CS532 Homework 10 - Critique
Archana Machireddy

Question 1

I used the method given in heapq page to update my priority queue. I first initialize the priority queue as a list containing 3 elements, the priority, count and the task. Initially priority of source is 0 and for rest of the systems it is infinity. The count initially is 0 for all tasks. Then I take the list from the priority queue using queue.priorityQueue.queue and give it as an input to a new class PriorityQueue_update. The class has methods to add, remove and pop task. The class header has a dictionary (entry_finder) mapping every task to its entry in the priority queue. So while asking it to delete, it just finds this system in O(1) time using this dictionary, and changes the task field in that list to a default value assigned to show that that node has been removed. I have used 1000000000000 to be that value here (just a number higher than the total number of systems). So this list will still be in priority queue but will have [priority, 100000000000] as its value.

The add_task method just adds a new entry with the [new_priority, count, task] to the priority queue using heappush function of heapq. Every time a new task is added the count is increased by 1. This count ensures that elements with equal priority are sorted in the order they were added into the heap. So this is done in O(log n) time. Therefore the entire update procedure takes only O(log n) time. Pop_task method pops only valid tasks, it checks for the default-removed value. So during execution of Dijkstra's instead of checking is the queue is empty we not have to check if all nodes have been processed. When all nodes are processed the size of set S of processed nodes will be equal to length of the priority queue. The Pop_task method also takes O(1), as we are using pushheap during insert, the first few task are always valid tasks. The deleted tasks remain in the rest of the length of the priority queue after all the valid tasks have been popped.

```
class Vertex:
    def __init__(self, identifier: Any):
        self.identifier = identifier
        self.d = float("inf")
        self.pi = None
        self.color = "white"

def euclidean_dist(a: [List[float]],b: [List[float]]) -> float :
    """Method will calculate Euclidean distance

    Arguments:
        a {[List[float]]} -- List containing x,y,z coordinates of system A
        b {[List[float]]} -- List containing x,y,z coordinates of system B

    Returns:
        Float -- Euclidean distance between system A and system B
```

```
    """
    return math.sqrt((b[0]-a[0])**2 + (b[1]-a[1])**2 + (b[2]-a[2])**2)

def parse_universe(fpath=Path("/Users/archana/Dropbox/Algo/HW10/sde-
universe_2018-07-16.csv")
                   ) -> Tuple[List[List[int]], Dict[int, str]]:
    """Method will parse the CSV file and build up a graph representation
of the eve universe used for que 1 and 2

    Keyword Arguments:
        fpath {[type]} -- path to the csv object ot import (default:
{Path("sde-universe_2018-07-16.csv")})

    Returns:
        graph Tuple[List[List[int]] -- An adjacency list reprenting the
graph in the Eve Universe
        name_to_index Dict[int, str] -- A dictionary with keys of indexes
in the adjacency list, and values as the system names
        security_rating_id {Dict[int, float]} -- Security rating for
different systems
        distances {Dict[(int, int),float]} -- Distance between two systems

    """

    # read in csv file build up dict of just system_id to adjacent id_S
    system_mapping = {}
    security_rating = {}
    coordinates = {}
    name_to_id: Dict[str, int] = {}
    with open(fpath) as csvfile:
        reader = csv.DictReader(csvfile)
        for row in reader:
            if int(row["system_id"]) < 31000000:
                name_to_id[row["solarsystem_name"]] = int(row["system_id"])
                if not row["stargates"]:
                    row["stargates"] = "[]"
                system_mapping[int(row["system_id"])] =
list(literal_eval(row["stargates"]))
                security_rating[int(row["system_id"])] = max(0.0,
float(row['security_status']))
                coordinates[int(row["system_id"])] =
list((float(row["x"]),float(row["y"]),float(row["z"])))

    # dictionary referencing system_id to index position
    id_to_index = {system: index for index, system in
enumerate(system_mapping.keys())}

    # constructing list of adjancency-list graph representations
    graph = [None] * len(system_mapping)
    for system, adjacents in system_mapping.items():
```

```python
        graph[id_to_index[system]] = [id_to_index[neighbor] for neighbor in
adjacents]

    # I need to know system names to index for future tracking
    name_to_index = {name: id_to_index[system_id] for name, system_id in
name_to_id.items()}
    security_rating_id = {id_to_index[system_id]: seq for system_id, seq in
security_rating.items()}
    coordinates_id = {id_to_index[system_id]: cord for system_id, cord in
coordinates.items()}

    distances = {}
    for system_index, neighbor_list in enumerate(graph):
        for neighbors in neighbor_list:
            dist = euclidean_dist(coordinates_id[system_index],
coordinates_id[neighbors])
            distances[(system_index,neighbors)] = dist

    return graph, name_to_index, security_rating_id, distances

def backtrace(distances, node: Vertex):
    """Method creates a list of elements that correspond to the order of
progression

        Arguments:
        distances {Dict[(int, int),float]} -- Distance between two vertices
        node {Vertex} -- Vertex to backtrace from

        Returns:
        dist[float] -- Total path distance
        List[int] -- reconstructing the back-pointers
    """
    path = [node.identifier]
    dist = 0
    while node.pi is not None:
        dist = dist + distances[node.pi.identifier,path[0]]
        path.insert(0, node.pi.identifier)
        node = node.pi
    return (dist,path)

class PriorityQueue_update(object):
    def __init__(self, pq):
        self.heap = pq
        self.entry_finder = dict({i[-1]: i for i in pq})
        self.REMOVED = 1000000000000
        self.counter = itertools.count()

    def add_task(self, task, priority=0):
        if task in self.entry_finder:
            self.delete(task)
```

```
        count = next(self.counter)
        entry = [priority, count, task]
        self.entry_finder[task] = entry
        heapq.heappush(self.heap, entry)

    def remove_task(self, task):
        entry = self.entry_finder.pop(task)
        entry[-1] = self.REMOVED

    def pop_task(self):
        while self.heap:
            priority, count , task = heapq.heappop(self.heap)
            if task is not self.REMOVED:
                del self.entry_finder[task]
                return priority, count, task
        raise KeyError('pop from an empty priority queue')


def dijkstra(graph, distances, source: int, destination: int
             ) -> List[int]:
    """Method calculates shortest path from a single source

        Arguments:
        graph {List[List[int]]} -- The adjacensy list representation of the
graph
        distances {Dict[(int, int),float]} -- Distance between two vertices
        source {int} -- The system index of the starting system
        destination {int} -- The system index of the destination system

        Returns:
        List[int] -- The list of system indexes representing the shortest
path from the source to target destination
        """
    # initialization of the nodes
    vertices = [Vertex(index) for index, _ in enumerate(graph)]
    vertices[source].d = 0

    S = set()
    Q = queue.PriorityQueue()

    for index, _ in enumerate(graph):
        Q.put([vertices[index].d, 0, vertices[index].identifier])

    pq = PriorityQueue_update(Q.queue)

    while len(pq.heap) > len(S):
        print(len(pq.heap))
        d,_,u = pq.pop_task()
        S.add(u)
        if u == destination:
```

```python
            return backtrace(distances, vertices[destination])
        for adj_star in graph[u]:
            if vertices[adj_star].d > vertices[u].d + distances[u,
adj_star]:
                vertices[adj_star].d = vertices[u].d + distances[u,
adj_star]
                vertices[adj_star].pi = vertices[u]
                pq.remove_task(adj_star)
                pq.add_task(adj_star,vertices[adj_star].d)


def q1_shortest_path(start: str, destination: str) -> List[str]:
    graph, mapping, security, distances  = parse_universe()
    reverse_map = {index: name for name, index in mapping.items()}
    if start not in mapping.keys():
        print('Source system does not exist')
        return
    if destination not in mapping.keys():
        print('Destination system does not exist')
        return
    startt = timer()
    dist, jita_dodixie_route = dijkstra(graph, distances, mapping[start],
mapping[destination])

    end = timer()
    print('Total time',end-startt)
    print('total distance', dist)
    route = [reverse_map[system] for system in jita_dodixie_route]
    print(route)
    return route

def question1():
    q1_shortest_path("6VDT-H", "Dodixie")
```

```
Archanas-MBP:HW10 archana$ python3 hw10.py
Total time 0.032070404035039246
total distance 5.009603035325423e+17
['6VDT-H', 'B17O-R', 'IGE-RI', 'OW-TPO', 'AL8-V4', 'JGOW-Y', 'APM-6K', '5-D82P', '9-V
O0Q', '3WE-KY', '4-EP12', 'XF-TQL', '7-692B', 'DB-6W4', 'R-OCBA', '3HQC-6', 'JKJ-VJ',
 'J9SH-A', 'LGUZ-1', '4C-B7X', 'F-XWIN', 'D5IW-F', '1L-BHT', 'E9G-MT', 'U09-YG', 'RL-
KT0', 'C0T-77', '3KNK-A', '8V-SJJ', 'K5-JRD', 'X-M2LR', 'FD-MLJ', 'X-BV98', 'Poitot',
 'F67E-Q', 'MHC-R3', 'Harroule', 'Ostingele', 'Stacmon', 'Aidart', 'Cistuvaert', 'Ale
ntene', 'Merolles', 'Tar', 'Pakhshi', 'Renyn', 'Grinacanne', 'Erme', 'Botane', 'Dodix
ie']
```

Question 1 Critique:

I m not explicitly printing it out when there is no path.

```
def q1_shortest_path(start: str, destination: str) -> List[str]:
    graph, mapping, security, distances  = parse_universe()
    reverse_map = {index: name for name, index in mapping.items()}
    if start not in mapping.keys():
        print('Source system does not exist')
        return
    if destination not in mapping.keys():
        print('Destination system does not exist')
        return
    startt = timer()
    dist, jita_dodixie_route = dijkstra(graph, distances, mapping[start],
mapping[destination])
    end = timer()
    print('Total time',end-startt)
    print('total distance', dist)
    route = [reverse_map[system] for system in jita_dodixie_route]
    print(route)
    if dist == 0:
        print('There exists no path from',start,'to', destination)
    return route
```

```
Archanas-MBP:HW10 archana$ python3 hw10_final.py
Total time 0.026118271052837372
total distance 0
['TP7-KE']
There exists no path from U8MM-3 to TP7-KE
Archanas-MBP:HW10 archana$
```

Question 2

Here is the distance term represents the security status. If two systems have the same security, the counter in the PriorityQueue_update method ensures that you are staying on the shortest path. If you do not have this counter, you will get a path but it might not be the shortest. Whenever I find a system with lower security status, I update the distance term to the new security status.

```
def question2():
#    q2_best_path("6VDT-H", "N-RAEL")
    q2_best_path("Egmur", "Javrendei")

def q2_best_path(start: str, destination: str) -> List[str]:
    graph, mapping, security, distances  = parse_universe()
    reverse_map = {index: name for name, index in mapping.items()}
    if start not in mapping.keys():
        print('Source system does not exist')
```

```
        return
    if destination not in mapping.keys():
        print('Destination system does not exist')
        return
    startt = timer()
    final_sec, jita_dodixie_route = dijkstra_2(graph, security,
mapping[start], mapping[destination])
    end = timer()
    print('Total time',end-startt)
    print('total security', final_sec)
    route = [reverse_map[system] for system in jita_dodixie_route]
    print(route)
    return route



def backtrace_2(security: Dict[int, float], node: Vertex) -> (float,
List[int]):
    """Method creates a list of elements that correspond to the order of
progression

        Arguments:
        security {Dict[int, float]} -- Security rating for different
systems
        node {Vertex} -- Vertex to backtrace from

        Returns:
        final_sec[float] -- Total security along the path
        List[int] -- reconstructing the back-pointers
        """
    path = [node.identifier]
    final_sec = security[path[0]]
    while node.pi is not None:
        path.insert(0, node.pi.identifier)
        final_sec = max(final_sec,security[path[0]])
        node = node.pi
    return (final_sec,path)


def dijkstra_2(graph: List[List[int]], security: Dict[int, float], source:
int, destination: int
              ) -> (float, List[int]):
    # initialization of the nodes
    vertices = [Vertex(index) for index, _ in enumerate(graph)]
    vertices[source].d = 0

    S = set()
    Q = queue.PriorityQueue()

    for index, _ in enumerate(graph):
```

```
        Q.put([vertices[index].d, 0,vertices[index].identifier])

    pq = PriorityQueue_update(Q.queue)
    while len(pq.heap) > len(S):
        d,_,u = pq.pop_task()
        S.add(u)
        if u == destination:
            return backtrace_2(security, vertices[destination])
        a = []
        b= []
        for adj_star in graph[u]:
            if adj_star not in S:
                if vertices[adj_star].d > max(vertices[u].d,
security[adj_star]):
                    vertices[adj_star].d = max(vertices[u].d,
security[adj_star])
                    vertices[adj_star].pi = vertices[u]
                    vertices[adj_star].l = len(S)
                    pq.remove_task(adj_star)
                    pq.add_task(adj_star,security[adj_star])
```

The security status of any processed node is the maximum security encountered in the path so far. So a path taken by the processed nodes in S is always the shortest path to reach those nodes, and the score along the path is always the lowest possible maximal security status. As each node gets processed it has the optimal shortest path till that node, and this forms part of the shortest path to the next node. The sub-path of any shortest path is a shortest path, so it has an optimal substructure.

```
Archanas-MBP:HW10 archana$ python3 hw10_que2.py
Total time 0.04496714996639639
total security 0.0
['6VDT-H', 'B17O-R', 'IGE-RI', 'OW-TPO', 'AL8-V4', 'JGOW-Y', 'APM-6K', 'RE
-C26', '00GD-D', 'C-N4OD', 'KVN-36', '4HS-CR', 'WMH-SO', 'LBGI-2', 'Y-2ANO
', 'ZXB-VC', '5-CQDA', 'KEE-N6', '4X0-8B', 'JP4-AA', 'D-W7F0', '4K-TRB', '
QX-LIJ', 'O-IOAI', 'NOL-M9', 'PR-8CA', 'FWST-8', 'YZ9-F6', '319-3D', 'D-3G
IQ', 'K-6K16', 'W-KQPI', 'PUIG-F', '0-HDC8', 'SVM-3K', '8QT-H4', '49-U6U',
 '4-07MU', 'RR-D05', '25S-6P', 'FAT-6P', 'CNC-4V', 'CZK-ZQ', '4NBN-9', 'X4
-WL0', 'Q-U96U', 'EX6-AO', 'HY-RWO', 'V-3YG7', 'B-3QPD', 'U-QVWD', '0SHT-A
', 'D87E-A', 'K-B2D3', 'VOL-MI', 'ARG-3R', '9PX2-F', 'N3-JBX', 'SG-75T', '
XV-MWG', 'Q-K2T7', '1V-LI2', 'LQ-OAI', 'U2-28D', 'PUZ-IO', 'HB-1NJ', 'F7A-
MR', '0-3VW8', '28-QWU', 'N-RAEL']
```

Question 2 Critique:

I was mistaken to say that every sub path is a shortest path. The shortest path between any two intermediary node is a part of the shortest path between outer nodes.

Question 3

I am using breadth first search algorithm to see if the systems are connected. If the distance between two universes is less than the maximum distance then I add it into the queue else I leave it out. Then I take the list of all systems popped out by the queue and compare it with all input systems. If the two sets are equal then the graph is fully connected, else it is not. This method returns True with a max_distance of 1.0e+18, and returns False with a max_distance of 1.0e+16, as there max_distance between any two systems is less than 1.0e+18, but there are some systems at distances greater than 1.0e+16, which do not get included in the connected components list of the BFS.

I use the same parse method to parse the csv file for both question 3 and 4.

```
def parse_universe_4(fpath=Path("/Users/archana/Dropbox/Algo/HW10/sde-
universe_2018-07-16.csv")
                     ) ->  (Dict[int, str], Dict[int,List[float]]):
    """Method will parse the CSV file and build up a graph representation
of the eve universe

        Keyword Arguments:
        fpath {[type]} -- path to the csv object ot import (default:
{Path("sde-universe_2018-07-16.csv")})

        Returns:
        name_to_index Dict[int, str] -- Dictionary with keys of indexes in
the adjacency list, and values as the system names
        coordinates_id Dict[int, List[float]] -- Dictionary with keys of
system indices and values of their x,y,z coordinates
        """

    # read in csv file build up dict of just system_id to adjacent id_S
    system_mapping = {}
    coordinates = {}
    name_to_id: Dict[str, int] = {}
    with open(fpath) as csvfile:
        reader = csv.DictReader(csvfile)
        for row in reader:
            if int(row["system_id"]) < 31000000:
                name_to_id[row["solarsystem_name"]] = int(row["system_id"])
                if not row["stargates"]:
                    row["stargates"] = "[]"
                system_mapping[int(row["system_id"])] =
list(literal_eval(row["stargates"]))
                coordinates[int(row["system_id"])] =
list((float(row["x"]),float(row["y"]),float(row["z"])))

    # dictionary referencing system_id to index position
    id_to_index = {system: index for index, system in
enumerate(system_mapping.keys())}

    # I need to know system names to index for future tracking
```

```
    name_to_index = {name: id_to_index[system_id] for name, system_id in
name_to_id.items()}
    coordinates_id = {id_to_index[system_id]: cord for system_id, cord in
coordinates.items()}

    return name_to_index, coordinates_id


def question_3():
    mapping, coordinates = parse_universe_4()
    reverse_map = {index: name for name, index in mapping.items()}
    systems = [index for name,index in mapping.items()]
    max_distance = 1.0e+16
    print('Max Distance:', max_distance)

    ### Compute adjacency matrix
    distance_matrix = [[euclidean_dist(coordinates[i], coordinates[j]) for
i in systems] for j in systems]
    visited = breadth_first_search(distance_matrix, max_distance,
systems[0])

    if set(systems) == set(visited):
        return True
    else:
        return False

def breadth_first_search(graph: List[List[int]], max_distance: int, source:
int) -> List[int]:
    """Perform BFS on the graph,

        Arguments:
        graph {List[List[int]]} -- The adjacensy list representation of the
graph
        max_distance {int} -- Maximum distance allowed between two systems
        source {int} -- The system index of the starting system

        Returns:
        List[int] -- The list of system indexes representing the shortest
path from the source to all reachable systems
        """
    # initialization of the nodes
    vertices = [Vertex(index) for index, _ in enumerate(graph)]
    vertices[source].color = "gray"
    vertices[source].d = 0


    queue = []
    processed = []
    queue.append(source)
    while queue != []:
```

```python
        u = queue.pop(0)
        processed.append(u)
        for adj_star, distance in enumerate(graph[u]):
            if vertices[adj_star].color == 'white':
                if distance <= max_distance:
                    vertices[adj_star].color = 'gray'
                    vertices[adj_star].d = vertices[u].d + distance
                    vertices[adj_star].pi = vertices[u]
                    queue.append(adj_star)
        vertices[u].color = 'black'
    return processed


def question3():
    answer = question_3()
    print('Final answer: ',answer)
```

```
Archanas-MBP:HW10 archana$ python3 hw10.py
Max Distance: 1e+16
Final answer:  False
Archanas-MBP:HW10 archana$ python3 hw10.py
Max Distance: 1e+18
Final answer:  True
Archanas-MBP:HW10 archana$
```

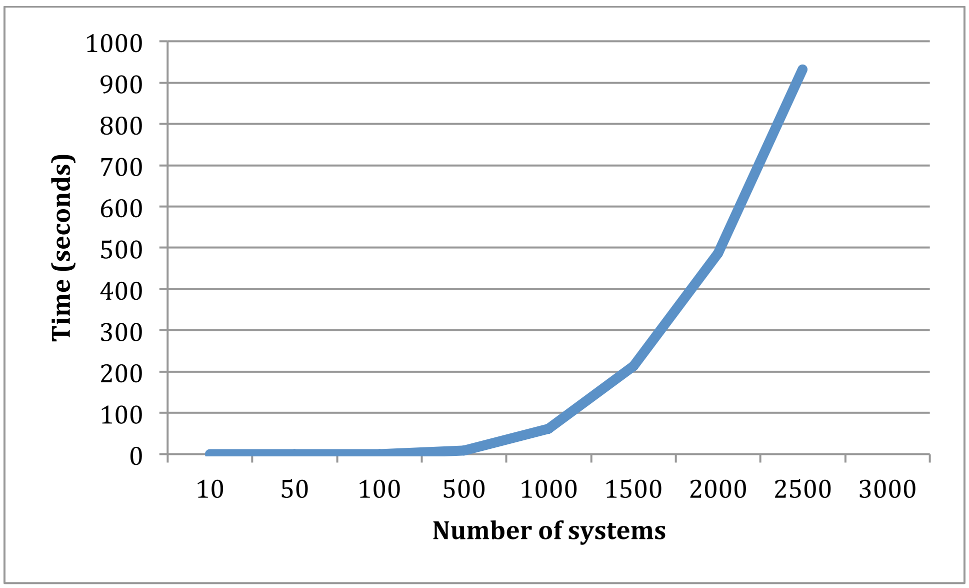
Question 4

I use the prim's algorithm adding the cheapest possible connection to the tree at each step. I start at a node and select the node that has the shortest distance from the start node, and add it to the queue. Then from that newly added node I search for the shortest distance among the nodes that haven't been added to the queue and select the next shortest distance. I repeat this by selecting each node as the start node once.

I couldn't run it completely. Considering source nodes from 0 to 4200, the minimum max_distance I obtained is 2.2139013776749814e+17 between G-M4GK and NRT4-U. From experiments on question 3, the minimum max_distance is in between 2.4675e+16 and 2.468e+16.

The time complexity of the algorithm is $0(V^3)$. The initial adjacency matrix calculation has time complexity $O(V^2)$. As we are using each system as source once the outer loop runs V times. In each run, we go through every system once in the while-loop. Inside the while –loop, min and index are both O(n) operations, so O(2V). Therefore on the whole it has time complexity of $O(V^3)$. This can be seen from the table below, as the constant seems pretty stable after from 500 to 2500 nodes. The constant is calculated by dividing time taken by (number of nodes)^3 and multiplytin the result by 100000000.

Number of nodes	Time	Constant
50	0.015029961	12.02396881
100	0.092646254	9.264625399
500	8.111535138	6.48922811
1000	61.95302052	6.195302052
1500	213.7826758	6.334301505
2000	486.8799357	6.085999196
2500	931.5669918	5.962028748




```
def question4():
    mapping, coordinates = parse_universe_4()
    reverse_map = {index: name for name, index in mapping.items()}
    systems = [index for name,index in mapping.items()]
    startt = timer()
    answer,start,finish = 
question_4(systems[1:2500],coordinates,reverse_map)
    end = timer()
    print('Total time',end-startt)
    print('Final answer 4: ',answer)
    print(reverse_map[start],reverse_map[finish])


def question_4(systems,coordinates,reverse_map):
    ### Compute adjacency matrix
    distance_matrix = [[euclidean_dist(coordinates[i], coordinates[j]) for
```

```
i in systems] for j in systems]
    visited,final_min_dist,start,finish = search4(systems,
distance_matrix,reverse_map)

    if set(systems) == set(visited):
        print(True)
    else:
        print(False)

    print(final_min_dist)
    return final_min_dist,start,finish

def search4(systems, graph: List[List[int]],mapping) -> List[int]:

    final_min_dist = 1.0e+25
    start = 0
    finish = 0

    for i,_ in enumerate(systems):
        source = i
        if source % 100 == 0:
            print('source',source)
        max_dist = 0
        start_in = 0
        finish_in = 0

        vertices = [Vertex(index) for index, _ in enumerate(graph)]
        vertices[source].d = 0

        queue = []
        processed = []
        queue.append(source)
        good_indices = list(range(0,len(systems)))
        while queue != []:
            u = queue.pop(0)
            processed.append(u)
            good_indices.remove(u)
            a = graph[u]
            remaining_nodes = [a[i] for i in good_indices]
            if remaining_nodes != []:
                min_dist = min(remaining_nodes)
                minpos = graph[u].index(min_dist)
                if min_dist > max_dist:
                    max_dist = min_dist
                    start_in = u
                    finish_in = minpos
                queue.append(minpos)

        if max_dist < final_min_dist:
            final_min_dist = max_dist
```

```
            start = start_in
            finish = finish_in
            print(source,final_min_dist,mapping[start],mapping[finish])

    return (processed,final_min_dist,start,finish)
```

```
Archanas-MBP:HW10 archana$ python3 hw10_que4.py
source 0
0 4.945040169251071e+17 W-Z3HW UAV-1E
1 3.3867133919310483e+17 UC-X28 GTB-04
6 2.937094682710582e+17 1-10QG M-XUZZ
18 2.7929736721636954e+17 UC-X28 XKM-DE
source 100
122 2.764032087669391e+17 M-ZJWJ UAV-1E
source 200
source 300
326 2.5360319789789315e+17 M-ZJWJ Skarkon
source 400
474 2.3919030974927622e+17 Ordion M0O-JG
source 500
source 600
624 2.2139013776749814e+17 G-M4GK NRT4-U
source 700
```

Question 4 Critique:

Using Dijkstra's algorithm is a much faster way to obtain the result.

```
def question_4(systems,coordinates,reverse_map):

    ### Compute adjacency matrix
    distance_matrix = [[euclidean_dist(coordinates[i], coordinates[j]) for
i in systems] for j in systems]
    visited, vertices = dijkstra_4(systems, distance_matrix,reverse_map)

    if set(systems) == set(visited):
        print(True)
    else:
        print(False)
```

```
    final_min_dist = 0
    for v,v_d in enumerate(vertices):
        if vertices[v].d > final_min_dist:
            final_min_dist = vertices[v].d
            longestv = vertices[v].pi
    print("Longest distance that must be traveled to reach all stars is
%11.4e" % final_min_dist)

    v = longestv.identifier
    while vertices[v].pi != None:
        u = vertices[v].pi.identifier
        if distance_matrix[u][v] == final_min_dist:
            print("Longest distance is between %s and %s"
%(reverse_map[u],reverse_map[v]))
            start = u
            finish = v
        v=u
    return final_min_dist,start,finish


def dijkstra_4(systems, graph, mapping):
    # initialization of the nodes
    source = 0
    vertices = [Vertex(index) for index, _ in enumerate(graph)]
    vertices[source].d = 0

    S = set()
    Q = queue.PriorityQueue()

    for index, _ in enumerate(graph):
        Q.put([vertices[index].d, 0,vertices[index].identifier])

    pq = PriorityQueue_update(Q.queue)
    while len(S) < len(systems):
        d,_,u = pq.pop_task()
        S.add(u)
        for adj_star in systems:
            if adj_star not in S:
                if vertices[adj_star].d > max(vertices[u].d,
graph[u][adj_star]):
                    vertices[adj_star].d = max(vertices[u].d,
graph[u][adj_star])
                    vertices[adj_star].pi = vertices[u]
                    pq.remove_task(adj_star)
                    pq.add_task(adj_star,max(vertices[u].d,
graph[u][adj_star]))

    return S, vertices
```

```
Archanas-MBP:HW10 archana$ python3 hw10_final.py
True
Longest distance that must be traveled to reach all stars is  2.4676e+16
Longest distance is between J-JS0D and 09-8TH
Total time 44.89383450895548
Final answer 4:  2.4676338729258668e+16
J-JS0D 09-8TH
```