CS532 Homework 10

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():

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:

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)

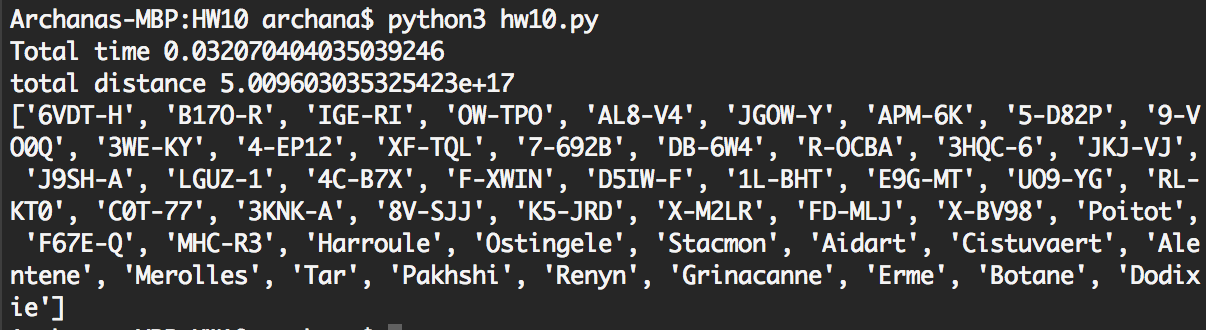
route = [reverse\_map[system] for system in jita\_dodixie\_route]

print(route)

return route

def question1():

q1\_shortest\_path("6VDT-H", "Dodixie")



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])

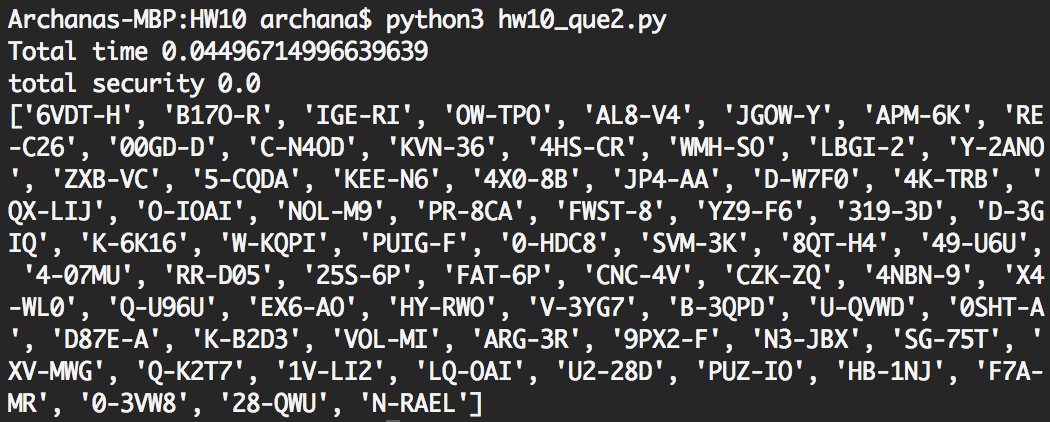
vertices[adj\_star].pi = vertices[u]

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.



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 != []:

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)

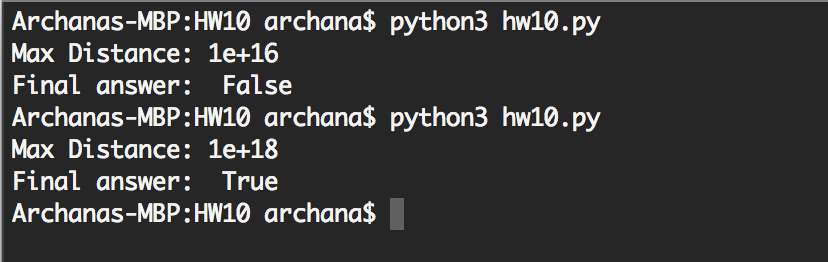
vertices[u].color = 'black'

return processed

def question3():

answer = question\_3()

print('Final answer: ',answer)



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(V3). The initial adjacency matrix calculation has time complexity O(V2). 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(V3). 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)

