HW 5

# 1.

A. (A->E) (E->B) (B->C) (C->D) (C->G) (G->F) (D->H)

B.

| Visited | Distance | |
|---|---|---|
| A | \|A\| | |
| A->E | \|E\| | 4 |
| A->B | \|B\| | 5 |
| A->B->C | \|C\| | 11 |
| A->B->C->D | \|D\| | 12 |
| A->E->F | \|F\| | 15 |
| A->B->G | \|G\| | 15 |
| A->B->C->D->H | \|H\| | 24 |

# 2.

First we would topologically sort the vertices of the DAG. This is done by using a depth first search from some arbitrary vertex. The reverse order of the values removed from the array or stack of vertices when fully searched gives a topological ordering. Second we iterate over the ordered values checking is sequential vertices share an edge on the DAG.If they do share an edge then there exists a hamiltonian path.

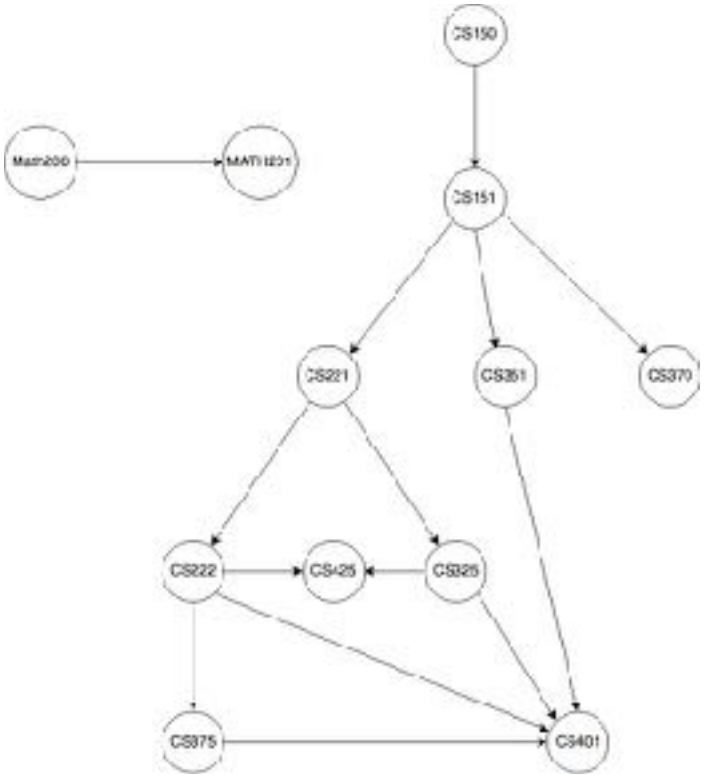
The algorithm works because it tests whether or not the graph can be traversed through all values without going backwards in the graph. While iterating through the vertices in topological order we are essentially walking a path in the graph if all sequential vertices are connected in the graph then the hamiltonian path exists but if there is a vertices that is not connected to a subsequent vertices then you would need to walk backward which violates the definition of a hamiltonian path.

The running time would = Theta(V+E) for v = vertices and E = edges. This is because we iterate through all vertices and iterate through all edges between vertices in a topological sort of DAG. Comparisons are made on each edge and vertices but they are only iterated through once. Therefore time is linear for both V and E and the total runtime is Theta(V+E)

Pseudocode:

```
def  topoSort(graph[],vertex){

        for all nodes in graph
                node<-unmarked

        while there are unmarked nodes
                select an arbitrary unmarked node

                if node has a permanent mark then return
                if node has a temporary mark then stop
                if node is not marked then
                        mark n temporarily
                        for each node x with an edge from x to y do
                                topoSort(x)
                                mark n permanently
                                 add n to head of sortedList
def     isPath(graph){

                topoSort(graph)

                while ( ! end of sorted)

                if ( ! adjacent{(sorted[vertex_1)(sorted[vertex_2])})


                        return false
                else
                        return true

}
```

3. A.




B. math200, math201, cs150, cs151, cs221, cs351, cs370, cs325, cs222, cs375, cs425, cs401

C. Term 1 : (cs150, math200)
Term 2: (cs151, math201)
Term 3: (cs221, cs351, cs370)
Term 4: (cs222, cs325)
Term 5: (cs375, cs425)
Term 6: (cs401)

D.

Longest Path: 5

Finding the longest path requires the DAG be sorted in topological order. Then the sorted list of vertices is iterated  with comparisons of the longest path of connected vertices from the start to the finish. It compares adjacent vertices of vertices with degree > 2 and finds the vertex with the maximum path length and sets that length as the longest path.

The longest path for this DAG represents the greatest path of prerequisites required for the most advanced course. In this case that would be CS401 which requires a path of prerequisites that is at it's greatest 5 classes long with cs150, cs151,cs221,cs222,cs375. Essentially this represents the greatest number of prerequisites required for the most advanced course.

# 4.

A. The algorithm would rely on a breadth first search to check all nodes of the graph. It starts at an initial arbitrary node and assigns it an arbitrary color say blue. It then traverses the graph and as it encounters uncolored nodes it sets the node to the appropriate opposite color of the previous node. If the search finds a node that is colored the same color as the previous adjacent node then the graphical not be two-colored and is therefore not bipartite.

Pseudocode:

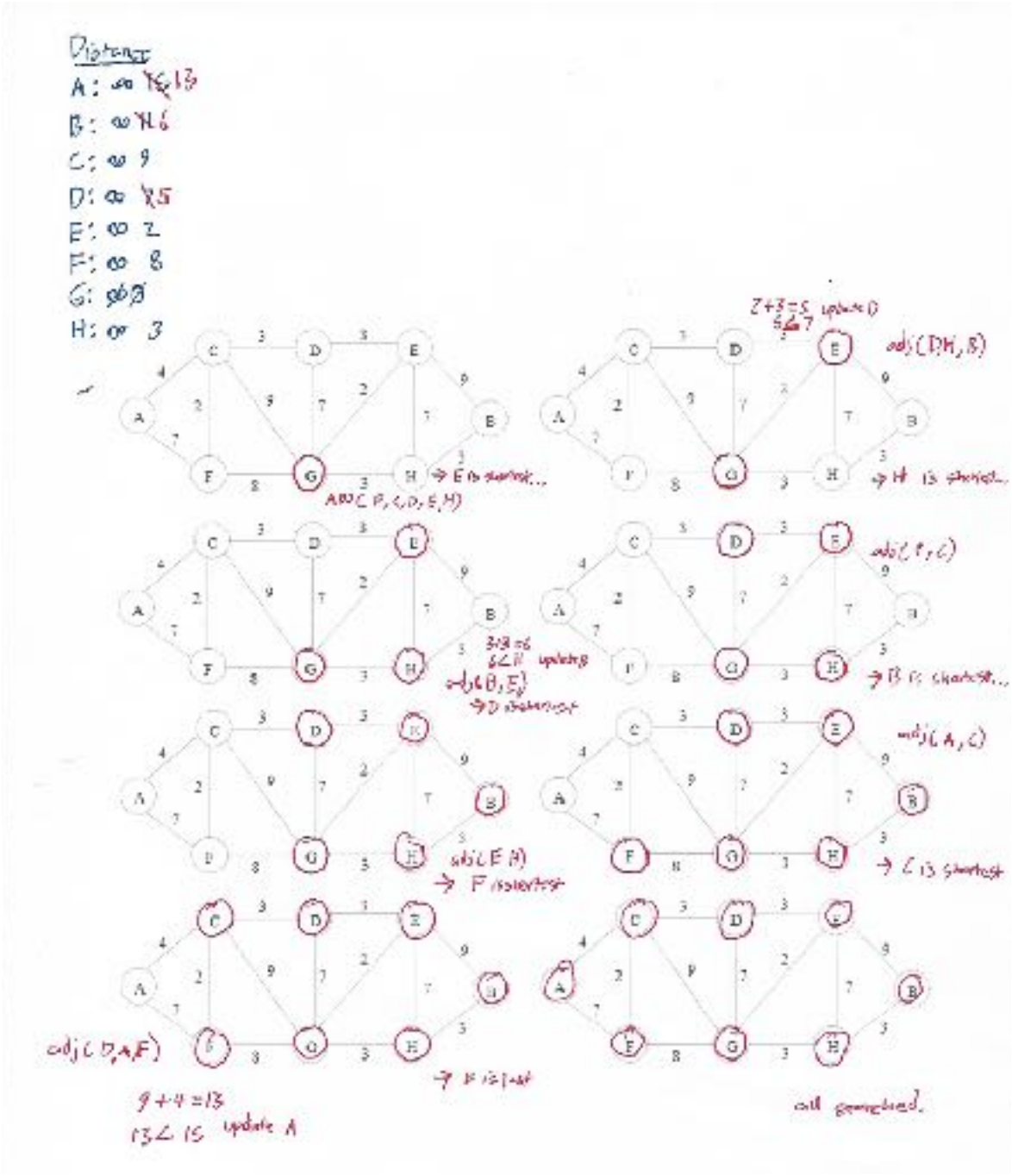
```
for all nodes in graph
        node<-not-colored

initial node<-color1

declare a queue
queue.push(initialNode)

while(queue is not empty)
        temp node = queue.front
        q.pop()
        for all not-colored adjacent vertices of temp node
        if color of vertices == not-colored
                        node<-color1
                        queue.push(node)
                else
                        if color == color of tempNode
                        return false

        return true;
```

B. Because this algorithm is based on a breadth first search on an adjacency list we can assume the runtime is O(V+E). This is true because we may assume that each

vertex is enqueued and dequeued once and finding the adjacent edges takes at most O(E) time therefore the runtime is bounded by the sum of time of E and V or O(V+E)

# 5.

A. I would use dijkstras algorithm for this problem. This is because dijkstras algorithm finds the shortest path to all vertices from a particular start point. It a destination is not determined the algorithm completes when the shortest path of all vertices has been found.



Shortest Routes:
A: g->c->a
B: g->h->b
C: g->c
D: g->e->d
E: g->e
F: g->f
G:(trivial)
H: g->h

B. The algorithm would be similar to dijkstras algorithm with the exception that it would run dijkstras algorithm once for every node in the graph and would then return starting node that resulted in the most minimal longest path from start node to any other node in the graph. The steps for this algorithm would be as follows.

1. Select a node from a graph.
2. perform dijkstras algorithm to find the shortest path to all other nodes in graph.
3. record the greatest of all the short paths calculated for dijkstras algorithm using that start node.
4. Repeat steps 1-3 for all nodes in the graph comparing the long path from each node. If the long path calculated for that run of dijkstras algorithm is less than the current long path stored set that node to the optimal location.
5. Return the optimal location.

The optimal location will be the location that the fire station will be placed such that the furthest intersection away has been optimally limited.

The runtime for this algorithm would be Theta((V+E)*V) the reason being that dijkstras algorithm must run for every vertices in the graph. Dijkstras algorithm runs at Theta(v+E) because it must iterate through all vertices and also make comparisons for all adjacencies (edges). This would be a significantly slow algorithm on a large graph with a great number of nodes.

C. The optimal location for the fire station would be at node E. This is because the maximum of the shortest paths to each node returned by a running of dijkstras on vertex E would be 10. Which is the path from E to A. This is less than the longest of the paths returned for all other runs of Dijkstras on the graphs other nodes. Therefore E is the optimal fire station location.

EXTRA CREDIT:

The algorithm for this problem would build on the previous algorithm that determines the minimal long path found for every node. This algorithm would have to look at every possible pair of nodes and determine the minimal long path for each pair. The most minimal long path found of all the pairs would yield the pair of locations that would be

optimal for placing two fire stations in a  town. The algorithm would need to iterate through each node performing dijkstras algorithm and then within that iteration for each node consider that node as a pair with all other nodes. This would result in consideration of every possible pair of nodes. This is a brute force algorithm and would likely perform slowly on large data sets.

The time complexity for this algorithm would be O((V+E)V^2). This is because O(V+E) is the runtime of dijkstras algorithm and dijkstras algorithm would need to tun for each node pair. The number of possible node pairs would be V^2 with V being the number of nodes in graph. Therefore the algorithm would be bounded by O((V+E)V^2).

The optimal placement of fire stations for the graph of the town given in problem 5 would be C and H. The longest path to any node from these two fire stations would be 5 which is less than all other longest paths found for every other node pair.