utilities.py

```python
import networkx as nx
import random
from enum import Enum
import copy

class ServiceType(Enum):
  Pickup = 1
  Dropoff = 2

class ServiceRequest:
  def __init__(self, customer_id, service_type, destination):
    self.self = self
    self.customer_id = customer_id
    self.service_type = service_type
    self.destination = destination

  def as_string(self):
    return f"\n(CustID: {self.customer_id}, Type: {self.service_type}, Dest: {self.destination})"

class Van:
  def __init__(self, id):
    self.self = self
    self.id = id
    self.queue = []
    self.route = []
    self.distance_travelled = 0
    self.trips_taken = 0

  def queue_as_string(self):
    queue_string = ""
    for request in self.queue:
      queue_string += request.as_string() + ", "

    return queue_string

  def pickup_or_dropoff(self):
    if len(self.queue) == 0:
      pass
    else:
      # If route tail is at the location of a pickup or dropoff, do it
      while len(self.queue) != 0 and self.route[-1] == self.queue[0].destination:
        request = self.queue.pop(0)

  def move_to_next_node(self, G):
    if len(self.queue) == 0:
      pass
    else:
      shortest_path = nx.shortest_path(G, self.route[-1], self.queue[0].destination, weight='weight', method='dijkstra')
      if len(shortest_path) > 1:
        next_node = shortest_path[1]

        self.distance_travelled += nx.astar_path_length(G, self.route[-1], next_node, weight='weight')
        self.trips_taken += 1

        self.route.append(next_node)

  def is_service_queue_full(self):
    customer_ids_in_queue = []
    if len(self.queue) == 0:
      pass
    else:
      for request in self.queue:
        if request.customer_id in customer_ids_in_queue:
          pass
        else:
          customer_ids_in_queue.append(request.customer_id)

    if len(customer_ids_in_queue) == 5:
      return True
    else:
      return False

  def sort_service_queue2(self, G):
    def distance(x):
      return nx.astar_path_length(G, self.route[-1], x.destination, weight='weight')

    self.queue = sorted(self.queue, key=lambda x: (distance(x), (x.customer_id, x.service_type == ServiceType.Dropoff)))


def assign_customers_to_best_van(vans, unassigned_service_requests, G):

  # Check if all vans are full, if so tell customers to try again
  full_van_counter = 0
  for van in vans:
    if van.is_service_queue_full():
      full_van_counter += 1

  if len(vans) == full_van_counter:
    return

  # Get the list of unassigned pickups
  unassigned_pickups = filter(lambda r: r.service_type == ServiceType.Pickup, unassigned_service_requests)

  for unassigned_pickup in unassigned_pickups:
    list_of_distances = []

    # Get distance of service request from each van
    for van in vans:
      if van.is_service_queue_full():
        pass
      else:
        distance = nx.dijkstra_path_length(G, van.route[-1], unassigned_pickup.destination, weight='weight')
        list_of_distances.append({"distance": distance, "van": van})

    # Sort the list of distances
    sorted_distances = sorted(list_of_distances, key=lambda x: x['distance'])

    # Check if the shortest distance in the list equals the next shortest distance in the list
    if len(sorted_distances) > 1 and sorted_distances[0]["distance"] == sorted_distances[1]["distance"]:
      assigned_to_van = False

      # If tiebreaker, try to assign to first (lowest ID) empty van
      for van in vans:
        if len(van.queue) == 0 and not assigned_to_van:

          # Add the pickup and dropoff request
          dropoff_request = next(filter(lambda r: r.service_type == ServiceType.Dropoff and r.customer_id == unassigned_pickup.customer_id, unassigned_service_requests))

          van.queue.append(unassigned_pickup)
          van.queue.append(dropoff_request)
          assigned_to_van = True

      # If no vans are empty, assign to lowest ID van
      if sorted_distances[0]["van"].id < sorted_distances[1]["van"].id and not assigned_to_van:

        # Add the pickup and dropoff request
        dropoff_request = next(filter(lambda r: r.service_type == ServiceType.Dropoff and r.customer_id == unassigned_pickup.customer_id, unassigned_service_requests))

        sorted_distances[0]["van"].queue.append(unassigned_pickup)
        sorted_distances[0]["van"].queue.append(dropoff_request)
        assigned_to_van = True

      elif not assigned_to_van:
        # Add the pickup and dropoff request
        dropoff_request = next(filter(lambda r: r.service_type == ServiceType.Dropoff and r.customer_id == unassigned_pickup.customer_id,
unassigned_service_requests))

        sorted_distances[1]["van"].queue.append(unassigned_pickup)
        sorted_distances[1]["van"].queue.append(dropoff_request)
        assigned_to_van = True

    # If not a tie
    elif len(sorted_distances) > 0:
      dropoff_request = next(filter(lambda r: r.service_type == ServiceType.Dropoff and r.customer_id == unassigned_pickup.customer_id, unassigned_service_requests))

      sorted_distances[0]["van"].queue.append(unassigned_pickup)
      sorted_distances[0]["van"].queue.append(dropoff_request)

```