```
import networkx as nx
nodes = 100
seed=1000           # seed the graph for reproducibility, you should be doing this
G= nx.gnp_random_graph (nodes, .02, seed=seed )       # here we create a random binomial graph with 10 nodes and an average (expe
print ( G.nodes() )
```

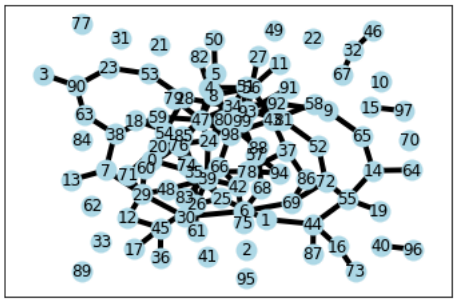
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 3
```

```
print(G.edges())
```

```
[(0, 35), (0, 54), (1, 25), (1, 44), (3, 90), (4, 47), (4, 56), (4, 82), (5, 50), (5, 80), (6, 30), (6, 68), (6, 69), (6, 75
```

```
import matplotlib.pyplot as plt
import random
import string
links = [(u, v) for (u, v, d) in G.edges(data=True)]
pos = nx.nx_pydot.graphviz_layout(G)

nx.draw_networkx_nodes(G, pos, node_size=250, node_color='lightblue', linewidths=0.07) # draw nodes
nx.draw_networkx_edges(G, pos, edgelist=links, width=4)                                 # draw edges

# node labels
nx.draw_networkx_labels(G, pos, font_size=12, font_family="sans-serif")
# edge weight labels

edge_labels = nx.get_edge_attributes(G, "weight")

nx.draw_networkx_edge_labels(G, pos, edge_labels)
fig = plt.figure(figsize=(100,20))
plt.show()
```

```
<ipython-input-3-1ceb8441640b>:5: DeprecationWarning: nx.nx_pydot.graphviz_

See https://github.com/networkx/networkx/issues/5723
  pos = nx.nx_pydot.graphviz_layout(G)
```



```
<Figure size 7200x1440 with 0 Axes>
```

van description

- id: unique id for the car
- cur: current node
- path: queue that includes all nodes that are going to be visited
- S: schedule queue. It is a queue of tuples. Tuple has 3 items where:
  - `tuple[0]` passenger name
  - `tuple[1]` location point
  - `tuple[2]` status `0 = pickup 1 = dropoff -1 = parked`
- clients: number of clients in the car
- pending: stores array of tuples, where each tuple has `passengerID`, `pickupNode` and `dropoffNode`

```
#function generates unique id that is used in vans and passengers
def generateUID():
    chars = ''.join(random.choices(string.ascii_uppercase, k=2))
    ints = ''.join(random.choices(string.digits, k=4))
    result = chars + ints
```

```
        return result

    #the folllowiing is an example how vans look like
    vans = [
        {'id':1,
         'cur':0,
        'path':[0,3,2],
        'S':[('p1',3,0),('p1',2,1)],
        'clients':1,
        'pending':[('p1',12,22)]
        },
        {'id':2,
         'cur':2,
        'path':[2,5,4],
        'S':[('p2',5,0),('p2',4,1)],
        'clients':1,
        'pending':[('p2',55,3)]
        },
        {'id':3,
         'cur':2,
        'path':[2],
        'S':[('p',2,-1)],
        'clients':0,
        'pending':[('p3',12,6)]
        }
    ]

    #functioin generates vans that are randomly placed in one of the nodes in a graph. Function returns an array with n vans
    def generateVans(number_of_nodes,n=1):
        arr = []
        for i in range(n):
            uid = generateUID()
            start = random.randint(0,number_of_nodes-1)
            arr.append({'id':uid,
                        'cur':start,
                        'path':[start],
                        'S':[('',start,-1)],
                        'clients':0,
                        'pending':[]
                    })
        return arr




    #updates path for next points
    def updatePath(van,newNodes):
        return van['path']+newNodes[1:]


    # R is a list of requests
    R = []
    #generates n number of new requests
    def generateOrders(number_of_nodes,R, n=1):
        newR = R
        counter = 0
        while counter<n:
            pickup = random.randint(0,number_of_nodes-1)
            dropoff = random.randint(0,number_of_nodes-1)
            #check if nodes have path
            if nx.has_path(G,pickup,dropoff):
                p = generateUID()
                newR.append((p,pickup,dropoff))
                counter+=1
        return newR



    #function returns shortest path between nodes a,b
    def dist(a,b):
      path = nx.shortest_path(G,a,b,weight='weight',method='dijkstra')
      return len(path)

    print(dist(17,6))
    #function adds pickup/dropoff points to the 'S' queue of van v
    def schedule(v,p,state):
```

```
  if state == 0: #picku[]
    path = nx.shortest_path(G,v['path'][-1],p[1],weight='weight',method='dijkstra')
    #check if parked
    if v['S'][0][2]==-1:
      v['S'].pop(0)
    v['S'].append((p[0],p[1],0))
    v['path']=updatePath(v,path)
  elif state == 1:
    path = nx.shortest_path(G,v['path'][-1],p[2],weight='weight',method='dijkstra')
    #check if parked
    if v['S'][0][2]==-1:
      v['S'].pop(0)
    v['S'].append((p[0],p[2],1))
    v['path']=updatePath(v,path)
  return v



   6
```

Simulation start here

First 4 minutes of simulation cars will stay parked

When orders come in you will see how every order is being added to a pending order of van v

When assigning orders is done, algorithm schedules pending order of every van

```
import time


#TODO
#generate vans here
vans = generateVans(nodes,60)
for i in vans:
  print(i)

print("Timer started")
# Define the timer duration (20 minutes)
duration = 480 * 60

total_distance = 0
total_trips = 0

# Start the timer
start = time.time()
elapsed_time=start


# Loop until elapsed time reaches the duration
while (elapsed_time - start) < duration:
    # Calculate elapsed time
    current_minute = int((elapsed_time - start)/60) + 1
    #current_minute = int((elapsed_time - start)) + 1
    # Sleep for 1 minute
    time.sleep(60)

    elapsed_time = time.time()
    # Print a message every minute
    print(f"\nMinute {current_minute} elapsed")

    if  current_minute % 4 == 0:
        print(f"Minute {current_minute} elapsed")
        R = generateOrders(nodes,R,30)
        print(R)
        while len(R)>0:
            #find closest driver
            d={}
            #number of unreached
            nu = 0
            for i in range(len(vans)):
                if nx.has_path(G,vans[i]['path'][-1],R[0][1]):
                    if vans[i]['clients']<3:
                        cur = len(nx.shortest_path(G,vans[i]['path'][-1],R[0][1],weight='weight',method='dijkstra'))
                        d[cur]=i
                    else:
```

```
                else:
                    # if full -> skip iteration
                    continue
            else:
                nu+=1

        #check if vans cant go there
        if len(vans)==nu:
          nu=0
          print('vans dont go there')
          R.pop(0)
        else:
          if len(d)>0:
            smallest_key = min(d.keys())
            #index of the closest driver
            van_index = d[smallest_key]
            print('order '+str(R[0])+' is added to van')
            print(str(vans[van_index])+' before update')
            vans[van_index]['clients']+=1
            vans[van_index]['pending'].append(R[0])
            print(str(vans[van_index])+' updated')
            total_trips+=1
            R.pop(0)
          else:
            break


    # iterate through drivers again and schedule everything
    for i in range(len(vans)):
        if len(vans[i]['pending'])==1:
          p1 = vans[i]['pending'][0]
          #p1 and p2 structure: (id, pickup, dropoff)
          w = vans[i]['path'][-1]

          print('\n\n1p case')
          vans[i] = schedule(vans[i],p1,0)
          vans[i] = schedule(vans[i],p1,1)
          vans[i]['pending'].pop(0)
          print('scheduling done: '+str(vans[i]))

        elif len(vans[i]['pending'])==2:
          print('\n\n 2p case')
          print(vans[i])
          p1 = vans[i]['pending'][0]
          p2 =  vans[i]['pending'][1]
          #p1 and p2 structure: (id, pickup, dropoff)
          w = vans[i]['path'][-1]

          # if dist(w,p2[1])+dist(p1[1],p2[1]<dist(w,p1[1])):
          if dist(w,p2[1])<dist(w,p1[1]):
            #pickup p2
            print('p p2')
            vans[i] = schedule(vans[i],p2,0)
            # path = nx.shortest_path(G,w,p2[1],weight='weight',method='dijkstra')
            # vans[i]['S'].append((p2[0],p2[1],0))
            # vans[i]['path']=updatePath(vans[i],path)

            #check if p1 has to be picked up before dropping off p2
            if dist(w,p1[1])<dist(w,p2[2])+dist(p2[2],p1[1]):
              #pick up p1
              print('p p1')
              vans[i] = schedule(vans[i],p1,0)
              if dist(w,p1[2])<dist(w,p2[2]):
                print('d p1')
                print('d p2')
                #dropoff p1 first
                vans[i] = schedule(vans[i],p1,1)
                #dropoff p2
                vans[i] = schedule(vans[i],p2,1)
              else:
                print('d p2')
                print('d p1')
                #dropoff p2 first
                vans[i] = schedule(vans[i],p2,1)
                #dropoff p1
                vans[i] = schedule(vans[i],p1,1)
            else:
              print('d p2')
```

```
        print('p p1')
        print('d p1')
        #dropping off p2 first
        vans[i] = schedule(vans[i],p2,1)
        #pickup p1

        vans[i]=schedule(vans[i],p1,0)
        #dropoff p1 after picking up p1
        vans[i] = schedule(vans[i],p1,1)

    else:
    #check if p2 has to be picked up before dropping off p1
    #elif dist(w,p1[1])>dist(w,p2[1])+dist(p2[1],p1[1]):
      #pickup p1 first
      print('p p1')
      vans[i] = schedule(vans[i],p1,0)
      if dist(w,p2[1])<dist(w,p1[2])+dist(p1[2],p2[1]):
        #pickup p2
        print('p p2')
        vans[i] = schedule(vans[i],p2,0)
        if dist(w,p1[2])<dist(w,p2[2]):
          print('d p1')
          print('d p2')
          #dropoff p1 first
          vans[i] = schedule(vans[i],p1,1)
          #dropoff p2
          vans[i] = schedule(vans[i],p2,1)
        else:
          print('d p2')
          print('d p1')
          #dropoff p2 first
          vans[i] = schedule(vans[i],p2,1)
          #dropoff p1
          vans[i] = schedule(vans[i],p1,1)
      else:

        #dropping off p1 first
        vans[i] = schedule(vans[i],p1,1)
        print('d p1')
        #pickup p2
        print('p p2')
        vans[i]=schedule(vans[i],p2,0)
        #dropoff p1 after picking up p1
        vans[i] = schedule(vans[i],p2,1)
        print('d p2')

    #clearing pending buffer after shceduling is done                        ]
    vans[i]['pending'].pop(0)
    vans[i]['pending'].pop(0)
    print('2 passenger scheduling done: '+str(vans[i]))

  elif len(vans[i]['pending'])==3:
    print('schedule 3 p')
    p1 = vans[i]['pending'][0]
    p2 =  vans[i]['pending'][1]
    p3 =  vans[i]['pending'][2]
    w = vans[i]['path'][-1]

    # check if p1 has to be picked up before p2 and p3
    if dist(w,p1[1])<dist(w,p2[1])+dist(p2[1],p1[1]):
      #pickup p1
      print('picking up p1')
      vans[i] = schedule(vans[i],p1,0)
      #check if p2 has to be picked up before p3
      if dist(w,p2[1])<dist(w,p3[1])+dist(p3[1],p2[1]):
        #pickup p2
        print('picking up p2')
        vans[i] = schedule(vans[i],p2,0)
        #check if p3 has to be picked up before p1
        if dist(w,p3[1])<dist(w,p1[2])+dist(p1[2],p3[1]):
          #pickup p3
          print('picking up p3')
          vans[i] = schedule(vans[i],p3,0)
          #check if p1 has to be dropped off before p2
          if dist(w,p1[2])<dist(w,p2[2]):
            #dropoff p1
            print('dropping off p1')
```

```
            vans[i] = schedule(vans[i],p1,1)
            #check if p2 has to be dropped off before p3
            if dist(w,p2[2])<dist(w,p3[2]):
              #dropoff p2
              print('dropping off p2')
              vans[i] = schedule(vans[i],p2,1)
              #dropoff p3
              print('dropping off p3')
              vans[i] = schedule(vans[i],p3,1)
            else:
              #dropoff p3
              print('dropping off p3')
              vans[i] = schedule(vans[i],p3,1)
              #dropoff p2
              print('dropping off p2')
              vans[i] = schedule(vans[i],p2,1)
          else:
            #dropoff p2
            print('dropping off p2')
            vans[i] = schedule(vans[i],p2,1)
            #check if p1 has to be dropped off before p3
            if dist(w,p1[2])<dist(w,p3[2]):
              #dropoff p1
              print('dropping off p1')
              vans[i] = schedule(vans[i],p1,1)
              #dropoff p3
              print('dropping off p3')
              vans[i] = schedule(vans[i],p3,1)
            else:
              #dropoff p3
              print('dropping off p3')
              vans[i] = schedule(vans[i],p3,1)
              #dropoff p1
              print('dropping off p1')
              vans[i] = schedule(vans[i],p1,1)
        else:
          #pickup p1
          print('picking up p1')
          vans[i] = schedule(vans[i],p1,0)
          #check if p2 has to be dropped off before p3
          if dist(w,p2[2])<dist(w,p3[2]):
            #dropoff p2
            print('dropping off p2')
            vans[i] = schedule(vans[i],p2,1)
            #dropoff p3
            print('dropping off p3')
            vans[i] = schedule(vans[i],p3,1)
          else:
            #dropoff p3
            print('dropping off p3')
            vans[i] = schedule(vans[i],p3,1)
            #dropoff p2
            print('dropping off p2')
            vans[i] = schedule(vans[i],p2,1)
      else:
        #pickup p3
        print('picking up p3')
        vans[i] = schedule(vans[i],p3,0)
        #check if p1 has to be dropped off before p2
        if dist(w,p1[2])<dist(w,p2[2]):
          #dropoff p1
          print('dropping off p1')
          vans[i] = schedule(vans[i],p1,1)
          #dropoff p2
          print('dropping off p2')
          vans[i] = schedule(vans[i],p2,1)
        else:
          #dropoff p2
          print('dropping off p2')
          vans[i] = schedule(vans[i],p2,1)
          #dropoff p1
          print('dropping off p1')
          vans[i] = schedule(vans[i],p1,1)
    else: #check if p2 has to be picked up before p1 and p3
      #pickup p2
      print('picking up p2')
      vans[i] = schedule(vans[i],p2,0)
      #check if p3 has to be picked up before p1
```

```
                #check if p3 has to be picked up before p1
                if dist(w,p3[1])<dist(w,p1[1]):
                  #pickup p3
                  print('picking up p3')
                  vans[i] = schedule(vans[i],p3,0)
                  #check if p1 has to be dropped off before p2
                  if dist(w,p1[2])<dist(w,p2[2]):
                    #dropoff p1
                    print('dropping off p1')
                    vans[i] = schedule(vans[i],p1,1)
                    #dropoff p2
                    print('dropping off p2')
                    vans[i] = schedule(vans[i],p2,1)
                  else:
                    #dropoff p2
                    print('dropping off p2')
                    vans[i] = schedule(vans[i],p2,1)
                    #dropoff p1
                    print('dropping off p1')
                    vans[i] = schedule(vans[i],p1,1)
                else:
                  #pickup p1
                  print('picking up p1')
                  vans[i] = schedule(vans[i],p1,0)
                  #check if p2 has to be dropped off before p3
                  if dist(w,p2[2])<dist(w,p3[2]):
                    #dropoff p2
                    print('dropping off p2')
                    vans[i] = schedule(vans[i],p2,1)
                    #dropoff p3
                    print('dropping off p3')
                    vans[i] = schedule(vans[i],p3,1)
                  else:
                    #dropoff p3
                    print('dropping off p3')
                    vans[i] = schedule(vans[i],p3,1)
                    #dropoff p2
                    print('dropping off p2')
                    vans[i] = schedule(vans[i],p2,1)


              #clearing pending buffer after shceduling is done
              vans[i]['pending'].pop(0)
              vans[i]['pending'].pop(0)
              vans[i]['pending'].pop(0)
              print('scheduling done: '+str(vans[i]))

        #update cars locations:
        for v in vans:
            if len(v['path'])==1 and v['S'][0][2]==-1 and (v['S'][0][1]==v['path'][0]) and v['clients']==0:
                print('van '+str(v['id']) +' is parked')
            else:
                if current_minute % 2 == 0:
                    #reached node so update cur
                    v['cur']=v['path'][0]
                    total_distance+=1
                    #if van is on pickup or dropoff point
                    while v['path'][0]==v['S'][0][1]:

                        if v['S'][0][2]==0:
                            print('van '+str(v['id']) +' picked up ' + v['S'][0][0])
                        else:
                            print('van '+str(v['id']) +' dropped off ' + v['S'][0][0])
                            v['clients']-=1

                        #if it is a last stop -> pop it and park. else -> pop it and go on
                        if len(v['S'])==1:
                            print('finished orders. Van is now parked')
                            v['S'].pop(0)
                            v['S'].append(('',v['path'][0],-1))
                            v['clients']=0
                            break
                        else:
                            v['S'].pop(0)
                            print('stops left'+str(v['S'])+'')
                else:
                    if v['S'][0][2]>-1 and v['cur']!=-1:
                        v['path'].pop(0)
```

```
                    v['cur']=-1
                print('van '+str(v) +' is on its way. path left: '+str(v['path']))
    average_distance = total_distance/len(vans)
    average_trips = total_trips/len(vans)
    print("Timer finished")
    print('Average distance traveled: '+str(average_distance))
    print('Average trips made: '+str(average_trips))
```

```
{'id': 'SG6087', 'cur': -1, 'path': [24, 47, 34, 92, 9, 65, 14, 55, 44, 1, 25, 39, 42, 66, 42, 39, 24], 'S': [('CM5021', 6
{'id': 'SG6087', 'cur': -1, 'path': [24, 47, 34, 92, 9, 65, 14, 55, 44, 1, 25, 39, 42, 66, 42, 39, 24], 'S': [('CM5021', 6

order ('UV8379', 7, 82) is added to van
{'id': 'OT9659', 'cur': -1, 'path': [45, 26, 35, 78, 37, 43, 58, 43, 98, 47, 34, 28, 53, 23, 90, 63, 38, 7, 29, 60], 'S':
{'id': 'OT9659', 'cur': -1, 'path': [45, 26, 35, 78, 37, 43, 58, 43, 98, 47, 34, 28, 53, 23, 90, 63, 38, 7, 29, 60], 'S':

order ('RP2453', 74, 73) is added to van
{'id': 'NT2270', 'cur': -1, 'path': [82, 4, 47, 98, 47, 4, 47], 'S': [('ZI1480', 82, 0), ('ZI1480', 98, 1), ('AT8944', 4,
{'id': 'NT2270', 'cur': -1, 'path': [82, 4, 47, 98, 47, 4, 47], 'S': [('ZI1480', 82, 0), ('ZI1480', 98, 1), ('AT8944', 4,

order ('NZ2648', 28, 52) is added to van
{'id': 'HW9177', 'cur': -1, 'path': [47, 59, 38, 7, 13, 7, 38, 59, 47, 98, 43, 37, 78, 94, 57, 99, 47, 34, 28, 53], 'S': [
{'id': 'HW9177', 'cur': -1, 'path': [47, 59, 38, 7, 13, 7, 38, 59, 47, 98, 43, 37, 78, 94, 57, 99, 47, 34, 28, 53], 'S': [

order ('YF6861', 30, 99) is added to van
{'id': 'SG3574', 'cur': -1, 'path': [42, 66, 98, 43, 58, 43, 98, 66, 42, 75, 6], 'S': [('HA5052', 42, 0), ('HA5052', 58, 1
{'id': 'SG3574', 'cur': -1, 'path': [42, 66, 98, 43, 58, 43, 98, 66, 42, 75, 6], 'S': [('HA5052', 42, 0), ('HA5052', 58, 1

order ('HY0356', 16, 68) is added to van
{'id': 'DR6622', 'cur': -1, 'path': [93, 24, 39, 25, 39, 48, 29, 7, 29, 48, 39, 25, 1, 44, 16, 73], 'S': [('CC7323', 25, 0
{'id': 'DR6622', 'cur': -1, 'path': [93, 24, 39, 25, 39, 48, 29, 7, 29, 48, 39, 25, 1, 44, 16, 73], 'S': [('CC7323', 25, 0

order ('LR4302', 76, 58) is added to van
{'id': 'IT6263', 'cur': -1, 'path': [47, 59, 38, 7, 13, 7, 38, 59, 47, 34, 51], 'S': [('AX6198', 47, 0), ('AX6198', 13, 1)
{'id': 'IT6263', 'cur': -1, 'path': [47, 59, 38, 7, 13, 7, 38, 59, 47, 34, 51], 'S': [('AX6198', 47, 0), ('AX6198', 13, 1)

order ('UF2932', 52, 14) is added to van
{'id': 'YS9308', 'cur': -1, 'path': [23, 53, 28, 34, 80, 5, 80, 88, 68, 6, 75], 'S': [('WT9200', 5, 1), ('FH6858', 88, 0),
{'id': 'YS9308', 'cur': -1, 'path': [23, 53, 28, 34, 80, 5, 80, 88, 68, 6, 75], 'S': [('WT9200', 5, 1), ('FH6858', 88, 0),

order ('EJ9999', 29, 66) is added to van
{'id': 'ZJ9331', 'cur': -1, 'path': [5, 80, 24, 39, 26, 35, 78, 35, 26, 25, 1, 25, 26, 61], 'S': [('DA5414', 5, 0), ('DA54
{'id': 'ZJ9331', 'cur': -1, 'path': [5, 80, 24, 39, 26, 35, 78, 35, 26, 25, 1, 25, 26, 61], 'S': [('DA5414', 5, 0), ('DA54

order ('PL0406', 61, 88) is added to van
{'id': 'VY8920', 'cur': -1, 'path': [8, 47, 34, 81, 34, 47, 85, 35, 26, 39, 42], 'S': [('SL2532', 8, 0), ('SL2532', 81, 1)
{'id': 'VY8920', 'cur': -1, 'path': [8, 47, 34, 81, 34, 47, 85, 35, 26, 39, 42], 'S': [('SL2532', 8, 0), ('SL2532', 81, 1)

order ('FE9531', 99, 35) is added to van
{'id': 'EQ5419', 'cur': -1, 'path': [44, 55, 14, 65, 9, 92, 34, 47, 4, 82, 4, 47, 85, 35, 78, 94, 57, 42], 'S': [('SH0117'
{'id': 'EQ5419', 'cur': -1, 'path': [44, 55, 14, 65, 9, 92, 34, 47, 4, 82, 4, 47, 85, 35, 78, 94, 57, 42], 'S': [('SH0117'

order ('NP0521', 28, 75) is added to van
{'id': 'CX0224', 'cur': -1, 'path': [43, 37, 86, 69, 72, 55, 19, 55, 72, 69, 6, 68, 6, 69, 86, 37, 43], 'S': [('HS6748', 1
{'id': 'CX0224', 'cur': -1, 'path': [43, 37, 86, 69, 72, 55, 19, 55, 72, 69, 6, 68, 6, 69, 86, 37, 43], 'S': [('HS6748', 1

order ('WN8306', 78, 54) is added to van
{'id': 'SJ9396', 'cur': -1, 'path': [56, 11, 56, 43, 98, 47, 34, 92, 34, 47, 4, 56, 27], 'S': [('HI0821', 11, 0), ('HI0821
{'id': 'SJ9396', 'cur': -1, 'path': [56, 11, 56, 43, 98, 47, 34, 92, 34, 47, 4, 56, 27], 'S': [('HI0821', 11, 0), ('HI0821



1p case
scheduling done: {'id': 'AP8155', 'cur': -1, 'path': [81, 34, 28, 34, 47, 59, 38, 7, 38, 63, 38, 59, 47, 34, 81, 52], 'S':


1p case
```

R4

Avg distance traveled by a van is 206.43 nodes per day

The average number of trips made by the van is 29.36 trips

R5

Avg distance traveled by a van is 219.083 nodes per day

The average number of trips made by the van is 37.616 trips

R6

Avg distance traveled by a van is 222.95 nodes per day

The average number of trips made by the van is 54.06 trips

Explanation

Average connectivity means that every node has a certain number of connected nodes. When average connectivity is 4, it means that nodes have 4 neighbors on average. During simulation in R5 it's been noted that with the connectivity of 2, some nodes are not connected at all, which means that no ven can go there. On the other hand, with average connectivity of 4, most of the nodes are connected, which means that vans can get to any node on the graph. Average trips with a connectivity of 4 are higher than those with a connectivity of 2. The reason is that with more connections, it is easier for the van to find a better path and get to more nodes in a shorter period of time.