Name: Alexander Miranda
Date: November 1st, 2016
Assignment: Homework 3

1) Consider the weighted graph:

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

1. AE
2. EB
3. BC
4. CD
5. CG
6. GF
7. DH

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.

1. A 0
2. E 4
3. B 5
4. C 11
5. D 12
6. F 15
7. G 15
8. 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) 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.

This algorithm (has_hamiltonian) will topologically sort the graph which is essentially reversing the visit list that results from doing depth first search on the graph. The next step will be to look at each vertex in the list and see that there is at least one edge between each consecutive vertex. The runtime of doing the topological sort will be $O(V + E)$ and the runtime of going through the vertex list created from the topological sort will be $O(V)$, so the overall runtime of the algorithm will be $O(V + E)$ because $O(V + E)$ will dominate over the $O(V)$ term.

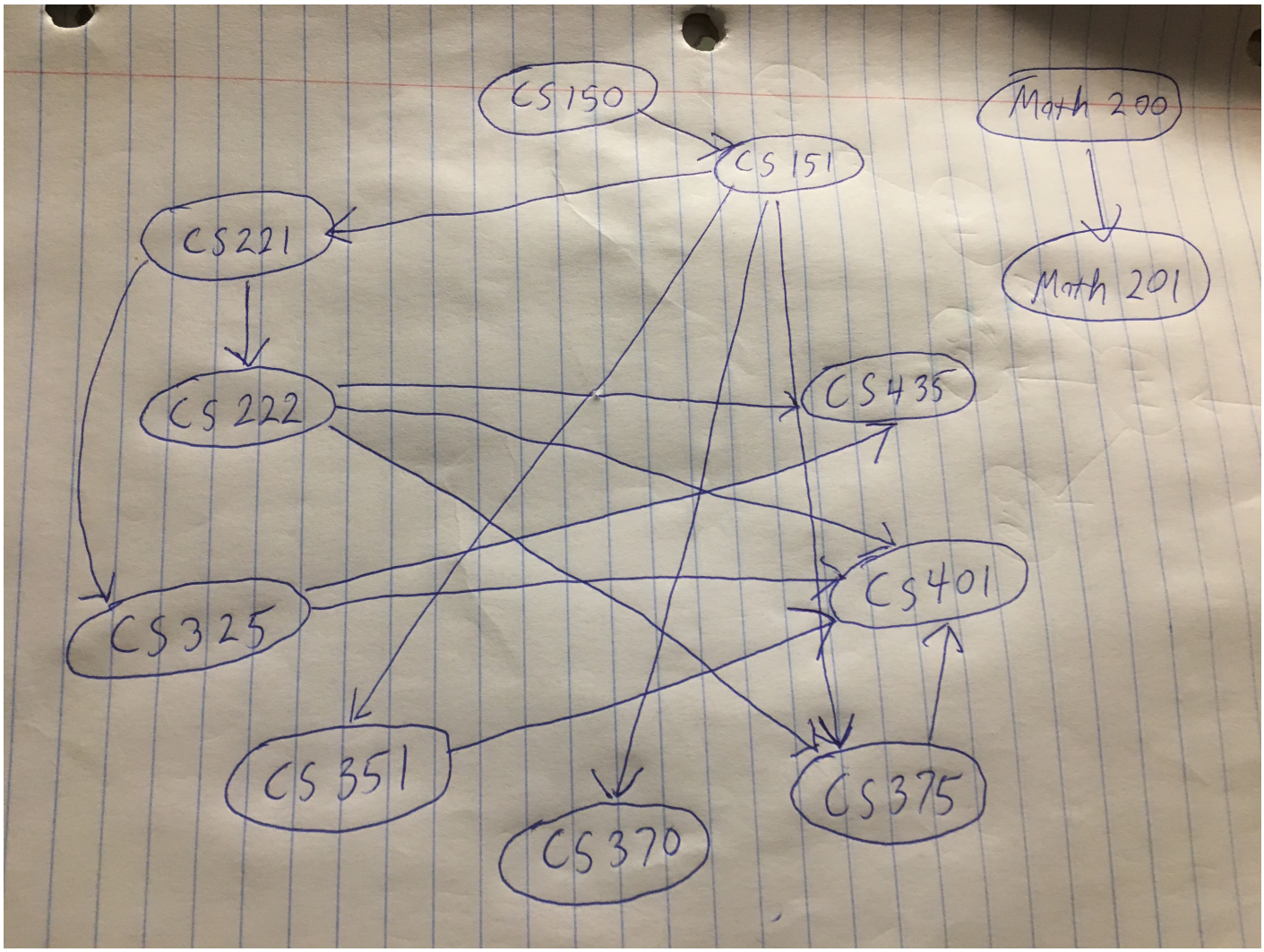
Pseudo-code:

```
# structure of adj_list = {4: [5], 6: [5], 5: [7], 7: []}

def dfs_topsort(graph):         # recursive dfs with
    L = []                      # additional list for order of nodes
    color = { u : "white" for u in graph }
    found_cycle = [False]
    for u in graph:
        if color[u] == "white":
            dfs_visit(graph, u, color, L, found_cycle)
        if found_cycle[0]:
            break

    if found_cycle[0]:          # if there is a cycle,
        L = []                  # then return an empty list

    L.reverse()                 # reverse the list
    return L                    # L contains the topological sort


def dfs_visit(graph, u, color, L, found_cycle):
    if found_cycle[0]:
        return
    color[u] = "gray"
    for v in graph[u]:
        if color[v] == "gray":
            found_cycle[0] = True
            return
        if color[v] == "white":
            dfs_visit(graph, v, color, L, found_cycle)
    color[u] = "black"      # when we're done with u,
    L.append(u)             # add u to list (reverse it later!)

def has_hamiltonian(adj_list):
    graph_sorted = dfs_topsort(adj_list)
    print(graph_sorted)
    for i in range(0, len(graph_sorted) - 1):
        cur_node = graph_sorted[i]
        next_node = graph_sorted[i + 1]
        if next_node not in adj_list[cur_node]:
            return False

    return True
```

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.

Note: CS 435 should have been CS 425



b) Give a topological sort of the graph.

One possible topological sort of the graph is:

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

This sort shows that courses will need to be in an order where a course's prerequisites precede them.

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.

One possible schedule of quarters the courses can be taken in:

Quarter One

- Math 200
- CS 150

Quarter Two

- Math 201
- CS 151

Quarter Three

- CS 221
- CS 351
- CS 370

Quarter Four

- CS 222
- CS 325

Quarter Five

- CS 375
- CS 425

Quarter Six

- 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 for this course graph has a length of 5 which I determined by visually looking over the graph to be the courses as listed:

- CS 150
- CS 151
- CS 221
- CS 222
- CS 375
- CS 401

In order to find the length of the longest path for an arbitrary DAG you would need to sort it topologically and then compute the length of the longest path ending at each vertex. The length of the longest path represents the minimum number of quarters that would need to be attended to finish the CS degree with the given prerequisites.

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.

The algorithm will iterate over all of the "colorless" vertices to then assign them a color that is opposite of the vertices it is adjacent to. If it is discovered that at least two vertices adjacent to the inspected node have both of the colors respectively the algorithm outputs the message that the graph is not two-colorable. Otherwise if the adjacent vertex has color 1, color 2 would be assigned to the inspected vertex and vice versa. If the color for the adjacent vertex is unknown the inspected vertex will be assigned the first color and the adjacent vertex will then be colored. At the conclusion of the algorithm the vertex list will be returned.

```
def main():
    vertice_list = []
    while there are uncolored vertices
        select an uncolored vertex called n
        color(n, L)
    return vertice_list

def color(vertex n):
    if n is colorless:
        for each vertex m with a common edge to n:
            if some m == color1 and some m == color2:
                return "Graph is not two-colorable"
            if m == color1:
                n = color2
            else if m == color2:
                n = color1
            else if m is unknown
                n = color1
                color(m)
        vertice_list.append(n)

main()
```

b) Analyze the running time.

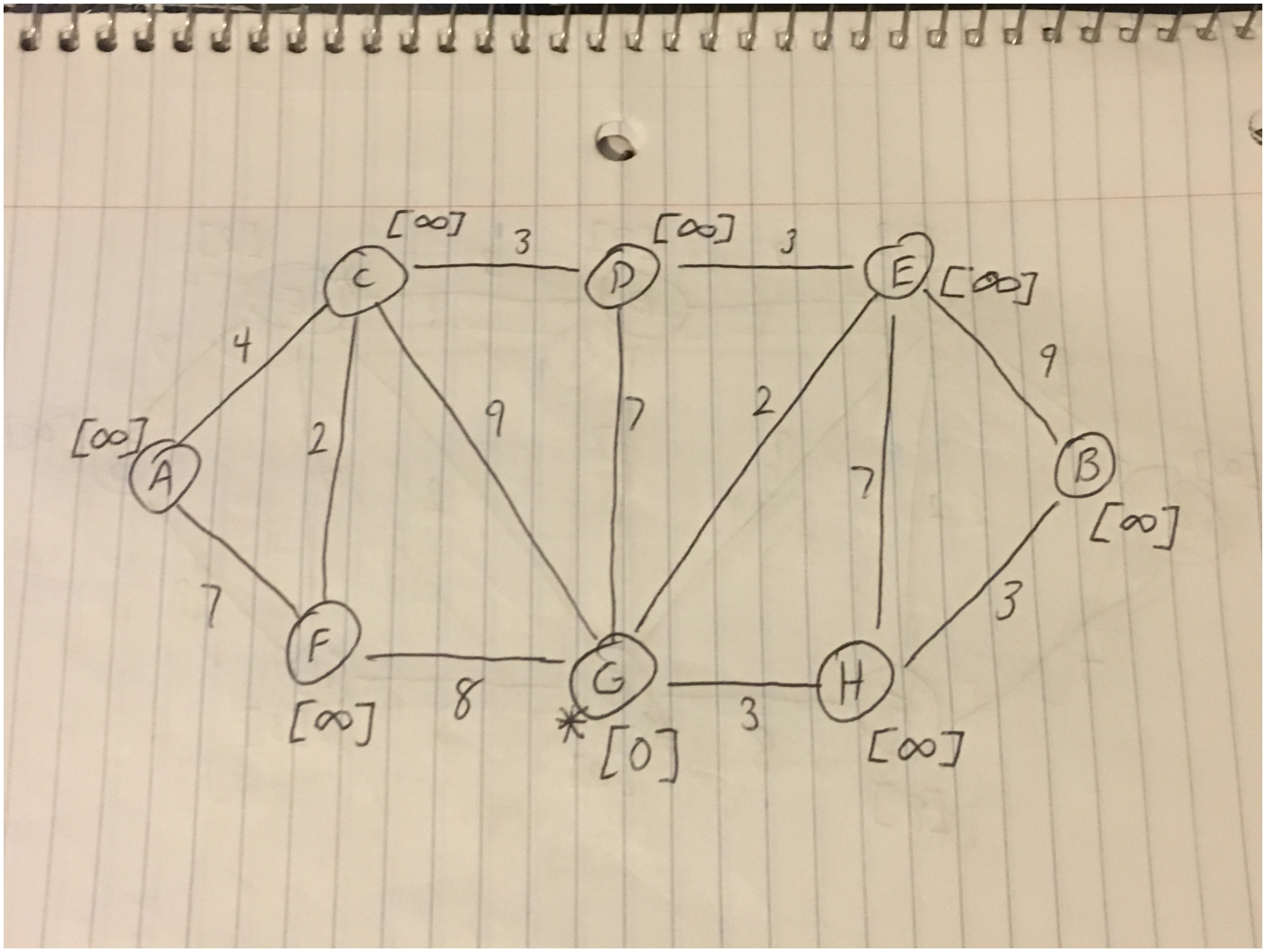
Each vertex and edge are only checked once, and all other operations are in constant time ($O(1)$) therefore the running time of this algorithm will be $O(V + E)$ where V and E are the numbers of vertices and edges respectively.

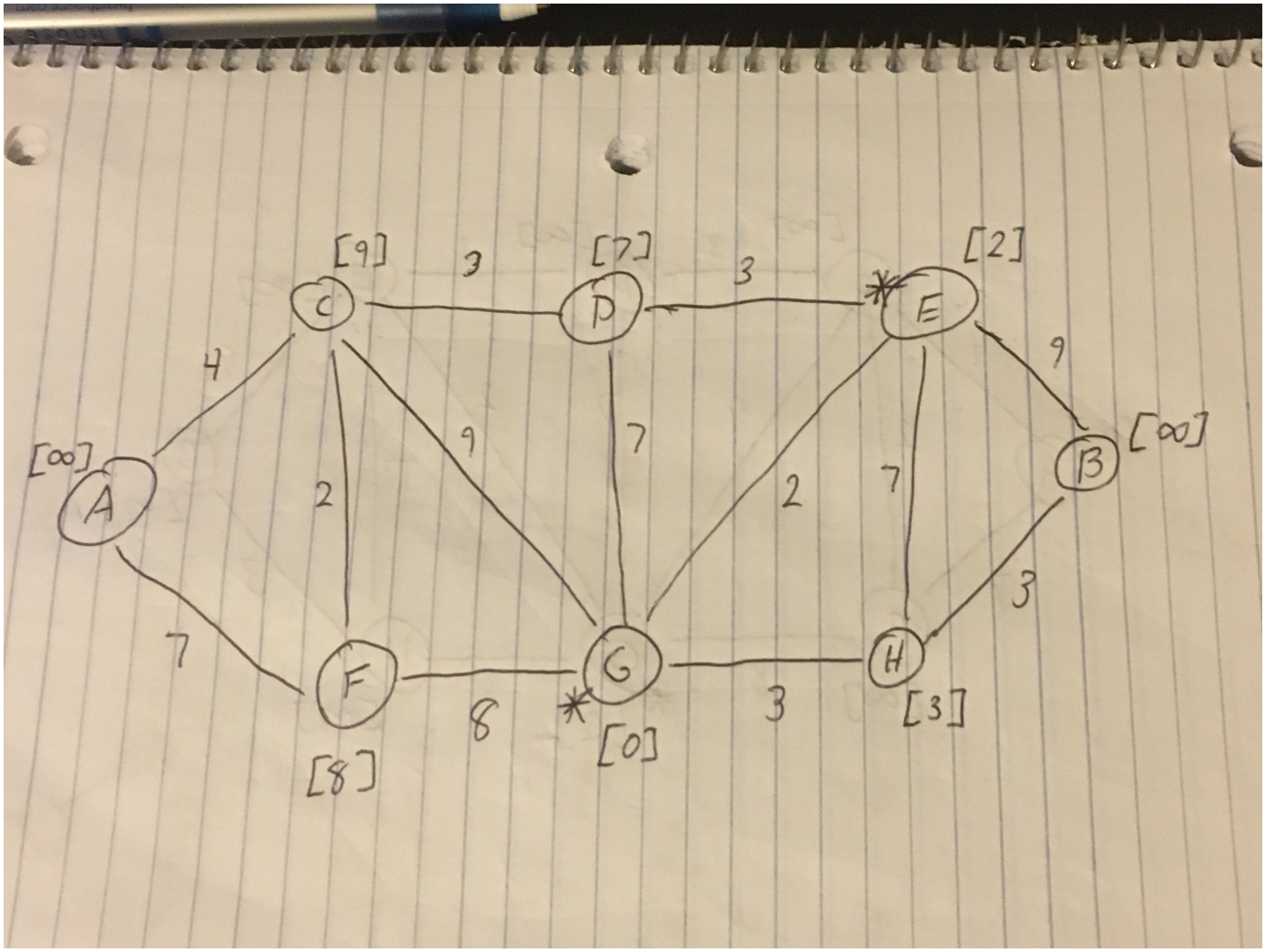
5) A region contains a number of towns connected by roads. Each road is labeled by the average number of minutes required for a fire engine to travel to it. Each intersection is labeled with a circle. Suppose that you work for a city that has decided to place a fire station at location G. (While this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

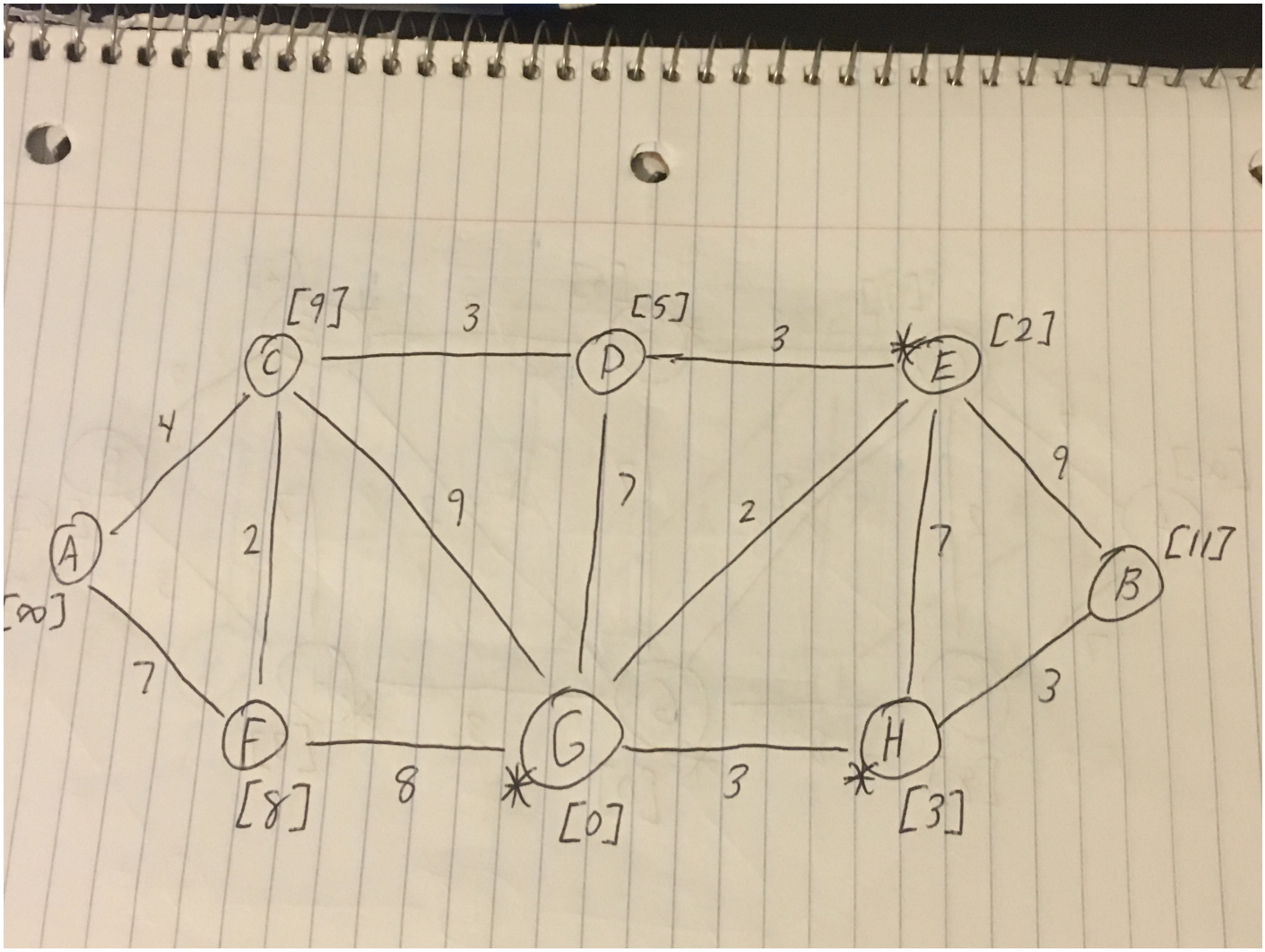
a) What algorithm would you recommend be used to find the fastest route from the fire station to each of the intersections? Demonstrate how it would work on the example above if the fire station is placed at G. Show the resulting routes.

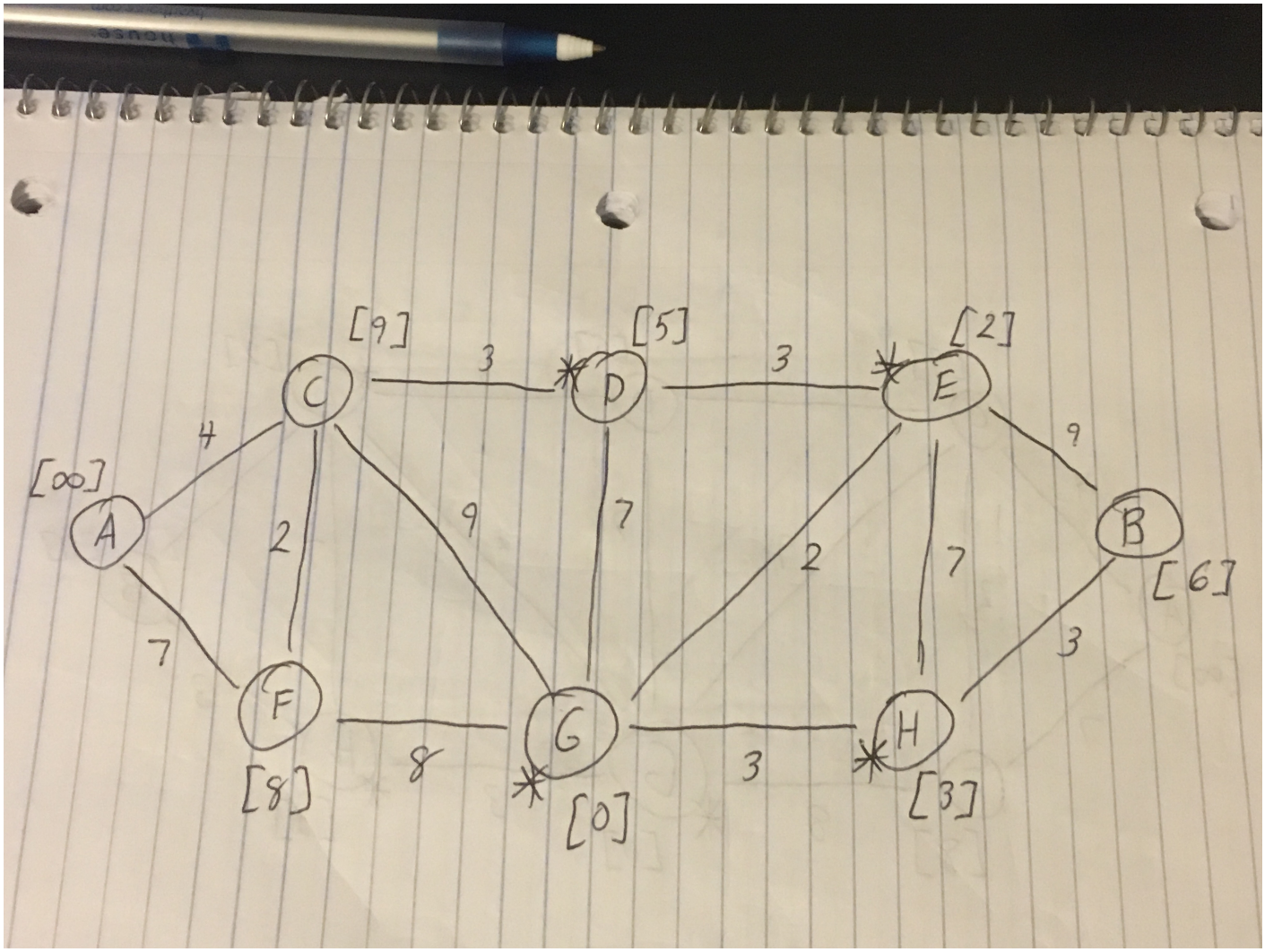
I would use Dijkstra's algorithm to find the fastest route from the fire station location to each of the intersections. The running steps of the algorithm on the graph above is shown in a series of images below:

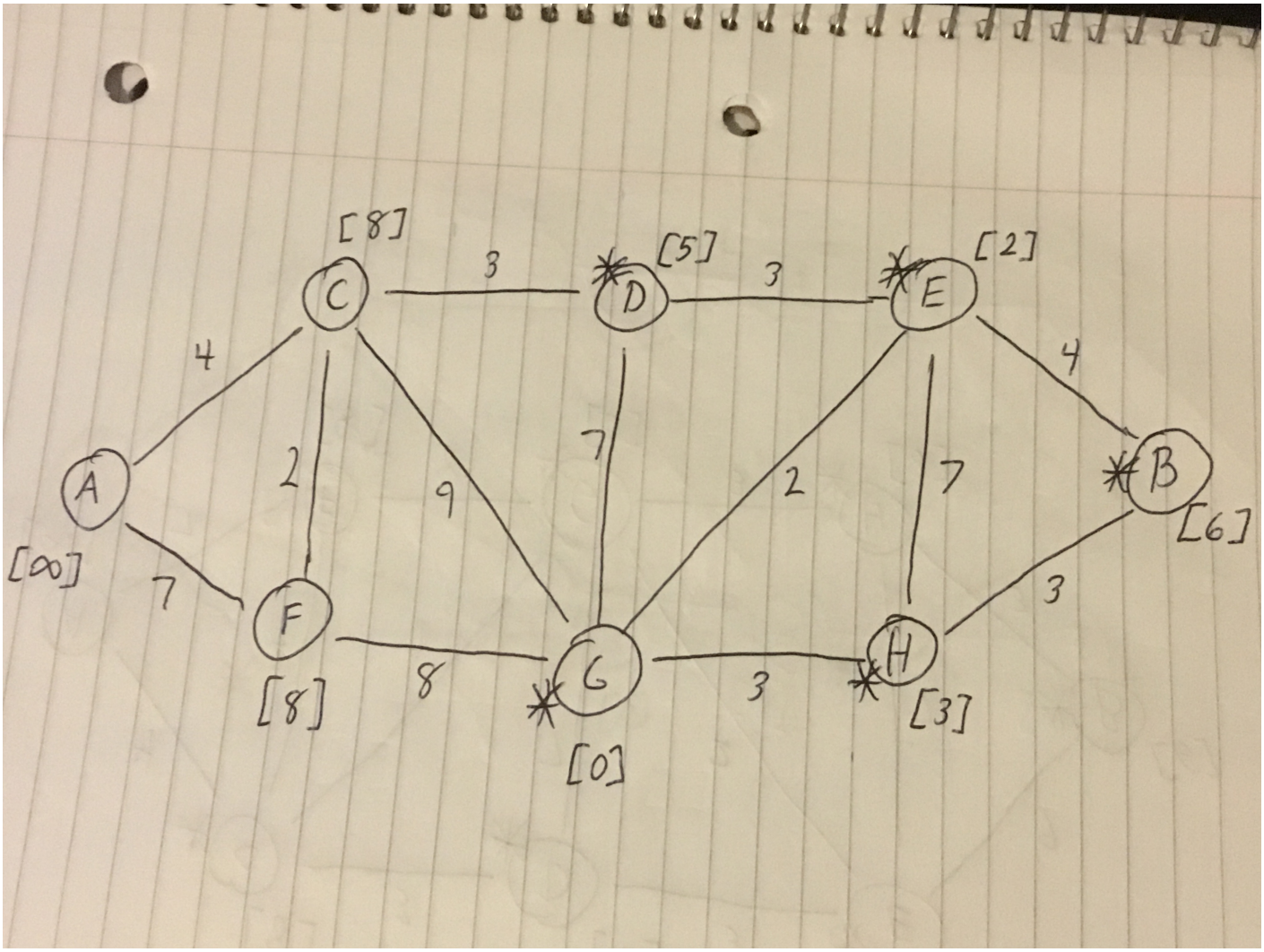
Asterisks denote when a vertex has been visited

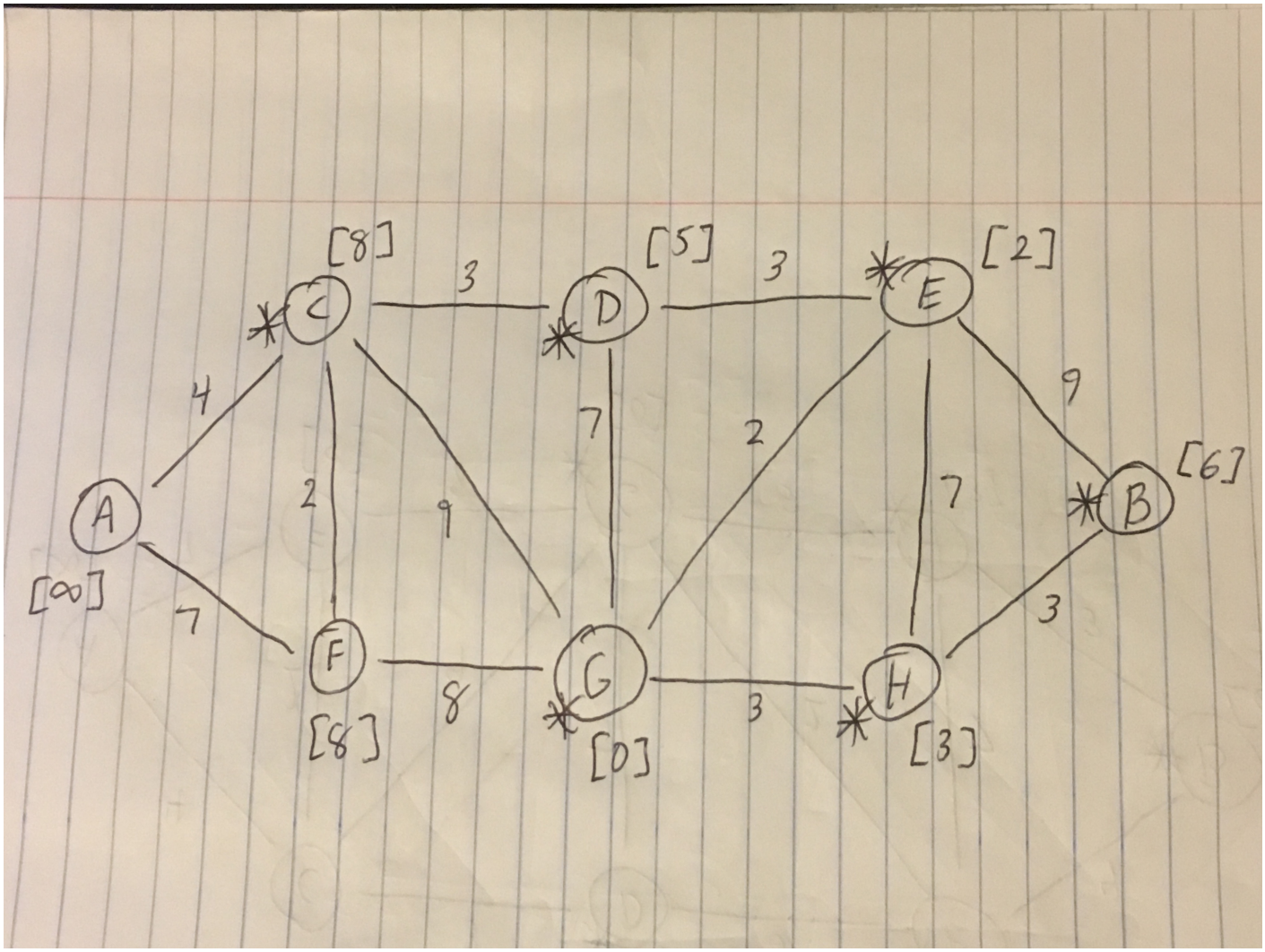

[∞]
3
[∞]
3
C
D
E
[∞]
4
9
[∞]
2
9
7
2
A
7
B
[∞]
7
3
F
G
H
8
3
[∞]
*
[0]
[∞]

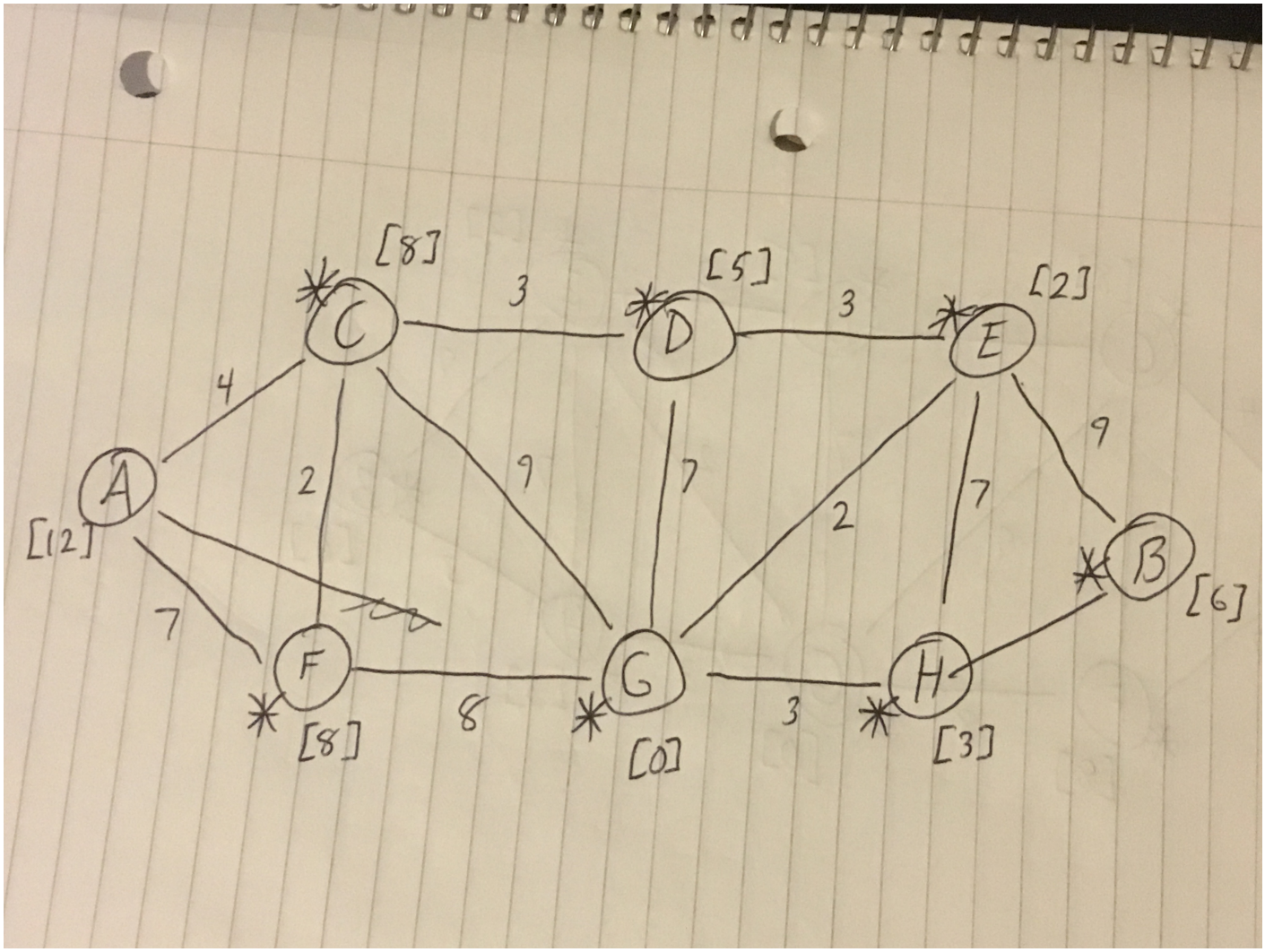

[9]
3
[7]
3
[2]
C
D
E
4
9
[∞]
A
9
7
2
7
2
B
[∞]
3
7
F
8
G
3
H
[3]
[8]
[0]

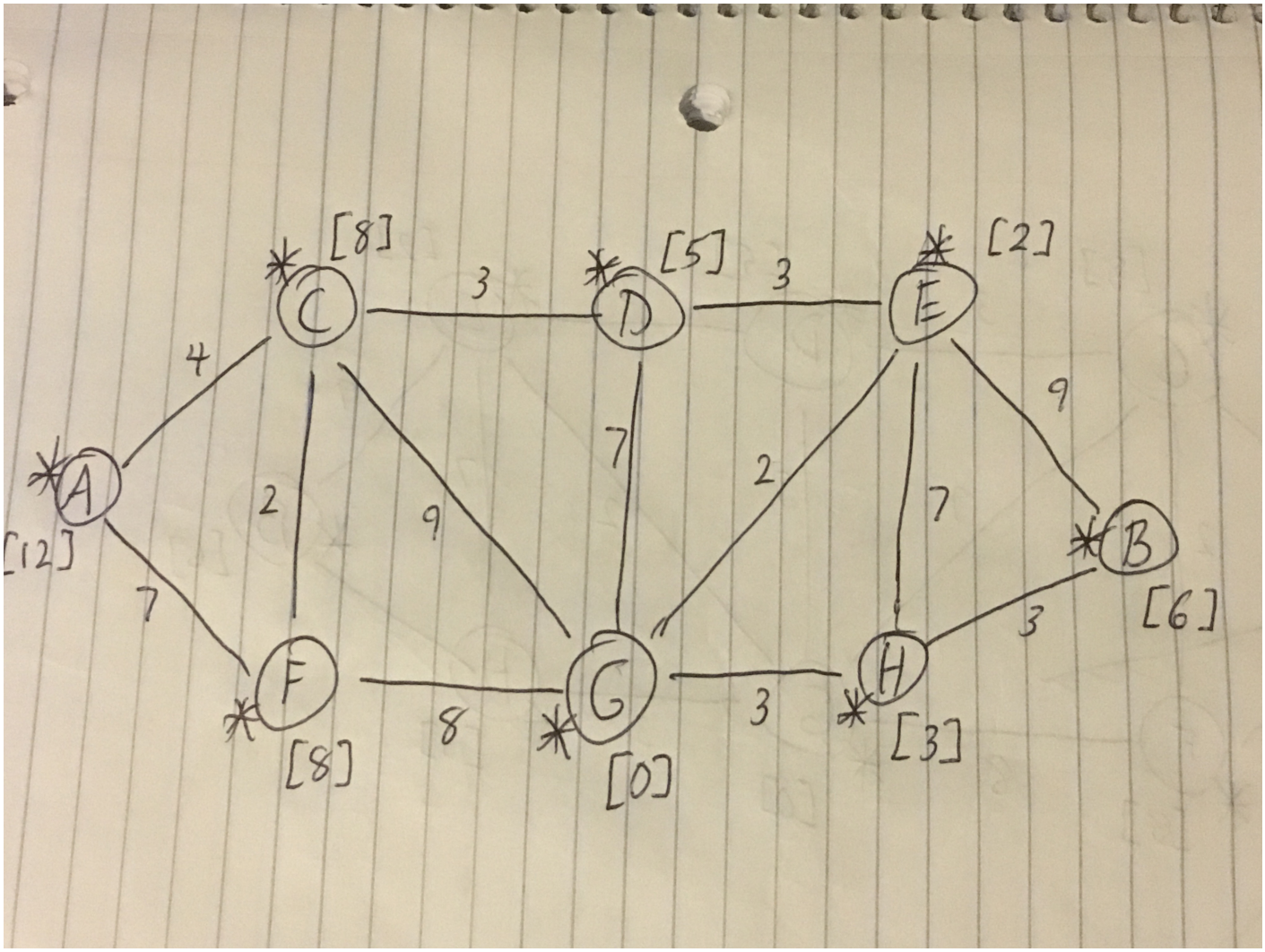

[9]
3
[5]
3
[2]
C
D
E
4
9
7
2
7
9
[11]
A
2
B
[∞]
7
3
F
G
H
[8]
8
[0]
3
[3]


[9]
[5]
[2]
C
D
E
3
3
4
9
[∞]
A
2
9
7
2
7
B
[6]
7
3
F
G
H
8
3
[8]
[0]
[3]


[8]
[5]
[2]
C
D
E
3
3
4
4
A
B
2
9
7
2
7
[∞]
[6]
7
3
F
G
H
8
3
[8]
[3]
[0]


[8]
[5]
[2]
C
D
E
3
3
4
7
2
9
[6]
A
2
9
7
B
[∞]
7
3
F
G
H
8
3
[8]
[0]
[3]


[8]
[5]
[2]
C
D
E
3
3
4
A
[12]
2
9
7
2
7
9
B
[6]
7
F
G
H
8
3
[8]
[0]
[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.

The overview of the algorithm will be to call Dijkstra's algorithm on every vertex and find the smallest max distance that results. The vertex that yields the smallest maximum distance would then be the "optimal" station location. The runtime of Dijkstra's algorithm is $O(r * log(f))$ where r is the same as the graph edges and f is the number of intersections. Dijkstra's algorithm will be run f times therefore the overall runtime of the algorithm to find the optimal location will be: $O(f * r * log(f))$.

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

When running the algorithm on the current intersection graph the optimal location for the fire station would be at intersection E where the smallest longest path distance would be E to A which is a distance of 10. Therefore placing the fire station at E will ensure minimal response times for all locations that need to be protected.