Colleen Minor
HW 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.

Minimum spanning tree:
{(A, E), (E, B), (B, C), (C, D), (C, G), (G, F), (D, H)}

Graph weight:
A—>E = 4 +
E—> B = 3 +
B—> C = 6 +
C—> D = 1 +
C—> G = 9 +
G—> F = 7 +
D—> H = 12
= graph weight 42.

(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.

a—> e = 4

a —> b = 5

a—>c = MIN([a—>g(15) + g—>c(9) = 24], [a—>b(5) + b—>c(6) = 11]) = 11

a—>d = min(a—>c(11) + c—>d(1)] = 12], [a—>h(28) + h—>d(12) = 40] = 12

a —> f = MIN([a—>e(4) + e—>f(11) =15], [a—>b(5) +b—>f(11) = 16]) = 15

a—>g = MIN([a—>f(11) +f—>g(7) = 18], [a—>b(5) + a—>b—>g(10) = 15], [a—>c(11) + c—>g(9) = 20]) = 15

a—>h = MIN([a—>g(15) + g—>h(15) = 30], [a—>c (11), c—>h(17) = 28], [a—>d(12) + d—>h(12) =12]) = 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.

One thing that we know is that a if a graph is a DAG. I believe that this would be a bunch of dots with arrows pointing to other dots, and if you follow the arrows from dot-to-dot, you would find that they would never go in a complete circle. For example, a DAG would not include “a—>b—>c—>a” because following those arrows will allow you to go in a complete circle from a-to-a.

Then, a Hamilton path is a single path that visits each vertex exactly one time. So if the only dots on my DAG were a, b, & c, and it was directed like, “a—>b—>c“ there we have a Hamilton path by just following from a to c. But if the graph was like, “a<—b—>c” there is no Hamilton path.

So now we want to write an algorithm to find out if a DAG has a Hamilton path, and the runtime should be the number of vertices + the number of edges.

Since we are talking about a DAG, we know that we can sort it with a topological sort:

https://en.wikipedia.org/wiki/Topological_sorting

I think that this is a sort where the idea is, if one vertex leads to another, that vertex MUST come before the vertex that it leads to. So for our a—>b<—c graph, maybe that would go “a, c, b.” Now, one issue, how do we know if there is a Hamilton path… well, we look at each vertex to make sure that there is a connecting edge to the successive vertex. So now that we have our a—>b<—c graph sorted like “a, c, b” all we have to do is see that there is no path from a to c to know that there is not a Hamilton path. So what we will do here is just do a topological sort (which runs in O(V+E)) and then we’ll just iterate through the list and make sure that everything contains a path to the vertex that’s next in the list. We only have to make sure that it contains a path to the next one, because, with basic logic, if at the end of the iteration we found that every vertex contained a path to the next vertex, then the obviously the first vertex contains a path to the last vertex by doing following that path. And of course, the same logic applies to every vertex that’s ordered before another vertex.

Pseudocode:

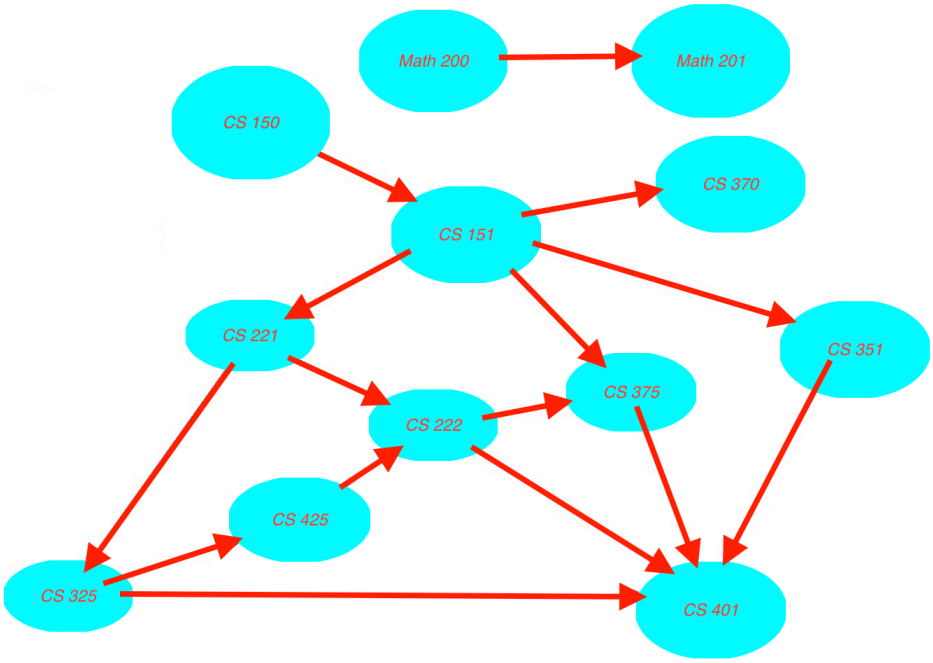
```
//Use a topologicalSort algorithm, which runs in O(V+E) to sort the list

  linkedList<—topologicalSort(graph)

  for x in linkedList

         if x contains path to x+1
            continue
         else
            return false //No Hamilton path

//If you got thru the whole list, and each contains a link to the next, there
is a Hamilton path.
return true
```

Explanation of running time:

As we specified, the topologicalSort will run in O(V+E), and then we run through the list once (which will run in O(V) time). Because we drop coefficients (we won't put O(2V), since that won't be significant eventually) we can say that this algorithm has a run time of O(V+E).

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

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



(b) Give a topological sort of the graph.

CS 150 —> CS 151 —> CS 351—> CS 221 —> CS 325—> CS 222 —> CS 425—> CS 375 —> CS 401 —> CS 370 —>Math 200 —> Math 201

(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.

First semester: CS 150, Math 200

Second semester: CS 151, Math 201

Third semsester: CS 370, CS 351, CS 221

Fourth semester: CS 222, CS 325

Fifth semester: CS 375, CS 425

Sixth semester: CS 401

(d) Determine the length of the longest path in the DAG. How did you find it? What does this represent?

The longest path in the DAG is the path from CS 150 to CS 401, which has a length of 5. This is found by counting the number of edges from circle-to-circle and seeing that the path (CS 150 -> CS 151 -> CS 221 -> CS 222 -> CS 375 -> CS 401) was the longest. It represents path to the course that requires you take the highest number of pre-reqs before taking it.

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.

Verbal: We will do a breadth-first search and alternate coloring neighbor vertices. We will begin at a source vertex, S, and color it red. Then we will color it's neighboring vertices blue. Then we will move to one of those blue vertices and color all of its neighbors red… If we ever find that the neighbors of a vertex are already colored the same color that we just assigned that vertex, we will know that the graph is not two-colorable.

This is a modified version of the code on page 595 in the text, which uses a first-in, first-out queue to get to all of the vertices.

Pseudocode:

```
S.color = red //Color source vertex, S, red

add S to the first-in-first-out queue, Q
while Q is not empty{
//get v by dequeuing a vertex from Q, fist-in-first-out style:
      v <— DEQUEUE(Q)

      for each vertex u that's connected to v{
                        if v.color is the same as u.color{
                              return false //cannot be two-colored
                        }
                        if v.color is red{
                              u.color<—blue

                        }
                        if v.color is blue{
                              u.color<—red
```

```
                    }
                    else { //if v is uncolored
                         v.color<—red
                }

           }
      return true //graph can be two-colored

}
```

(b) Analyze the running time.
The operations for enquiring and dequeuing take O(1) time, and so the total time devoted to queue operations is O(V). This procedure scans the adjacent list of each vertex only when the vertex is dequeued, so it scans each adjacency list at most one time, so that's O(E). Adding this together, the running time of this algorithm 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.

I would recommend Dijkstra's algorithm, because it is more efficient than Bellman Ford's, and there are no negative edges to deal with.

First, lets look at the pseudocode of Dijkstra's:

```
 1  function Dijkstra(Graph, source):
 2
 3      create vertex set Q
 4
 5      for each vertex v in Graph:             // Initialization
 6          dist[v] ← INFINITY                  // Unknown distance from source to v
 7          prev[v] ← UNDEFINED                 // Previous node in optimal path from
source
 8          add v to Q                          // All nodes initially in Q (unvisited
nodes)
 9
10      dist[source] ← 0                        // Distance from source to source
11
12      while Q is not empty:
13          u ← vertex in Q with min dist[u]    // Source node will be selected first
14          remove u from Q
15
16          for each neighbor v of u:           // where v is still in Q.
17              alt ← dist[u] + length(u, v)
```

```
18              if alt < dist[v]:               // A shorter path to v has been found
19                  dist[v] ← alt
20                  prev[v] ← u
21
22      return dist[], prev[]
```

This would unfold live (with G as the source) like this:

```
Line 2:
      Q = {A, B, C, D, E, F, G, H} // all vertices and their distances
Lines 5-7:
      dist[A, B, C, D, E, F, H] = infinity, and prev[A, B, C, D, E, F,
H] = undefined
Line 10:
            dist[G] = 0
Line 11:
      Q = {all the vertices and their distances}
Lines 10-20:
      while loop iteration 1:
            u = G
            remove G from Q
            for neighbor D:
                  alt = 0 + 7
                  dist[D] = 7
                  prev[D] = G
            for neighbor C:
                  alt = 0 + 9
                  dist[C] = 9
                  prev[C] = G
            for neighbor E:
                  alt = 0 + 2
                  dist[E] = 2
                  prev[E] = G
            for neighbor H:
                  alt = 0 + 3
                  dist[H] = 3
                  prev[H] = G
            for neighbor F:
                  alt = 0 + 8
                  dist[F] = 8
                  prev[F] = G



      while loop iteration 2:
            u = E
            for neighbor B:
                  alt = 2 + 9
                  dist[B] = 11
                  prev[B] = E
```

```
      for neighbor H:
            alt = 2 + 7
            dist[H] = still 3 (unchanged)
      for neighbor D:
            alt = 2 + 3
            dist[D] = 5
            prev[D] = E
while loop iteration 3:
      u = H
      for neighbor E:
            alt = 3 + 7
            dist[E] = still 2 (unchanged)
      for neighbor B:
            alt = 3 + 3
            dist[B] = 6
            prev[B]= H
while loop iteration 4:
      u = D
      for neighbor E:
            alt = 5 + 3
            E = still 8 (unchanged)
      for neighbor C:
            alt = 5 + 3
            dist[C] = 8
            prev[C] = E
while loop iteration 5:
      u = B
      for neighbor E:
            alt = 6 + 9
            dist[E] = still 2 (unchanged)
      for neighbor H:
            alt = 6 + 3
            dist[H] = still 3 (unchanged)
while loop iteration 6:
      u = F
      for neighbor C:
            alt = 8 + 2
            dist[C] = still 8 (unchanged)
      for neighbor A:
            alt = 8 + 7
            dist[A] = 15
            prev[A] = F
while loop iteration 7:
      u = C
      for neighbor A:
            alt = 8 + 4
            dist[A] = 12
            prev[A] = C
      for neighbor F:
```

```
            alt = 8 + 2
            dist[F] = still 8 (unchanged)
      for neighbor D:
            alt = 8 + 3
            dist[D] = still 5 (unchangeD)
while loop iteration 8:
      u = A
      for neighbor C:
            alt = 12 + 4
            dist[C] = still 8 (unchanged)
      for neighbor F:
            alt = 12 + 7
            dist[F] = still 8 (unchanged)
```

At this point, the queue should be empty and we should have assigned all of the vertices by their shortest path to the fire-station at G. The results are:

dist[A] = 12
dist[B] = 6
dist[C] = 8
dist[D] = 5
dist[E] = 2
dist[F] = 8
dist[G] = 0
dist[H] = 3

(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.

```
optimalFireSationFinder(graph)
        maxDistance = infinity
        for each f in graph: //for each location…


                //Here we use dijkstra's to return an array of the distances from each node to
                // the current f
                arrayOfDistances <— dijkstra(graph, f)
                //Set the max distance to whatever the highest distance in arrayOfDistances is
                f.maxDistance = max(arrayOfDistances.dist)
                if f.maxDistance < maxDistance:
                        maxDistance = f.maxDistance
                        bestStartingLocation = f
                return f
```

Time complexity analysis:
The time complexity of this algorithm is O(f^3), because it calls Dijkstra's algorithm f times, and Dijkstra's algorithm is O(f^2).

(c) In the above graph what is the "optimal" location to place the fire station? Why?

I ran Dijkstra's algorithm on all 8 of the vertices, and I found that the most optimal fire-station is location e, which has a maximum distance of just 10, from e—>a.

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.*

How I did this was to first create an iteration through the list of locations, where for each index I call Dijkstra's algorithm to get a list of the minimum distances of that location to each other location. I call this matrix "matrixOfDistances1." Then I start a second iteration, and once again I call Dijkstra's on each of index, so that I can get a list of the minimum distances for that location in a matrix called "matrixOfDistances2." Then I compare the values in each index in matrixOfDistances1 and matrixOfDistances2 with each other, putting the minimum of the two in a third matrix, matrix3. If matrix3 has the **lowest maximum** firestation-to-node distance that we've seen so far, maxDistance is set to that distance, and a little matrix called bestLocations is set to [location1, location2].

```
optimalFireSationFinder(graph)
	maxDistance = infinity //Stores the lowest maximum distance for 2 stations
	bestLocations = [null, null] //Stores the best 2 locations
	for each location as location1 { //for each location…

//Here we use dijkstra's to return an array of the distances from each node to
// the current location1.
		matrixOfDistances1 <— dijkstra(graph, location1)

		//Then we try pair with location2…
		for each location as location2 {
			matrixOfDistances2 <— dijkstra(graph, location2)

	 //Then we create a matrix of the lowest distances for if we used both location
			 for each location as i{
			     matrix3[i] = MAX(matrixOfDistances1[i], matrixOfDistances2[i])

		}
		/*Now get the highest value in matrix 3 for the maximum distance with
those 2 locations and compare it to our current maxDistances*/
		if max(matrix3) < maxDistance{
			maxDistance = max(matrix3)
			bestLocations = [location1, location2]
		}


	}
}
```

```
return bestLocations //return the best 2 locations
```

Time complexity analysis:

In opening loop, we call Dijkstra's algorithm n times. Dijkstra's algorithm runs in O(n^2) time, so that's O(n^3).

Then inside that loop, we run another iteration n times— running it a total of n^2 times— and that loop also calls Dijkstra's. So that's O(n^4).

Then inside of that third loop, we run another iteration n times, so that runs in of O(n^3) time.

Therefor, f(n) = O(n^3) + O(n^4) + O(n^3).