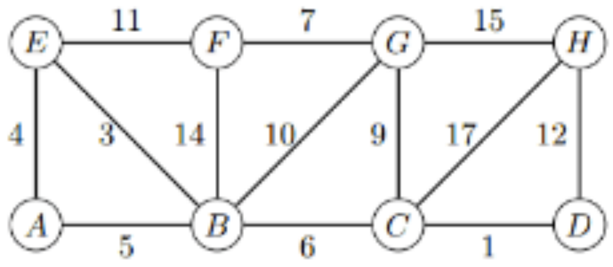
Kelsey Helms
CS 325
11/3/16
Homework 3

1. **Consider the weighted graph below:**



   a. **Demonstrate Prim's algorithm starting from vertex A. Write the edges in the order they were added to the minimum spanning tree.**

Step	Edge	Root	Priority Queue	Path
1		{A}	{E(4), B(5)}	{A}
2	AE	{A, E}	{B(3), F(11)}	{A, E}
3	EB	{A, B, E}	{C(6), G(10), F(11)}	{A, E, B}
4	BC	{A, B, C, E}	{D(1), G(9), F(11), H(17)}	{A, E, B, C}
5	CD	{A, B, C, D, E}	{G(9), F(11), H(12)}	{A, E, B, C, D}
6	CG	{A, B, C, D, E, G}	{F(7), H(12)}	{A, E, B, C, G}
7	GF	{A, B, C, D, E, F, G}	{H(12)}	{A, E, B, C, G, F}
8	DH	{A, B, C, D, E, F, G, H}		{A, E, B, C, D, H}

   b. **Demonstrate Dijkstra's algorithm on the graph, using vertex A as the source. Write the vertices in the order which they are marked and compute all distances at each step.**

Step	Vertex	Root	Priority Queue	Path
1	A	{A}	{E(4), B(5)}	{A} = 0
2	E	{A, E}	{B(5), F(15)}	{A, E} = 4
3	B	{A, B, E}	{C(11), F(15), G(15)}	{A, B} = 5
4	C	{A, B, C, E}	{D(12), F(15), G(15), H(28)}	{A, B, C} = 11
5	D	{A, B, C, D, E}	{F(15), G(15), H(24)	{A, B, C, D} = 12
6	F	{A, B, C, D, E, F}	{G(15), H(24)}	{A, E, F} = 15
7	G	{A, B, C, D, E, F, G}	{H(24)}	{A, B, G} = 15
8	H	{A, B, C, D, E, F, G, H}		{A, B, C, D, H} = 24

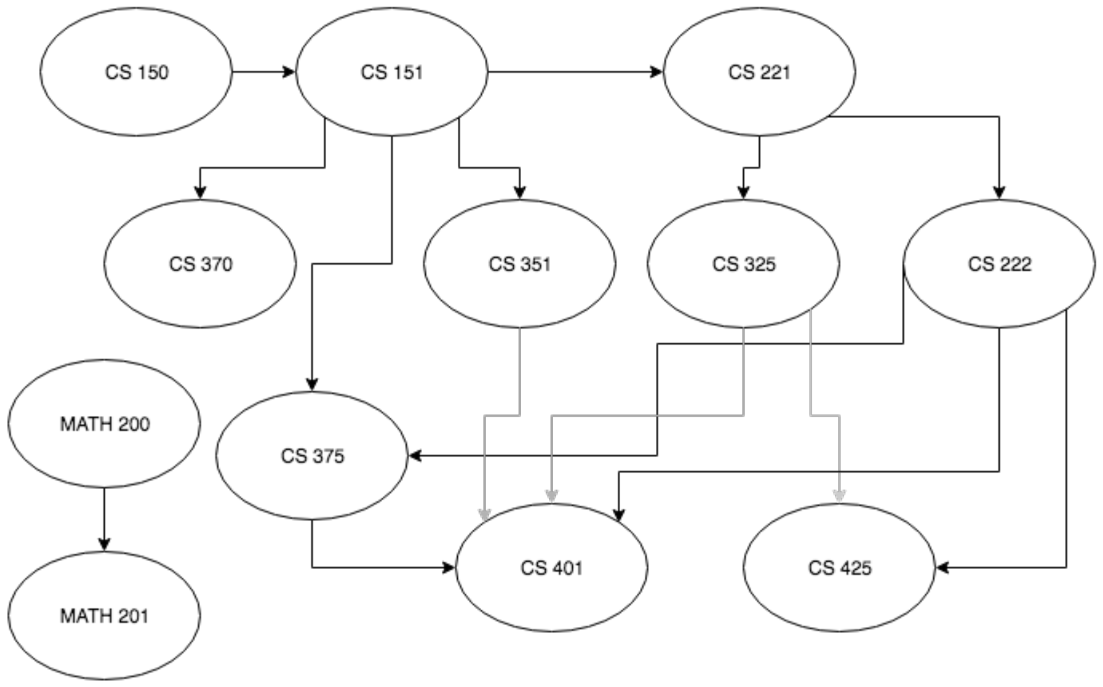
2. **A Hamiltonian path in a graph G = (V, E) is a simple path that includes every vertex in V. Design an algorithm to determine if a directed acyclic graph (DAG) G has a Hamiltonian path. Your algorithm should run in O(V+E). Provide a written description of your algorithm including why it works, pseudocode and an explanation of the running time.**
   First, topologically sort the DAG in O(V+E). Once this is done, by definition of topological sort, the edges go from higher index vertices to lower. This means that there exists a Hamiltonian path if and only if there are edge between consecutive vertices, e.g. (1, 2), (2, 3), ..., (n-1, n). This is because a Hamiltonian path can't go backwards, yet all vertices must be visited, so the only way is to not skip any vertex. Because you are checking every vertex, the complexity is O(V). Thus, the overall complexity is O(V+E).

```
L ← Empty list that will contain the sorted vertices
while there are unmarked vertices
     select an unmarked vertex n
     visit(n)
return L

function visit(vertex n)
     if n had not been visited
          for each vertex m with an edge from n to m
               visit(m)
          mark n
          add n to head of L

for each index of L
     if index does not point to index + 1
          no Hamiltonian Path
return L as Hamiltonian Path
```

3. **Below is a list of courses and prerequisites for a factious CS degree.**

Course	**Prerequisite**
CS 150	**None**
CS 151	**CS 150**
CS 221	**CS 151**
CS 222	**CS 221**
CS 325	**CS 221**
CS 351	**CS 151**
CS 370	**CS 151**
CS 375	**CS 151, CS 222**
CS 401	**CS 375, CS 351, CS 325, CS 222**
CS 425	**CS 325, CS 222**
MATH 200	**None**
MATH 201	**MATH 2000**

a. **Draw a directed acyclic graph (DAG) that represents the precedence among the courses.**



b. **Give a topological sort of the graph.**
One instance of a topological sort is MATH 200, MATH 201, CS 150, CS 151, CS 370, CS 351, CS 221, CS 325, CS 222, CS 375, CS 401, CS 425.

c. **Find an order in which all the classes can be taken. You are allowed to take multiple courses at one time as long as there is no prerequisite conflict.**

Quarter	Classes
1	CS 150, MATH 200
2	CS 151, MATH 201
3	CS 221, CS 370, CS 351
4	CS 222, CS 325
5	CS 375, CS 425
6	CS 401

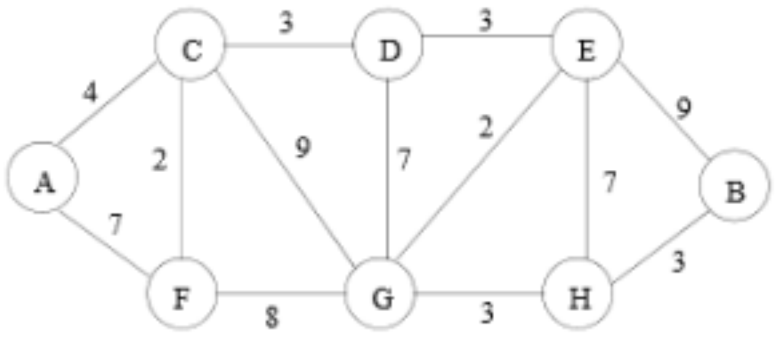
d. **Determine the length of the longest path in the DAG. How did you find it? What does this represent?**
$CS\ 150 \rightarrow CS\ 151 \rightarrow CS\ 221 \rightarrow CS\ 222 \rightarrow CS\ 375 \rightarrow CS\ 401]$
This is the longest path, which represents the most amount of quarters required to complete the degree. It can be found using topological sort, putting courses in order such is in c., and finding the courses with dependencies.

4. **Suppose you have an undirected graph G = (V, E) and you want to determine if you can assign two colors (blue and red) to the vertices such that adjacent vertices are different colors. This is the graph Two-Color problem. If the assignment of two colors is possible, then a 2-coloring is a function C: V ->{blue, red} such that C(u) ≠ C(v) for every edge (u, v) ∈ E. Note: a graph can have more than one 2-coloring.**
   Give an O(V + E) algorithm to determine the 2-coloring of a graph if one exists or terminate with the message that the graph is not Two-Colorable. Assume that the input graph G = (V, E) is represented using adjacency lists.
   a. **Give a verbal description of the algorithm and provide detailed pseudocode.**
      Because two vertices of the same color cannot touch each other, you must color one vertex then find all adjacent vertices to color the other color. This continues on until all vertices have been colored. If at any point this is not possible, then the graph isn't two-colorable.

```
L ← Empty list that will contain the colored vertices
while there are uncolored vertices
   select an uncolored vertex n
   color(n)
return L

function color(vertex n)
   if n has not been colored
         for each vertex m with an edge from n to m
               if some m == color1 and some m == color2
                     return "Graph is not Two-Colorable"
               if m == color1
                     n = color2
               if m == color2
                     n = color1
               if m is unknown
                     n = color1
                     color(m)
         add n to L
```

   b. **Analyze the running time.**
      Because you only check each vertex and each edge once, and all other operations are in constant time, the running time is O(V+E).

5. **A region contains a number of towns connected by roads. Each road is labeled by the average number of minutes required for a fire engine to travel to it. Each intersection is labeled with a circle. Suppose that you work for a city that has decided to place a fire station at location G. (While this problem is small, you want to devise a method to solve much larger problems).**



a. **What algorithm would you recommend be used to find the fastest route from the fire station to each of the intersections? Demonstrate how it would work on the example above if the fire station is placed at G. Show the resulting routes.**

Step	Vertex	Root	Priority Queue	Path
1	G	{G}	{E(2), H(3), D(7), F(8), C(9)}	{G} = 0
2	E	{E, G}	{H(3), D(5), F(8), C(9), B(11)}	{G, E} = 2
3	H	{E, G, H}	{D(5), B(6), F(8), C(9)}	{G, H} = 3
4	D	{D, E, G, H}	{B(6), F(8), C(8)}	{G, E, D} = 5
5	B	{B, D, E, G, H}	{F(8), C(8)}	{G, H, B} = 6
6	F	{B, D, E, F, G, H}	{C(8), A(15)}	{G, F} = 8
7	C	{B, C, D, E, F, G, H}	{A(12)}	{G, E, D, C} = 8
8	H	{A, B, C, D, E, F, G, H}		{G, E, D, C, A} = 12

b. **Suppose one "optimal" location (maybe instead of G) must be selected for the fire station such that it minimizes the distance to the farthest intersection. Devise an algorithm to solve this problem given an arbitrary road map. Analyze the time complexity of your algorithm when there are f possible locations for the fire station (which must be at one of the intersections) and r possible roads.**

```
def allPairs(graph, vertices):

    # set map dist to graph
    dist = map(lambda i : map(lambda j : j , i) , graph)

    # pick k as an intermediate vertex
    for k in range(V):

        # pick i as source vertex
        for i in range(V):

            # pick j as destination vertex
            for j in range(V):
                # If vertex k is on the shortest path
                # from i to j, then update the value of
                # dist[i][j]
                dist[i][j] = min(dist[i][j], dist[i][k]
                     + dist[k][j])

    # array to keep longest distances for each location
    maxDist = [0] * V

    # for each vertex as source
    for a in range(V):
         maxDist[a] = dist[a][0]

         # pick furthest destination
         for b in range(V):
              maxDist[a] = max(maxDist[a], dist[a][b])

    # shortest distance to furthest locations
    shortest = maxDist[0]
    # location that has the shortest furthest distance
    optSpot = 0

    # pick shortest furthest distance
    for s in range (V):
         shortest = min(shortest, maxDist[s])
         if shortest == maxDist[s]
              # if new shortest, update location
              optSpot = s

    # this is the optimal location
    return optSpot
```

The time complexity of the above algorithm is O(V^3). This is because the for loops make the algorithm CV^3 +CV^2 + CV + C. By dropping all constants and all terms but the dominant term, we get O(V^3).

c. **In the above graph what is the “optimal” location to place the fire station? Why?**
By running this algorithm on the graph provided, the optimal location is E. The maximum distances for each location is A->B = 18, B->A = 18, C->B = 14, D->B = 11, E->A = 10, F->B = 14, G->A = 12, H->A = 15. Because E->A = 10 is the shortest of these, E is the optimal location.

d. **EXTRA CREDIT: Now suppose you can build two fire stations. Where would you place them to minimize the farthest distance from an intersection to one of the fire stations? Devise an algorithm to solve this problem given an**

arbitrary road map. Analyze the time complexity of your algorithm when there are f possible locations for the fire station (which must be at one of the intersections) and r possible roads.

When the following algorithm is run, it is clear that the best place for the two fire stations would be at location C and location H.

The time complexity of the following algorithm is O(V^3). This is because the for loops make the algorithm C2V^3 + CV^2 + CV + C. By dropping all but the dominant term, we get O(V^3).

```
def allPairs(graph, vertices):

# set map dist to graph
dist = map(lambda i : map(lambda j : j , i) , graph)

# pick k as an intermediate vertex
for k in range(V):

     # pick i as source vertex
     for i in range(V):

          # pick j as destination vertex
          for j in range(V):

               # If vertex k is on the shortest path
               # from i to j, then update the value of
               # dist[i][j]
               dist[i][j] = min(dist[i][j], dist[i][k]
                    + dist[k][j])

# number of current ordering
count = 0
# number of total pairs
place = ((V-1)*V)/2

#keeps track of min distance between the two locations
bestDist = [[0 for x in range(V)] for y in
range(place)]

# pick a as first location
for a in range(V-1):

     # pick b as second location
     for b in range(a+1, V):

          # pick c as destination
          for c in range (V):

               # if a is closer to c, keep distance
               # from a
               if dist[a][c] < dist[b][c]:
                    bestDist[count][c] = dist[a][c]

               # else keep distance from b
               else:
                    bestDist[count][c] = dist[b][c]

          # increase number of current ordering
          count = count +1

# array to keep longest distances for each location
maxDist = [0] * place

# for each vertex as source
for a in range(place):
     maxDist[a] = bestDist[a][0]

     # pick furthest destination
     for b in range(V):
          maxDist[a] = max(maxDist[a], bestDist[a][b])

# shortest distance to furthest locations
shortest = maxDist[0]
# location that has the shortest furthest distance
optSpot = 0

# pick shortest furthest distance
for s in range (place):
     shortest = min(shortest, maxDist[s])
     if shortest == maxDist[s]
          # if new shortest, update location
          optSpot = s

# number of pairs in first round
n = 7
# first location
spot = 0

# spot is first location
# optSpot +1 will be how many spots the second
# location is from spot
while (optSpot – n >= 0)
     optSpot = optSpot – n
     n = n – 1
     spot = spot +1

# this is the optimal location
return spot, (spot + 1 + optSpot)
```