CS532 Homework 10

Archana Machireddy

Question 1

My priority queue update method is not O(log n). I used a procedure similar to heap increase key which is O(log n) but in order to use this method, I need to know the index of the system in the list given by queue.PriorityQueue.queue, and that take O(n) time.

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 parent(i: int) -> int:

return (i-1)//2

def update\_priority(Q, index, name: int, new\_priority: float):

"""Method updates priorities of the nodes in a procedure similar to Heap-increase key; by comparing an element with its parent and exchanging if parent key is higher

Arguments:

Q {Priority Queue} -- Priority queue to update

name {int} -- ID of the system whose priority needs to be updated

new\_priority {float} -- New priority to be updated to

Returns:

None

"""

Q.queue[index] = (new\_priority, name)

while index > 0 and Q.queue[parent(index)][0] > Q.queue[index][0]:

Q.queue[index],Q.queue[parent(index)] = Q.queue[parent(index)], Q.queue[index]

index = parent(index)

#def backtrace(distances: Dict[(int, int),float], node: Vertex) -> (float, List[int]):

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)

#def dijkstra(graph: List[List[int]], distances: {Dict[(int, int),float]}, source: int, destination: int

# ) -> List[int]:

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,vertices[index].identifier))

while not Q.empty():

d,u = Q.get()

S.add(u)

if u == destination:

return backtrace(distances, vertices[destination])

for adj\_star in graph[u]:

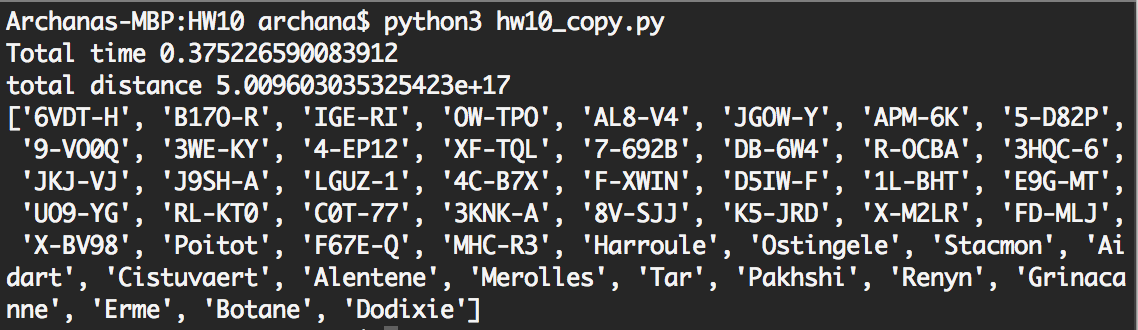
if vertices[adj\_star].d > vertices[u].d + distances[u, adj\_star]:

vertices[adj\_star].d = vertices[u].d + distances[u, adj\_star]

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

index = [y[1] for y in Q.queue].index(adj\_star)

update\_priority(Q, index, adj\_star, vertices[adj\_star].d)



Question 2

Here is the distance term represents the security status. Whenever I find a system with lower security status, I update the distance term to the new security status.

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 = dijkstra1(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 dijkstra1(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,vertices[index].identifier))

while not Q.empty():

d,u = Q.get()

S.add(u)

if u == destination:

return backtrace1(security, vertices[destination])

for adj\_star in graph[u]:

if adj\_star not in S:

if vertices[adj\_star].d > security[adj\_star]:

vertices[adj\_star].d = security[adj\_star]

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

index = [y[1] for y in Q.queue].index(adj\_star)

update\_priority(Q, index, adj\_star, security[adj\_star])

def backtrace1(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 = 0

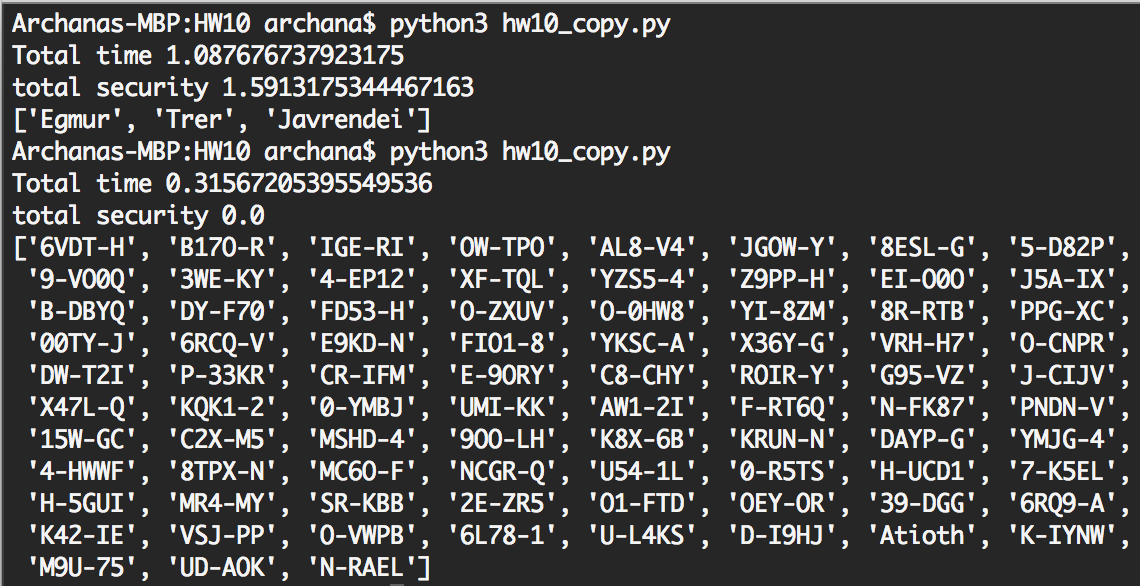
while node.pi is not None:

final\_sec = final\_sec + security[path[0]]

path.insert(0, node.pi.identifier)

node = node.pi

return (final\_sec,path)



Whenever you find a system with lower security status, the parent is updated. So a path taken by the processed node in S is always the shortest path and the score along the path is always minimal. 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 s fully connected, else it is not. 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+18

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'

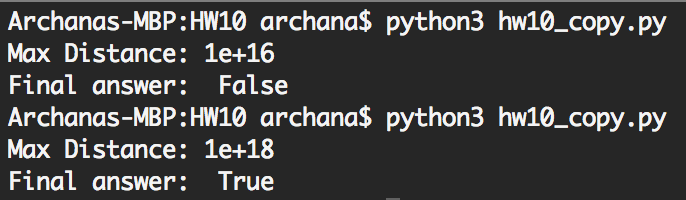
vertices[adj\_star].d = vertices[u].d + distance

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

queue.append(adj\_star)

vertices[u].color = 'black'

return processed



Question 4

I use the Prim’s algorithm to find the shortest path from one source. I tried running it with each system as the source. But it took too long for the program to run. So I randomly selected multiple start points and ran the algorithm. During these runs I got the minimum max\_distance as 9.312714997115474e+17 between RKE-CP and MVUO-F.

I use Dijkstra’s algorithm and find the shortest path by using each system as the source once. I got the minimum max\_distance as 9.712106816772236e+17 between 16-31U and MVUO-F.

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,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: [List[int]], coordinates\_id: Dict[int, List[float]],reverse\_map: Dict[int,str]):

"""Implements Prim's Algorithm starting from 50 different start points

Arguments:

systems {List[int]} -- List of all systems

coordinates\_id {Dict[int, List[float]]} -- Dictionary containg the x,y,z coordinates of all systems

reverse\_map {Dict[int,str]} -- Mapping from system index to system name

Returns:

Final\_min\_dist[float] -- Returns the final minimal distance

start[int] -- System index of the star from which the longest path starts

finish[int] -- System index of the star at which the longest path ends

"""

distance\_matrix = [[euclidean\_dist(coordinates\_id[i], coordinates\_id[j]) for i in systems] for j in systems]

# initialization of the nodes

final\_min\_dist = 0

start = 0

finish = 0

total = len(systems)

a = 0

while a < 50:

source = random.randint(0,total)

vertices = [Vertex(index) for index, \_ in enumerate(systems)]

vertices[source].d = 0

S = set()

Q = queue.PriorityQueue()

for index, \_ in enumerate(systems):

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

num = 0

while not Q.empty():

d,u = Q.get()

num = num + 1

S.add(u)

for adj\_star,dist in enumerate(distance\_matrix[u]):

if adj\_star in [y[1] for y in Q.queue] and dist < vertices[adj\_star].d:

vertices[adj\_star].d = dist

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

index = [y[1] for y in Q.queue].index(adj\_star)

update\_priority(Q, index, adj\_star, dist)

if dist > final\_min\_dist:

final\_min\_dist = dist

start = u

finish = adj\_star

print(source,final\_min\_dist,reverse\_map[start],reverse\_map[finish])

return final\_min\_dist,start,finish