Aleks Itskovich

Collaborator: Elsie Kenyon

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

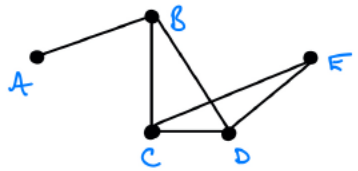
This algorithm is just a modification of BFS.

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 $\boldsymbol{O(V + E)}$.

c)

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



- If we start searching at vertex C and process **E** first, the solution above would find cycle E-C-D.
- If we start searching at 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 $O(V^2)$ 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 $O(V + E)$. Performing that work V times leaves with a runtime complexity of $O\big(V(V + E)\big) = O(VE + V^2)$.
- Overall, the runtime complexity is therefore $O(VE + V^2 + V^2)$ which is just $\boldsymbol{O(VE + V^2)}$**.**

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 =2
    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 $O(V + E)$:
    - For the algorithm, we run DFS twice over all vertices and edges. Both runtimes take time $O(V + E)$.
    - 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 $O(V + k)$ where k is some finite timestamp, so $O(V)$.
    - Transposing a graph takes time $O(V + E)$ (because we perform work for each vertex and each edge).
    - Therefore the overall runtime is $O(V + E)$.
- 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 $O(k)$.
- 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 $O(k + V + E)$ which is $\boldsymbol{O(V + E)}$.

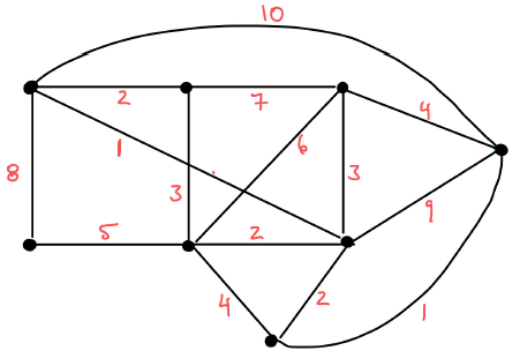
e)

```
# 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)
```

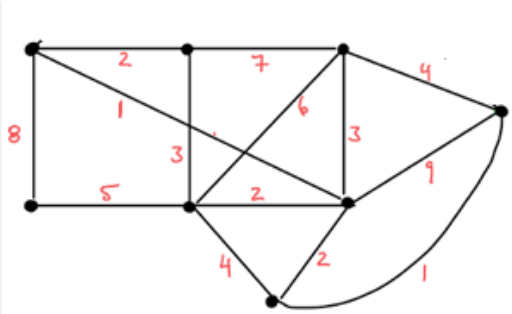
2.

a)

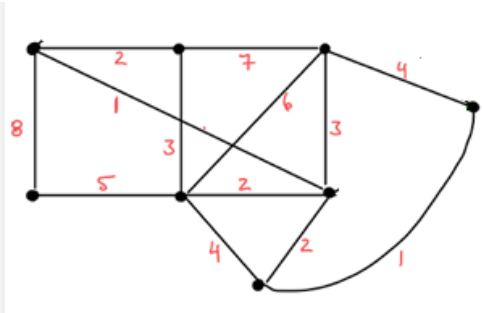
Step 0: Starting Graph



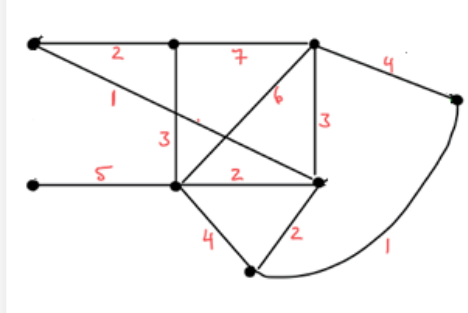
Step 2: Remove edge with weight 10



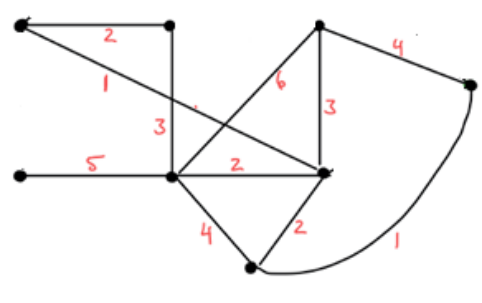
Step 3: Remove edge with weight 9



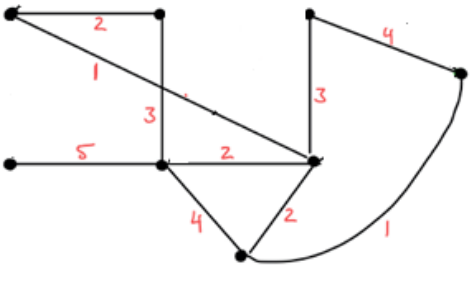
Step 4: Remove edge with weight 8

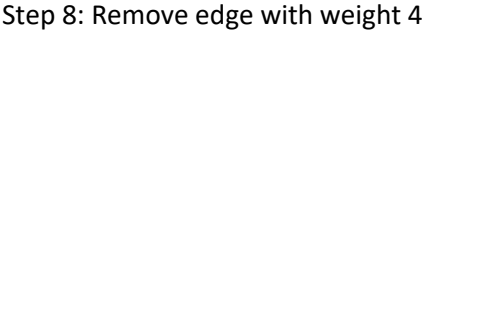


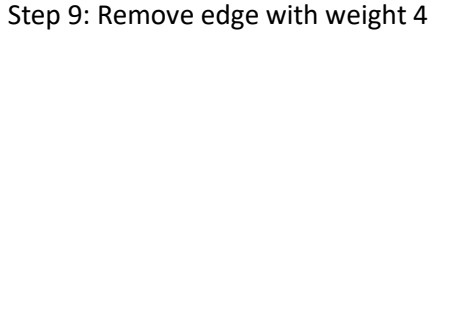
Step 5: Remove edge with weight 7



Step 6: Remove edge with weight 6



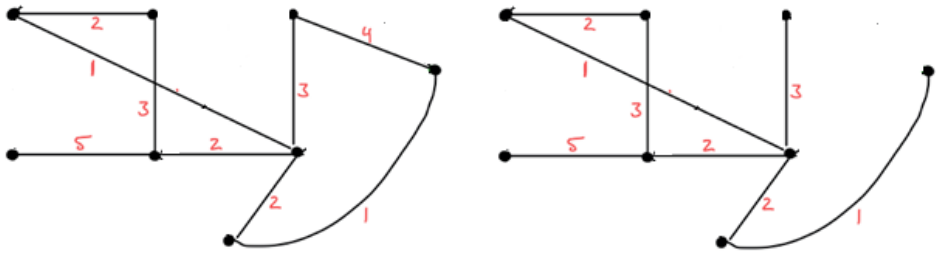




Step 7: Examine edge with weight 5. Removing would disconnect the graph, so do not remove.

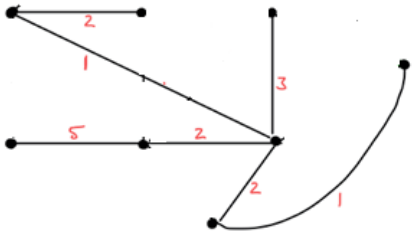
Step 8: Remove edge with weight 4

Step 9: Remove edge with weight 4



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.

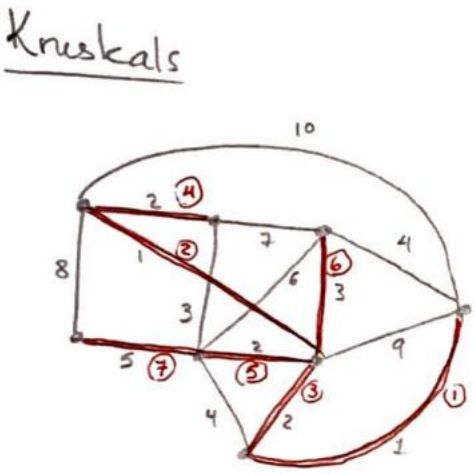


All other edges would disconnect the graph if they were removed; thus this is the MST.

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

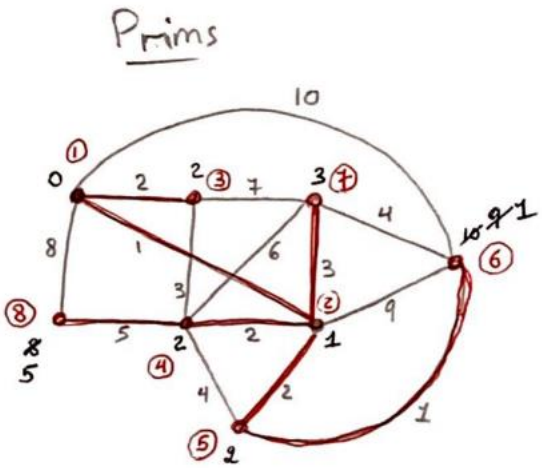
- Kruskal's algorithm would produce the same MST.

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.



- 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.



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 = (u, v) in edges-in-descending-order:
        # 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 $O(E \log E)$. Looping through all the edges takes time $O(E)$.

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 $O(V + E)$; perming this step E times leaves us with a runtime of $O(EV + E^2)$. We notice that this can be simplified to $O(E^2)$:

- If $E = V$, then $O(EV + E^2) = O(2V^2) = O(E^2)$.
- If $E = V^2$, then $O(EV + E^2) = O(V^3 + V^4) = O(V^4) = O((V^2)^2) = O(E^2)$

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

All together, we have $O(E + E \log E + EV + E^2)$ which simplifies to $O(E^2)$.

The overall runtime of this algorithm is therefore $\boldsymbol{O(E^2)}$.

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
    for each v in G.V:
        v.mycaptain = v
    sorted-edges = sortIncreasingOrder(G.E)

    # run Kruskal's algorithm on the edges, stopping when the desired number of clusters is reached
```

```
numComponents = 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)
                Merge(u, v)
                # having merged two vertexes, the number of components is decreased by one
                numComponents = numComponents - 1

# initialize list of mycaptain nodes; each mycaptain node points to a discrete component/cluster
all-captains = new List()

# get all my captain nodes
for v in G.V:
        if v.mycaptain not in all-captains:
                all-captains.add(v.mycaptain)

# print out the clusters
print("Point:                    ClusterNumber:")

# The list components contains all the mycaptain nodes 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; we use each mycaptain node to point to the rest of the points in each cluster
clusterNumber = 1
for each captain in all-captains:
        n = captain
        while n != nil
                print(n.x + ", " + n.y + clusterNumber)
                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)
```

Runtime Justification:

- In the algorithm above, we being with *n* points. We model the problem as a graph G where each point is a vertex. We therefore end up with a graph with *n* vertices.
- Moreover, we model the problem as a *complete* graph. That means that each vertex is connected to every single other vertex in the graph. That means we have V^2 edges; because V = *n*, that means we have n^2 total edges.
- Creating the graph means first creating all the vertices. That means looping through *n* points and converting each point into a vertex. This takes time $O(n)$.
- Next, we have to create all the edges. This means looping though all the points in a nested for-loop to create edges between them. Computing the distance and setting the weight, etc. all takes constant time. Therefore this steps takes total time $O(n^2)$.
- Next, we have to initialize the vertices and sort the edges in descending order. Initialization means looping through each vertex; this takes time $O(n)$. Sorting the edges takes time $O(n^2 \log n)$, assuming we use a sorting algorithm like mergesort.
- We proceed to run Kruskal's algorithm on G. We know from class that the runtime of Kruskal's algorithm is $O(E \log V)$. In our algorithm, we actually stop when the number of clusters = k, so we don't run Kruskal's as many times, but this is a relatively small constant factor and does not change the overall runtime of the algorithm. Therefore, because $E = V^2$ and $V = n$, running Kruskal's algorithm on G takes time $O(n^2 \log n)$.
- After running Kruskal's we need to find all the mycaptain vertices; this means looping through *n* vertices and filtering for new mycaptain vertices. This takes time $O(n)$.
- Finally, for each captain, we print out the linked list of vertices associated with it. All together we print out each vertex once, so the work done is $O(n)$.
- We see from the analysis above that the dominant runtime step takes time $(n^2 \log n)$. Therefore the runtime for the entire algorithm is $(n^2 \log n)$.

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.
- Step 1: we run a *modified* version of BFS-visited on G1 where X is the source node. This DFS algorithm returns a list of all nodes that can be visited from x. In other words, this gives us all the bus stops we can reach if we started on the red bus line.
    - Next, we loop through all the vertices. If Vertex Y is in the list of vertices that can be reached from Vertex X using G1, we return true.
- Step 2: if we *cannot* reach Vertex Y in Step 1, we repeat the steps above but now using G2, i.e. the blue line.
    - We loop through all the vertices returned by BFS-visited on the blue line. If Vertex Y is in that list of vertices that can be reached from Vertex X using G2, we return true.
- Step 3: if we *cannot* reach Vertex Y in step 2, we proceed as follows:
    - First we take the transpose of G1, called $G1^T$. Next we run BFS-visited on $G1^T$ using Vertex Y as the source node. This gives us This gives us all the vertices from which it is possible to reach Vertex Y using the red line.
    - Next, we loop through all the blue vertices that can be reached *from* Vertex X. Let's call these *blue vertices from X.* For each *blue vertex from X* we loop through all the vertices from which we can reach Vertex Y. Let's call these *red vertices to Y*.
    - If we find any overlap between *blue vertices from X* and *red vertices to Y*, that means we have found a bus stop at which we can transfer from the blue line to the red line such that we are able to reach Vertex Y from Vertex X.
    - If such a vertex is found, we return true.
- Step 4: If Step 3 does not return true, we repeat the operations in Step 3, but now using the transpose of G2, and checking if there is an overlap between *red vertices from X* and *blue vertices to Y*. If there is at least one vertex in common, then we know we can get from X to Y using a single transfer from red to blue.
- If Step 4 does not return true, we return false for the entire algorithm.

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

    return false
```

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

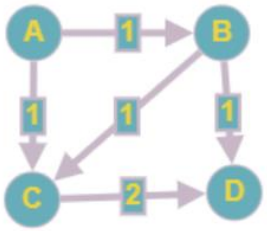
- The time it takes to run DFS is $O(V + E)$
- For n cities, this becomes $O(n + E)$.
- The time it takes to loop through all red and blue vertices is in the worst case $O(n^2)$ because there are *n* vertices and we have a nested for-loop: looping through all the red vertices, and

then looping through all the blue vertices. Even if we have $\frac{n}{2}$ edges in each list, the runtime would still be $O\left(\frac{n}{2}\right)^2 = O(n^2)$.

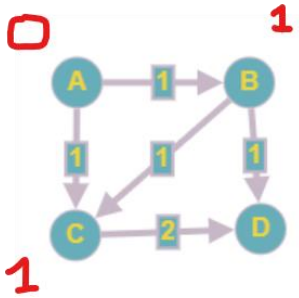
- Therefore the overall runtime is $O(n^2)$.

d)

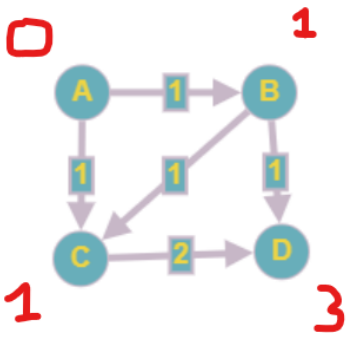
- If we update Dijkstra's algorithm to update to the *longest* path instead of the *shortest* path the algorithm would *not* work for any graph.
- Here is a counter example:



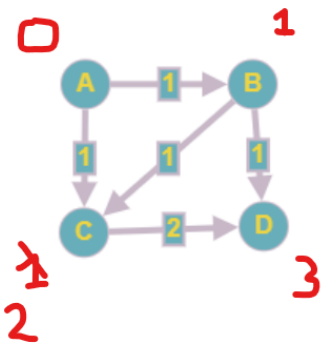
- Say in this graph we begin with **Node A**. We look at **(A, B)** and **(A, C)**, updating their max lengths accordingly:



- 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:



- **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:



- 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 $O(E \log V)$. In this case we know we have *n* points on the plane, therefore $V = n$. 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 $\frac{n(n-1)}{2}$ which simplifies asymptotically to n^2. That is, $E = n^2$.
- Therefore, using the above reasoning, the runtime on this graph model is $\boldsymbol{O(n^2 \log n)}$

SSSP(G)

```
# initialize vertices
for each v in G.V:
        v.distance = ∞
        v.visited = false

# initialize some source vertex
s.distance = 0
s.visited = true

# keep track of visited vertices versus total number available
totalNumVertices = G.V.size
totalVerticesVisited = 1

# instead of checking for an empty Queue, check if all vertices have been visited
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 $O(V^2)$. Because we have n vertices, the runtime is therefore $O(n^2)$
- This is a better runtime than the algorithm using a priority queue for the graph given above.

3.

a)

- *Polynomial.* Model this problem as a graph. Each vertex represents a city, and each edge (u, v) represents a taxi route between two cities. We want to determine a set of 4 cities, such that the cost of getting between the cities is the cheapest in the entire graph. There are exactly *n* choose *4* ways of selecting sets of 4 cities. This takes time $O(n^4)$. From these sets of cities we can find just those cities whose edges form a straight path. Checking to see if we have a path between cities takes time $O(1)$. For each set of selected cities connected by a path, we simply have to add up the weights of the edges between all cities and determine if they correspond to the minimum cost overall. This also takes time $O(1)$. Therefore we perform a constant amount of work *n* choose *4* times, which corresponds to a polynomial runtime. In addition, producing *n choose 4* combinations is also polynomial. Therefore the entire runtime is polynomial.
- *Polynomial.* 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 (Practice Set 12, Problem 7). 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 $O(V + E)$, which because we have *n* rooms becomes $O(n + n^2) = O(n^2)$. This is a polynomial runtime. Therefore the problem is not NP-Complete.
- *Polynomial.* 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.
- *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*. Our goal is to find if we can purchase at least *k* items given *m* people that can each list *n* items. (1) Each person can only buy one item, and (2) if a given item is bought, any person with that item on their list must buy it. Given a solution where we are told which *k* items we can purchase, we can verify it in polynomial time as follows. We need to loop through the set of *m* people to check if there is a way to buy these *k* items. For each person, we loop through their list of items, checking to make sure that none of the selected items appears together on the same list. The runtime for looping through *m* people and checking *n* items is $O(nm)$. Therefore the entire runtime is polynomial.

Step 2: Show this problem is NP-complete using a reduction from the Independent Set problem: an instance to the Independent Set problem consists of a graph G such that the goal is to find a group of *at least k* vertices in the graph that are not adjacent to each other. We show how the Independent Set problem can be solved using *SummerCamp*. To create an input to *SummerCamp* we need to define the items and their relationship to each other. We create an instance of *SummerCamp* as follows: each vertex represents an item available for purchase. An edge = (*u, v*) connects two vertices (items) if and only if both items appear together on some person's list. In other words, if Person A lists Items 1 and 2, then there is an edge connecting Items 1 and 2. This implies that if two or more items are listed together, it is only possible to purchase *at most* one of those items. This is because the person listing them can only contribute to buying one item, and any item can only be purchased if *all* the people that listed it are able to buy it. Another way of saying this is that multiple items can only be purchased of they never appear together on the same list.

1) If there is an independent set of at least size k in G, then SummerCamp has at least k items that can be purchased. Assume G has an independent set of size *k*. The vertices of the group correspond to items available for purchase. If we can select at least *k* vertices that are not adjacent to any neighboring vertices, that means that we can select at least *k* items that never appear together on any list. Again, when two items are connected by an edge, that means they appear together on at least one list. If we select vertices such that there are no neighbors (as we do in independent set), we are simultaneously selecting items that never appear on the same list. In other words, there are *k* distinct, mutually-exclusive groups of people able to purchase those items. Therefore we know that there are *at least k* items that can be bought.

2) If there are at least k items that can be purchased in SummerCamp, then graph G has an independent set of size k. If there are at least *k* items that can be purchased, that means there are *k* items that are not adjacent to each other in the graph. They never appear together in the same list. Because they do not appear together on any list, there are no edges connecting them. If there are no edges connecting them, that means that they are not neighbors. That means that in our graph we have selected *k* vertices, none of which are adjacent to each other. This corresponds exactly to an independent set of size *k*.

c)

Step 1: Show that a solution to this problem can be verified in polynomial time: Given a solution to *TrainSalesman* (the path the Salesman took, a set of vertices and edges), 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 $O(n)$ time. Then we can loop through the edges to make sure there are exactly $n - 1$ edges and that each edge connects a city within *n*. This ensures that there is exactly one edge between each pair of cities. Finally, we add up the cost of each edge and ensure that it is less than or equal to *k*. This would also take time $O(n)$. The overall runtime is therefore $O(n)$, 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 *k* of the salesman's trip 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 exactly once 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.