v in sortedVertexesDescFinishTime:

if v.visited = false:

*# add the first node of the strongly connected component as the DFS root*

sccs.add(v)

# *increment number of strongly connected components by 1*

*k = k + 1*

*# perform DFS search to find all vertexes connected to this one,*

*# marking each one as visited in the process*

DFS-visit-with-children(v)

*#* *create an edge between nodes from each strongly connected component*

*# e.g: SCC1 <-----> SCC2 <-----> SCC3 <-----> SCC4*

i =1

while i < k

u = sccs[i - 1]

v = sccs[i]

*# create a pair of edges between any two nodes from these components*

Adj[u].add(v) *# add edge going from u to v*

Adj[v].add(u) *# add edge going from v to u*

i = i + 1

**Runtime:**

* 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 :
  + For the algorithm, we run DFS twice over all vertices and edges. Both runtimes take time .
  + We also sort the nodes, but because timestamps are integers and we know the range, we can use a linear sorting approach. This takes time where k is some finite timestamp, so .
  + Transposing a graph takes time (because we perform work for each vertex and each edge).
  + Therefore the overall runtime is .
* Step 2: Create a pair of edges between nodes from each strongly connected component.
  + Loop over all SCCs in pairs. For each iteration, take a vertex from each SCC and add it to the other vertex’s adjacency list. The work performed at each iteration is constant, the loop itself takes time .
* Once step 3 is complete, we have a bidirectional connection between all SCCs 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. Therefore the entire graph is now strongly-connected.
* 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 is .

**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.

An image of the result of running Kruskal’s is shown below. The red lines indicate the final MST, which has the same shape as the MST pictured above. The numbers circled in red represent the steps in the algorithm. First, we make connections between both edges with weight 1. These connect 2 vertices each. Next, we connect all edges with weight 2. These connect 6 vertexes into a single component. Next we connect the rightmost edge with weight 3, ignoring the leftmost edge with weight 3 because its vertexes are already connected. We skip edges of weight 4 because their vertexes are already connected. Finally we connect the edge with weight 5. This connects all the vertexes and creates the MST.

Diagram

Description automatically generated

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

The red lines represent the final MST. We can see it has the same shape as the graph above. The steps of running the algorithm are the numbers circled in red. Beginning with the vertex with distance 0, we proceed connecting the rest of the vertices based on their proximity to the growing MST.

Diagram

Description automatically generated

**MST-Delete(G)**

*# loop over all edges, adding them to a separate list*

edges = newList()

for e in G.E:

append(edges, e)

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

edges-in-descending-order = sort-descending(edges)

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

for edge in edges-in-descending-order:

# get nodes from edge

u = edge[0]

v = edge[1]

*# remove edge from graph*

G.Adj[u].remove(v)

G.Adj[v].remove(u)

G.E.remove(edge)

*# check to see if the graph is still connected after removing this edge*

*# if not, add the add back in*

if (All-connected(G) = false)

G.Adj[u].append(v)

G.Adj[v].append(u)

G.E.add(edge)

*# check if all vertices in the graph are connected*

**All-connected(G)**:

*# initialize all vertices in the graph as unvisited*

for each v in G.V:

v.visited = false

*#* *pick any vertex to be the source vertex*

s = G.V[0]

*# run DFS to help determine whether all vertices are reachable from any vertex in the graph*

DFS-helper(s)

*# check to see if all vertices have been visited; if not return false since the graph is unconnected*

for each v in G.V:

if v.visited = false

return false

return true

**DFS-helper(u):**

u.visited = true

for each v in G.Adj[u]:

if v.visited = false

v.parent = u

DFS-helper(v)

**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 . Looping through all the edges takes time .

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 if necessary, add it back into the graph. The time complexity of DFS is ; perming this step E times leaves us with a runtime of . We notice that this can be simplified to :

* If , then .
* If , then

Finally, we have to loop through the vertices to determine if the number of visited vertices is unchanged. This takes time . Performing this step E times has a runtime of .

All together, we have which simplifies to ).

The overall runtime of this algorithm is therefore .

**b)**

**FindClusters(G, *k*):**

*# initialize all vertices to put into the graph, assigning x and y coordinate values for each*

V = newList();

for i = 0 to n:

v = new Vertex()

v.x = p[i].x

v.y = p[i].y

V.append(v)

*# now we have to create edges*

E = newList()

Adj = newAdjacencyList()

*# initialize an adjacency matrix to keep track of edges we’ve already created*

AdjMatrix = initialize list [1..u, 1..v]

*# create an edge between every pair of vertices in the graph*

*# i.e. make a* ***complete graph***

for each v in V

for each u in V

*#* *make sure to check that u does not equal v and that*

*# (u, v) hasn’t already been added*

if u != v and AdjMatrix[u][v] != 1

AdjMatrix[u][v] = 1

d = Compute-distance(v, u) *# get the edge weight between v and u*

e = newEdge(v, u)

set-weight(e, d)

Adj[v].append(u)

Adj[u].append(v)

E.append(e)

*# initialize a new graph with these vertices, edges and adjacency list*

G = new Graph(V, E, Adj)

*# we run Kruskal’s algorithm to build the MST through these vertexes, stopping when*

*# the number of remaining unconnected components equals k*

*# initialize the vertexes and sort the edges*

T = initialize empty Tree

for each v in G.V:

v.mycaptain = v

sorted-edges = sortIncreasingOrder(G.E)

components = []

*# run Kruskal’s algorithm on the edges, stopping early when number of clusters is reached*

countComponents = G.V.size

for each e = (u, v) in sorted-edges:

*# if the number of components is the same as the number of expected clusters,*

*# stop the MST algorithm early to keep from merging the remaining clusters*

if countComponents = k

break

if Find(u) != Find(v)

add edge (*u, v*) to T

Merge(u, v)

if u not in components:

components.add(u)

numComponents += 1

*# having merged two vertexes, the number of components is decreased by one*

countComponents = countComponents - 1

*# print out the clusters*

print(“Point: ClusterNumber:”)

*# T contains all the edges for all the clusters; we want to print out each cluster with its own label*

*# we can do this using the fact that all clusters are stored as linked lists; when we encounter an*

*# unvisited node, we can get its mycaptain node and print out the full linked list*

clusterNumber = 1

for each edge (u,v) in T:

if u.visited = false *# we haven’t processed this cluster*

n = edge.u.mycaptain

while n != nil

print(n.x + “, ” + n.y + clusterNumber)

n.visited = true

n = n.next *# set n to be the next node in the linked list*

clusterNumber = clusterNumber + 1

**Compute-distance(v, u):**

*#* *use the distance formula*

a = (v.x – u.x) \* (v.x – u.x)

b = (v.y – u.y) \* (v.y – u.y)

return sqrt(a + b)

**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-visited(G1)

for each v in redLineVerticesFromX:

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

if v = Y return true

blueLineVerticesFromX = BFS-visited(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-visited(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-visited(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-visited(G)**

*# visited nodes*

visited = newList()

*# initialize graph*

for v in G

v.visited = false

Q = new Queue()

s.visited = true

ENQUEUE(Q, s)

while Q is not empty:

u = DEQUEUE(Q)

u.visited = true

visited.add(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 A**. 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: **Node** **B** or **Node** **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 longest 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]

if t.distance > , u.distance + w[u, t]

t.distance = , u.distance + w[u, t]

t.parent = u

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.

**3.**

**a)**

* A
* Model this problem as a graph, where each vertex *v* is a room, and each edge (*u, v*) represents whether or not two rooms are adjacent to each other. In other words, if an edge (u, v) exists between two vertices, that means the rooms are adjacent. From here we want to see if we can fill all *n* rooms with students and teachers such that no student is adjacent to another student, and no teacher is adjacent to another teacher. We want a pattern such as *teacher-student-teacher-student*. We can achieve this if the graph is *bipartite*. We can determine if this is a bipartite graph by using the graph coloring algorithm from the practice set. That is, we try to color the graph using black and white such that no black node is adjacent to another black node, and no white node is adjacent to another white node. If we can color the graph in this manner, that means we can fill all *n* rooms with students and teacher according to the conditions described above. The graph coloring algorithm uses BFS which has a runtime of , which because we have *n* rooms becomes . This is a polynomial runtime. Therefore the problem is not NP-Complete.
* Model this problem as a graph where each vertex is a type of food, and each edge (u,v) is a student. Each student must select *exactly* two types of food; therefore each student edge (u,v) will be connected to two food vertices. We want to select *k* students such that all foods are the favorite of at least *one* student. That is, we want to select *k* edges such that each vertex is adjacent to at least one selected edge. This is the *edge cover* problem from class, which is known to have a polynomial runtime. Therefore it is not NP-Complete.
* Given the same input, we model the problem in the same way as above: vertices are items of food and edges represent students. In this case, we want to select *k* food items such that each student has at least on of their food preferences selected. That is, we want to select at most *k* vertices such that each edge is adjacent to at least one selected vertex. This is the *vertex cover* problem from class, which is known to be NP-Complete.

**b)**

*Step 1: Show that a solution to this problem can be verified in polynomial time:* Given a solution to *SummerCamp*. Given a group of *m* people, each with a list of items, and a value *k* representing the *least* number of items this group could purchase, we would have to verify that there is at least one distinct person per item. In that way, there is *at least* one person that must buy that item. Therefore, we need to ensure that there are at least *k* distinct people among the *k* items listed. In order to do this, we need to loop through all *k* items; for each item we loop through the group of people that have listed that item as something they wish to purchase. As we do this, we keep a running mathematical *set of unique people* encountered through the iteration. Once we have looked through all items, we verify that the size of the final set is equal to *k*, the least number of items. We have *n* items and a possible number of *m* people per item. Therefore the runtime to loop through all the items, and then through all the people per item, is . Therefore the time to check verify this problem is polynomial.

Step 2: Show this problem is NP complete using a reduction from the Finding Clique problem: an instance to the Finding Cliques problem consists of a graph G such that the goal is to find a group of *k* vertices in the graph that are all *pair-wise* connected. We show how the Finding Clique problem can be solved using *SummerCamp*. The goal is to return yes if and only if there is a clique of size *k*. To create an input to *SummerCamp*, we need to define the people and their relationship with the items they want to purchase. We create an instance of *SummerCamp* as follows: each vertex in G becomes a person. Each edge = (u, v) becomes a link to another person *if and only if* the two people are interested in purchasing the same item. The key is that we constrain the problem by only allowing people to be able to purchase one item to ensure that there can only ever be one edge between the same two people. If a group of people are interested in purchasing the same item, in other words, that group of peoples makes up a *complete graph* where each person is connected by an edge to every other person. So, assuming we want to find a clique of size *s*, we create a group of *s* people that all want to buy the same item. Call this group *Item1*. Then we create an additional *k – 1* people where each additional person wants to buy a different item.

*1) If there is a clique of size s in G, then SummerCamp has at least k items that can be purchased.* Assume G has a clique of size *s*. The vertices in this group correspond to the set of people in group *Item1*. The rest of the groups will be comprised of the people in the rest of the *k – 1* groups we created as input.

*2) If there are at least k items that can be purchased in SummerCamp, then graph G has a clique of size s.* Assume that *SummerCamp* input has *k* independent groups of people wanting to purchase different items. One of those groups is group *Item1*. Therefore graph G has a clique of size *s*.

**c)**

*Step 1: Show that a solution to this problem can be verified in polynomial time:* Given a solution to *TrainSalesman*, we can verify if this solution is valid as follows. Given a set of edges that represent the train connection between cities, and a set of vertices representing the cities those edges connect, we can first loop through all cities (verticies) to make sure that all *n* cities are present on the salesman’s path. This would take time. Then we can loop through the edges to make sure there are exactly edges and that each edge connects a city within *n*. This ensures that there is exactly one edge between each pair of cities. This would also take time . The overall runtime is therefore , which is polynomial.

*Step 2: Show this problem is NP complete using a reduction from Hamilton Path:* an instance to Hamilton Path consists of a graph G such that the goal is to find a path that visits each vertex exactly once. We create an instance of *TrainSalesman* as follows: each vertex in G becomes a city. Each edge e = (u, v) becomes a train route between two cities, where cities *u and v* are connected by e. In addition, we apply a weight of 1 to all edges, representing the cost to travel by train between any two cities. We also set the total input cost of the trip *k* to be *n - 1*. In other words, the total cost of the trip should be equal to the cost of visiting all cities in G exactly once.

*1) If there is a Hamilton Path in G, then there is a way to visit all cities in TrainSalesman with a cost of at most n.*

Assume there is a Hamilton Path in G. The verticies in G correspond to cities that are visiting. We know there are *n* cities, and therefore *n – 1* edges connecting those cities. We know that the cost of moving between two cities is also set to be 1, and that the total cost of the Hamilton Path is therefore *n – 1*. We also know that an existing Hamilton Path means that each city is visited exactly once. Because we want to know if the salesman can visit all cities with a cost of at most *n – 1* and there are *n – 1* train routes, each with a cost of 1, *TrainSalesman* returns true.

*2) If a salesman can visit n cities for a cost of at most n – 1, then there is a Hamilton Path in G.*

If the salesman can visit *n* cities with at most a cost of *n – 1*, we know, because the cost of each train route between cities is 1, that that means there is exactly one distinct edge connecting all cities on the path, and each city is visited exactly once. If a city were visited more than once, the cost would be greater than *n* – 1. Because cities represent vertices in G, we therefore also know that each vertex is visited exactly once. Therefore there is a Hamilton path in G.