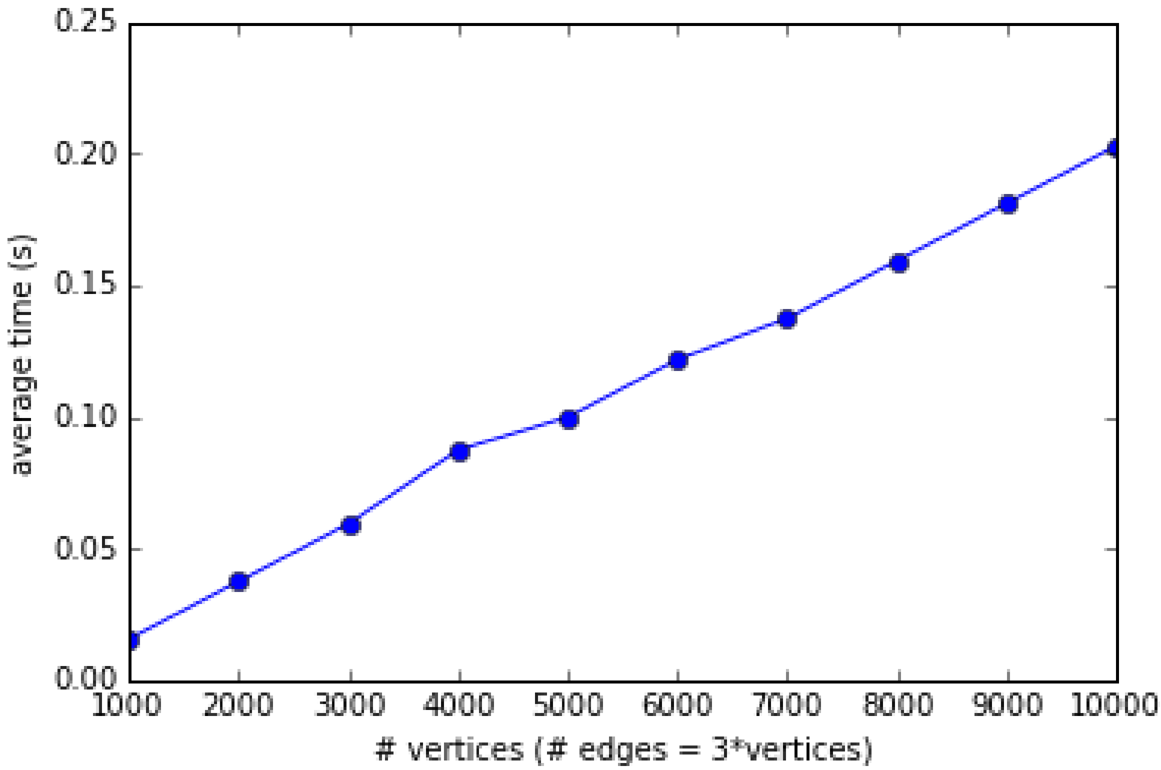
Assignment 3 – Steven Lin

Exercise 1

See code

Exercise 2

Complexity: linear (O(V)) , where V = number of vertices



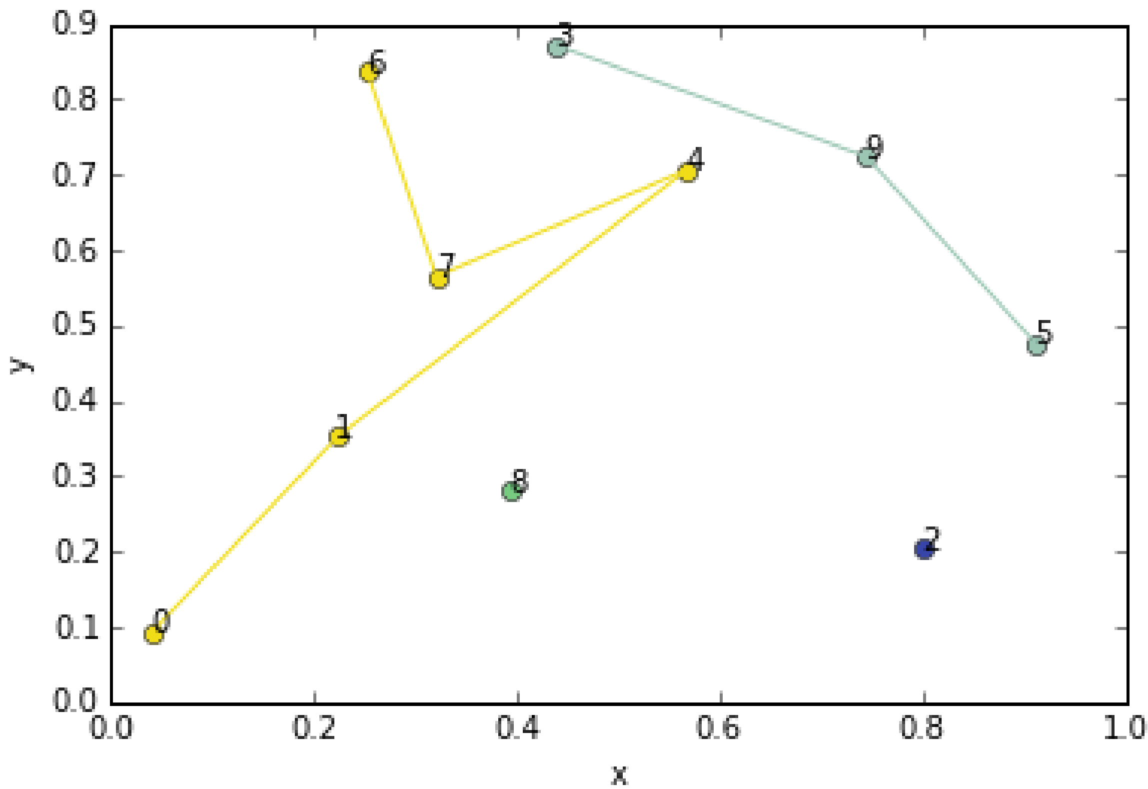
Exercise 3

See Code

Test:

```
**Key: connected_component, value: list of vertices**
{0: [0, 1, 4, 6, 7], 1: [2], 2: [3, 5, 9], 3: [8]}
```

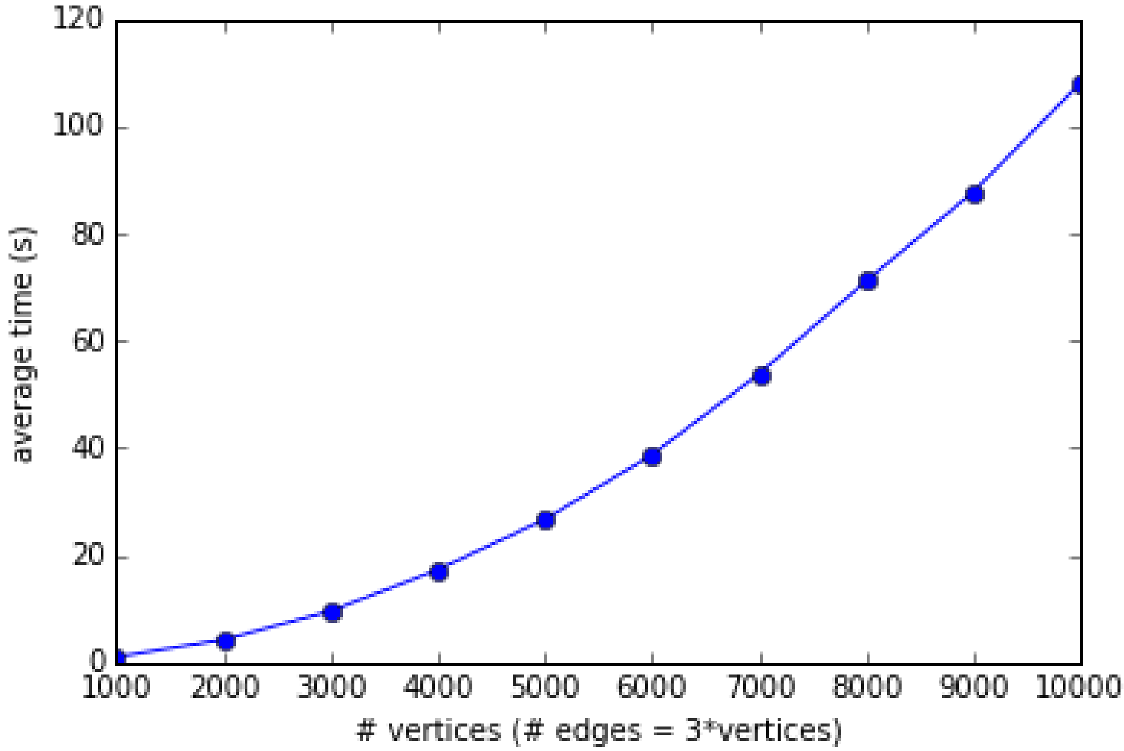
```
In [273]:
```



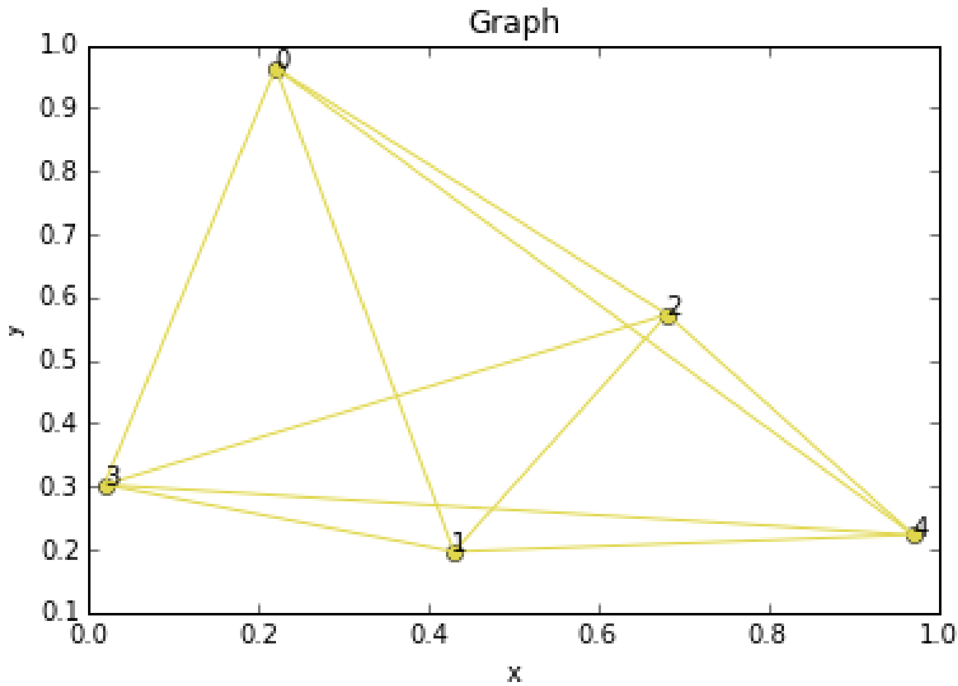
Exercise 4

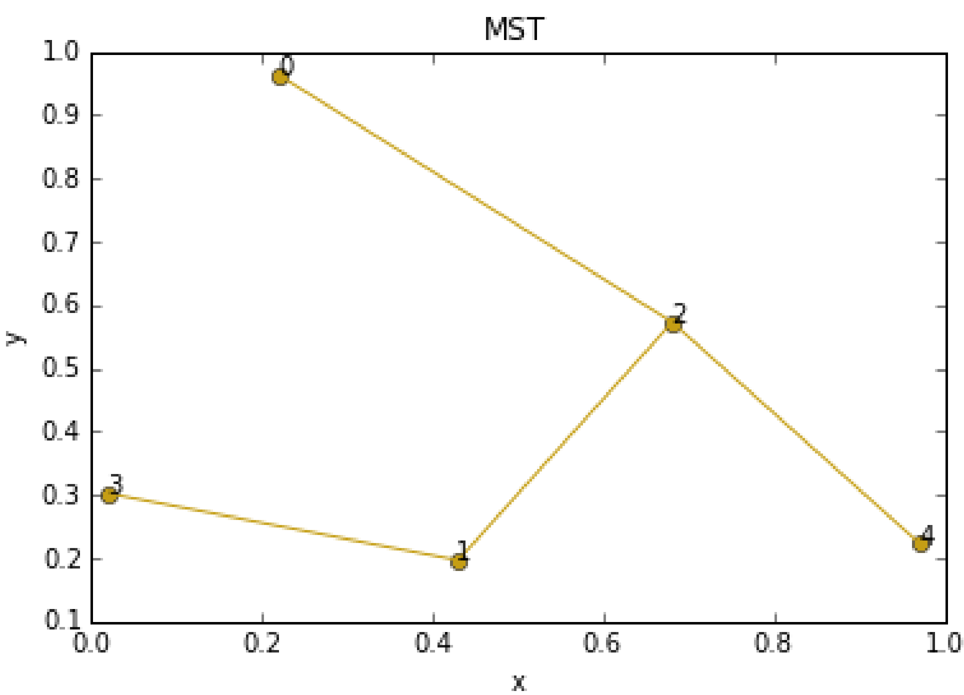
See Code

Complexity: O(E+V)*O(LogV) = O((E+V)*LogV) = O(E*LogV) where E = # edges, V = # vertices, for a connected graph, V = O(E)



Exercise 5


Graph
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0.2
0.4
0.6
0.8
1.0
x
y
0
1
2
3
4


MST
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0.2
0.4
0.6
0.8
1.0
x
y
0
1
2
3
4

Code

```
# -*- coding: utf-8 -*-
"""
Assignment 3
Steven Lin
4/20/15

Python: 3.4
Editor: Spyder
"""

import random
import queue
import time # I prefer using time.perf_counter rather than timeit as shown below
import matplotlib.pyplot as plt
import numpy as np
from copy import deepcopy

#%% Classes  ##############################################################

class Vertex:
    """
    Vertex class with attributes id and location
    Methods: set location

    """
    def __init__(self, id):
        self.id = id
        self.location = (None,None) # x,y coordinates

    def setLocation(self,x,y):
        self.location = (x,y)

class Edge:
    """
    Edge class with attributes id, pair of vertices, weight (default = 1)

    """
    def __init__(self, id, V1, V2, weight =1):
        self.id = id
        self.vertices = (V1,V2)   # Vertices objects
        self.weight = weight      # default value of if not given

class Graph:
    """
    Edge graph with attributes id
    Contains adjcancy list  (key: (Vertex id, Vertex id), value: weight)
    dictionry for vertices and edges where key is id and value object
    """
    def __init__(self, id=None):

        self.id = id
        self.adjacencyList = dict() # key: (Vertex id, Vertex id) value=  wt
        self.vertexList = dict() # key: id vertex, # value: vertex object
        self.edgeList = dict()    # key: id edge , # value: edge ojbect
        #self.connected = False

    def createAdjacecnyList2(self):
```

```
        """

        Method creates adjcacency list (dictionary) where
        #key: (vertex id), value: list adj vertex

        Takes no input and does not return anything


        """
        self.adjacencyList2 = dict()
        # note this list does not include vertices that are not
        # connected to any vertices

        pairs = list(self.adjacencyList.keys())

        for v1,v2 in pairs:

            # does not exist, so add key and value
            if v1 not in self.adjacencyList2:
                self.adjacencyList2[v1] = [v2]

            # already exists, so update list of the key
            else:
                self.adjacencyList2[v1].append(v2)

            # does not exist, so add key and value
            if v2 not in self.adjacencyList2:
                self.adjacencyList2[v2] = [v1]

            # already exists, so update list of the key
            else:
                self.adjacencyList2[v2].append(v1)

#%% Functions  ##############################################################
def findDistance(s,t, p=2):
    """
    returns the p-norm distance between two points

    @param s: coordinates of point 1 (x1,y1,z1...)
    @param t coordiantes of point 2 (x2,y2,z2,..)
    @param p: p-norm optional (p = p-norm, p = inf infinity norm, default =2)

        """

        points = zip(s,t)
        dist = 0

        # for each corresponding pair coordintates compute distance and sum up

        if p == 'inf':
            for p1,p2 in points:
                dist = max(dist,abs(p1-p2))

        else:
            for p1,p2 in points:
                dist += abs(p1-p2)**p
            return dist**(1/p)

    def randomPair(s, order=True):
        """
        Pick random pair of elements from a set
```

```

    @ param ls: set of elements (this implies unique elements in the sets)
    @ param order: T = order matters, F = doesn't
    @ return: tuple with random pair
      if order doesn't matter, sort tuple so that return (smallest, largest)

    Raise Exception: if the number is less than 2

    """
    ls = list(s)
    if len(ls)<2:
        raise RuntimeError("Number of items has to be >=2")
    v1 = random.choice(ls) # random value
    v2 = random.choice(ls) # random value

    # this will not be an infinit loop because unique elements and at least 2
    while(v1 == v2 ):
        v2 = random.choice(ls)

    if order:
        return (v1,v2)

    return tuple(sorted((v1,v2)))


#%% Exercise 1 ##############################################################

def randomGraph(v,e, graphID = None, connected = False):
    """
    Generate a graph object with random locations from (0,1) for (x,y) vertices

    @param v: number of vertices
    @param e: number of edges
    @param graphID: optional ID for graph (Default = None)
    @param connected: True = connected graph, False = not required to be
    connected (Default)

    @return: graph object

    """
    graph = Graph(graphID)
    graph.connected = connected

    # check edges does not exceed max number possible
    if e > v*(v-1)/2:
        raise RuntimeError("e: number of edges has to be <= v*(v-1)/2 ")

    # check for connected graph have the min number of edges
    if connected and e < v-1:

        raise RuntimeError("e: number of edges has to be at least >=  v-1")

    # create v vertices with id = i
    for vertexID in range(v):

        # generate random location
        x = random.random()
        y = random.random()

        # create instance and set location
        V = Vertex(vertexID)
```

```
        V.setLocation(x,y)

        # store object key = id, value = vertex object
        graph.vertexList[vertexID]=V


    allVertices = set(range(v))
    assignedVertices = set()
    remainingVertices = v

    # pick random start vertex
    assignedVertices.add(random.choice(list(allVertices)))

    # iterate until create e edges:
    edgeID = 0
    while (edgeID<e):

        # random pair of vertices (order in pair of vertices doesn't matter)
        # (v1,v2) is the same as (v2,v1), but need to store as one
        # uniqe key for dictionary, so get (smallest, largest) to be
        # consistent

        if (remainingVertices  == 0 or not connected ):
            pair = randomPair(allVertices, order=False)
            v1,v2 = pair

        else:
            v1 = random.choice(list(assignedVertices))
            v2 = random.choice(list(allVertices.difference(assignedVertices)))

            pair = tuple(sorted((v1,v2)))

        # generate a random pair of vertices until pair does not
        # exist in the adjancecyList (as a key in the dictionary)
        if (pair not in graph.adjacencyList):

            # get vertex objects and compute weight

            V1 = graph.vertexList[v1]
            V2 = graph.vertexList[v2]
            weight = findDistance(V1.location, V2.location)

            # create edge and add to edge list
            edge = Edge(edgeID, V1,V2, weight)
            graph.edgeList[edgeID] = edge

            # add pair as key and value = euclidean distance
            graph.adjacencyList[pair]= edge.weight

            # remove v1 and v2 from set unassignedVertices if present.
            # no error if not present (might be the case that the pair
            # does not exist, but one of the vertices might have already
            # been removed because was used in a different edge)

            # only do this for connected graph
            if (connected):
                assignedVertices.add(v2)

                remainingVertices = len(allVertices.difference(assignedVertices))

```

```python
            edgeID+= 1 # go to next edge

    return graph
#%% test not connected ##
g = randomGraph(5,3)
g.createAdjacecnyList2()

print(g.vertexList)
print(g.edgeList)

print(" ***** Adjacency List")
print(g.adjacencyList)
print(" ***** Adjacency List2")
print(g.adjacencyList2)

print(" ***** Vertex List")
for V in g.vertexList.values():
    print("ID: " + str(V.id) + ", " + "location: " + str(V.location))

print(" ***** Edge List")
for E in g.edgeList.values():
    print("ID: " + str(E.id) + ", " + "vertices: " +
    str(E.vertices[0].id) + "," + str(E.vertices[1].id) + ", " +
    "weight: " + str(E.weight))

v1 = g.edgeList[0].vertices[0].id
v2 = g.edgeList[0].vertices[1].id

print("***** Distance vertex " + str(v1) + " and " + str(v2))
findDistance(g.vertexList[v1].location,g.vertexList[v2].location )

#%% test connected ##
g = randomGraph(5,5,connected=True )
g.createAdjacecnyList2()

print(g.vertexList)
print(g.edgeList)

print(" ***** Adjacency List")
print(g.adjacencyList)
print(" ***** Adjacency List2")
print(g.adjacencyList2)

print(" ***** Vertex List")
for V in g.vertexList.values():
    print("ID: " + str(V.id) + ", " + "location: " + str(V.location))

print(" ***** Edge List")
for E in g.edgeList.values():
    print("ID: " + str(E.id) + ", " + "vertices: " +
    str(E.vertices[0].id) + "," + str(E.vertices[1].id) + ", " +
    "weight: " + str(E.weight))

v1 = g.edgeList[0].vertices[0].id
v2 = g.edgeList[0].vertices[1].id

print("***** Distance vertex " + str(v1) + " and " + str(v2))
findDistance(g.vertexList[v1].location,g.vertexList[v2].location )
```

```
#%% Exercise 2 #############################################################

#%% Functions ##############################################################

'''''
connected_componentList = dict()
# or dict.fromkeys(keys, None)
unassignedList = {key: None for key in g.vertexList.keys()}
currentGroup = dict()

# loop until unassigned list is empty
groupID = 0
while(bool(unassignedList)):

    # pick a vertex
    v = list(unassignedList.keys())[0]

    # holds vertices that need to be checked
    #q = queue.Queue()
    #q.put(v)

    # check vertex exists in current group
    #if (v not in currentGroup):

    #while (not q.empty()):

    pendingList = dict()
    pendingList[v]=None

    while(bool(pendingList)):

        #v = q.get(v) # pick vertex in queue and delete from queue

        # pick a vertex
        v = list(pendingList.keys())[0]

        currentGroup[v] = groupID # add vertext to current group
        del unassignedList[v] # delete from unassigned list
        del pendingList[v] # delete from pendingList

        print("processing" + str(v))

        # for all adjacent vertices that have not been assigned and
        # are not in the pending list

        if v in g.adjacencyList2:
            for i in g.adjacencyList2[v]:
                if ((i not in pendingList) and (i in unassignedList)):
                    print("adding to queue"+ str(i))
                    pendingList[i] = None
                    #q.put(i)

    print("finished group" + str(groupID))

    groupID +=1
        # check if adjacent vertices are in currentGroup and add to queue

'''


def findConnected_Components(graph):
```

```
        """
        Adds an integer property connected_component to each vertex so that
        vertexes in the same connected component have the same value and vertexes
        in different connected components have different values

        @return: None


        """

        unassignedSet = set(graph.vertexList.keys())  # set with all vertices not in group
        foundVertices = set() # queue OR assigned vertices included
        # loop until unassigned set is empty
        groupID = 0
        while(bool(unassignedSet)):

            v = unassignedSet.pop()  # pick an arbitrary vertex (removevs it so...)

            unassignedSet.add(v)     # add it back in

            q = queue.Queue()        # holds vertices that need to be checked
            q.put(v)                 # add vertex to queue
            #foundVertices[v] =None  # add vertex to found list
            foundVertices.add(v)

            # loop until no items in queue
            while (not q.empty()):

                v = q.get(v) # pick vertex in queue and delete from queue

                graph.vertexList[v].connected_component = groupID # assign group
                unassignedSet.remove(v)               # remove from unassigned set

                ##print("processing: " + str(v))

                # for all adjacent vertices that have not been assigned and
                # are not in the pending list

                # get adjacent vertices, if none then only 1 vertex in group
                # so move to another item in the unassgined set

                if v in graph.adjacencyList2:

                    # check every adjacent vertex
                    for i in graph.adjacencyList2[v]:

                        # only add to queue if vertex has not been found
                        if (i not in foundVertices):
                            q.put(i)                  # add vertex to queue
                            foundVertices.add(i)  # add vertex to found list
                            ##print("adding to queue: "+ str(i))

            ##print("finished group: " + str(groupID))
            groupID +=1

    def createConnectedComponentDict(graph):
        """
        Creates connectedComponent dictionary where key = vertexID, value = groupID

        @ param: graph with connected_components attribute
```

```
        @ return: dictionary key = vertexID, value = groupID

        """
        graph.connected_components = {}

        for V in graph.vertexList.values():
            graph.connected_components[V.id] = V.connected_component

        graph.connected_components2 = invertDictionary(graph.connected_components)



    def invertDictionary(dic):
        """
        Invers a dictionary k,v to v,k
        Function works for cases when values in dictionary not unique

        @param: dictionary
        @return: inverted dictionary
        """
        inv_dic = {}
        for k,v in dic.items():
            # if value has not been added as a key in inverse dic, then
            # set the value in the inv dic an empty list and then append key
            # as value in the inverted dictionary
            inv_dic[v] = inv_dic.get(v, [])
            inv_dic[v].append(k)

        return inv_dic

    def createEdgeListPair(graph):
        """
        Add a dictionary key = (vertex ID, vertex ID), value = Edge Object
        to graph

        @param: graph
        @return: none (add a dicionary to graph)

        """
        g.edgeListPair = {}

        for E in g.edgeList.values():
            pair = tuple(sorted((E.vertices[0].id, E.vertices[1].id)))
            g.edgeListPair[pair] = E


    #%% test function
    g = randomGraph(7,4)
    g.createAdjacecnyList2()
    findConnected_Components(g)

    createConnectedComponentDict(g)

    print(g.adjacencyList2)
    print(g.connected_components)
    print(g.connected_components2)

    #%% Scalability Function

    def findTimes(f,vertices,nreps):
        """
```

```
        Plots the average time vs # vertices for function f by creating
        a random graph in each repetition

        @param f:  function to test with argument = graph
        @param vertices: number of vertices of graph (3*n edges)
        @param nreps: number of repetitions

        @return: none (plots)

        """

        avg_time = []

        # iterate for different size of graphs
        for v in vertices:

            totalTime = 0

            #  repetitions random graphs
            for rep in range(0,nreps):

                # create random graph
                g = randomGraph(v,3*v, connected=True)
                g.createAdjacecnyList2()
                createEdgeListPair(g)

                # time findConnected_Components(g)
                timeStamp = time.process_time() # get the current cpu time
                f(g)
                timeLapse = time.process_time() - timeStamp
                totalTime += timeLapse

                print("rep: {}, time: {}".format(rep,timeLapse))

            # store average time
            avg_time.append(totalTime/nreps)

            print('p time: vertices[{0}]={1}'.format(v, totalTime/nreps))

        return [vertices,avg_time]



    #%% Scalability Test

    minsize = 1000
    maxsize = 10000+minsize
    stepsize = minsize
    vertices = list(range(minsize,maxsize,stepsize))
    # vertices = [10000]
    f =  findConnected_Components
    nreps = 5
    times_list =findTimes(f,vertices,nreps)

    # plot avg time vs # vertices

    plt.plot(times_list[0],times_list[1],'-bo')
    # Place a legend above this legend, expanding itself to
    # fully use the given bounding box.

    plt.xlabel("# vertices (# edges = 3*vertices)")
```

```
        plt.ylabel("average time (s)")
        plt.show()

        # complexity of algorithm is linear (O(V)) , V = Number of Vertices

        #%% Exercise 3 ##############################################################
        #%% Function

        def plotGraph(g, MST=False):
            """
            Plots a graph with vertices and edges and colored connected components
            Plots the mst if argument is given (edges need to have mst attribute)

            @param g: graph object
            @return: none (plots)

            """

            # get the x and y coordinates of vertices
            vertexLocations = []

            for V in g.vertexList.values():
                vertexLocations.append(V.location)

            x,y = list(zip(*vertexLocations))

            colors = deepcopy(g.connected_components2)

            # assign a random color to each connected_component
            for k in colors:
                colors[k] = (random.random(), random.random(), random.random())

            #N = len(g.vertexList)
            #colors = np.random.rand(N)
            #colors = list(cc.values()) # colors = corresponding connected_components

            # (r,g,b) where values between 0 and 1

            #fig, ax = plt.subplots()

            #plt.xlim(0,1.1)
            #plt.ylim(0,1.1)
            plt.xlabel("x")
            plt.ylabel("y")

            if MST:
                plt.title("MST")
            else:
                plt.title("Graph")
            #ax.scatter(x, y, c=colors, s = area, alpha=0.9)

            # plot points and labels
            for i in g.vertexList:
                plt.plot(x[i],y[i], marker = 'o',
                         color = colors[g.connected_components[i]])
                #print(i)
                #print(x[i])
                #print(y[i])
                #print("**")
                plt.annotate(i, (x[i],y[i]))

```

```
    # plot edges
    for e in g.edgeList:

        if MST:
            # plot only components in mst
            if g.edgeList[e].mst:

                # (x,y) locations of a pair points of edge
                p1 = g.edgeList[e].vertices[0].location
                p2 = g.edgeList[e].vertices[1].location

                # pick a point to get the color = connected_component
                c = g.edgeList[e].vertices[0].connected_component

                # get all the x's and y's separate
                x,y = zip(p1,p2) # or zip(*(p1,p2))

                #print(e)
                #print(x)
                #print(y)
                #print("**")
                plt.plot(x,y, linestyle = '-', color = colors[c])
        else:
            # (x,y) locations of a pair points of edge
            p1 = g.edgeList[e].vertices[0].location
            p2 = g.edgeList[e].vertices[1].location

            # pick a point to get the color = connected_component
            c = g.edgeList[e].vertices[0].connected_component

            # get all the x's and y's separate
            x,y = zip(p1,p2) # or zip(*(p1,p2))

            #print(e)
            #print(x)
            #print(y)
            #print("**")
            plt.plot(x,y, linestyle = '-', color = colors[c])


    plt.show()


#%% Test
g = randomGraph(10,5)
g.createAdjacecnyList2()
findConnected_Components(g)
createConnectedComponentDict(g)

print(g.adjacencyList2)
print(g.connected_components)
print("**Key: connected_component, value: list of vertices**")
print(g.connected_components2)
plotGraph(g)


#%% Exercise 4 ##############################################################

#%% Function

def findMST(graph):
```

```
        """
        Given one (weighted) graph, function adds a boolean property mst to each
        Edge that is True if the edge is part of the mst and False if it is not.


        @ param: graph object
        @ return: none

        Note that the concept of  MST only applies to connected graphs.
        For disconnected graph, the conecpt of minimun spanning forest applies
        (union of minimum spanning trees for its connected components)

        Thus, the function only applies to connected graphs
        """

        if not graph.connected:
            raise RuntimeError("Cannot find MST for not connected gaph")

        V_all = set(range(len(graph.vertexList)))
        V_assigned = set()
        E_assigned = set()

        # pick a random vertex to start
        V_assigned.add(random.choice(list(V_all)))
        V_unassigned = V_all.difference(V_assigned)

        # create an adjacency set that gets updated as edges are added to mst
        # so that avoid scanning these edges. Also uses sets instead of
        # lists as value for search and removal efficiecny
        # k: vertex, value: set of adjacent vertices
        adjacencySet = dict()
        for k,v in g.adjacencyList2.items():
            adjacencySet[k]= set(v)

        # loop until all vertices have been assigned
        while (bool(V_unassigned)):

            minWeight = float( "inf")
            minV = None
            minE = None

            #print("Assigned:" + str(V_assigned))
            #print("UnAssigned:" + str(V_unassigned))
            #print("EAssigned:" + str(E_assigned))

            # from all vertices in assigned set
            for v1 in V_assigned:

                #print("v1:" + str(v1))

                # look for connected edges
                for v2 in adjacencySet[v1]:

                    #print("v2:" + str(v2))

                    # with vertex in unassigned set
                    if v2 in V_unassigned:

                        edge = tuple(sorted((v1,v2)))
                        edgeWeight = graph.adjacencyList[edge]

```

```python
                    #print("edge:" + str(edge))
                    #print("edgeWeight:" + str(edgeWeight))
                    #print("minweigth:" + str(minWeight))

                    # update temporary min weight and min vertex
                    if edgeWeight < minWeight:

                        minWeight = edgeWeight
                        minV = v2
                        minE = edge

                        #print("minweigth:" + str(minWeight))
                        #print("minV:" + str(minV))
                        #print("edge:" + str(edge))

        # after min edge found, update sets
        V_assigned.add(minV)
        V_unassigned.remove(minV)
        E_assigned.add(minE)

        #print("minE: " + str(minE))

        # remove edge from set
        #adjacencySet[minE[0]].discard(minE[1])
        #adjacencySet[minE[1]].discard(minE[0])

        #print(adjacencySet)

        #print("**********")

    # set mst attribute of edge

    for pair,E in graph.edgeListPair.items():
        if pair in E_assigned:
            E.mst = True
        else:
            E.mst = False

    #print("MST: " + str(E_assigned) )


#%% Test

g = randomGraph(7,7, connected=True)
g.createAdjacecnyList2()
findConnected_Components(g)
createConnectedComponentDict(g)
createEdgeListPair(g)

print(g.adjacencyList2)
print(g.connected_components)
print("**Key: connected_component, value: list of vertices**")
print(g.connected_components2)
plotGraph(g)

findMST(g)
print(g.edgeListPair.keys())

for pair, E in g.edgeListPair.items():
    print (str(pair) + ": " + str(E.mst))

```

```python
#%% Scalability test


"""
Plots the average time vs # vertices for function f by creating
a random graph in each repetition

@param f:  function to test with argument = graph
@param vertices: number of vertices of graph (3*n edges)
@param nreps: number of repetitions

@return: none (plots)

"""

minsize = 1000
maxsize = 10000+minsize
stepsize = minsize
vertices = list(range(minsize,maxsize,stepsize))
f =  findMST
nreps = 1 # do one rep because time very consistent for large graphs

avg_time = []

# iterate for different size of graphs
for v in vertices:

    totalTime = 0

    #  repetitions random graphs
    for rep in range(0,nreps):

        # create random graph
        g = randomGraph(v,3*v, connected=True)
        g.createAdjacecnyList2()
        createEdgeListPair(g)

        # time findConnected_Components(g)
        timeStamp = time.process_time() # get the current cpu time
        f(g)
        timeLapse = time.process_time() - timeStamp
        totalTime += timeLapse

        print("rep: {}, time: {}".format(rep,timeLapse))

    # store average time
    avg_time.append(totalTime/nreps)

    print('p time: vertices[{0}]={1}'.format(v, totalTime/nreps))



times_list = [vertices,avg_time]

# plot avg time vs # vertices

plt.plot(times_list[0],times_list[1],'-bo')
# Place a legend above this legend, expanding itself to
# fully use the given bounding box.

plt.xlabel("# vertices (# edges = 3*vertices)")
```

```
    plt.ylabel("average time (s)")
    plt.show()

    # complexity of algorithm is O(E+V)*O(LogV) = O((E+V)*LogV) = O(E*LogV)
    # where E = # edges, V = # vertices, for a connected graph, V = O(E)


    #%% Exercise 5 ###############################################################

    g = randomGraph(5,10, connected=True)
    g.createAdjacecnyList2()
    findConnected_Components(g)
    createConnectedComponentDict(g)
    createEdgeListPair(g)

    print(g.adjacencyList2)
    print(g.connected_components)
    print("**Key: connected_component, value: list of vertices**")
    print(g.connected_components2)

    findMST(g)
    plotGraph(g,MST=False)
    plotGraph(g, MST= True)
    print(g.edgeListPair.keys())

    for pair, E in g.edgeListPair.items():
        print (str(pair) + ": " + str(E.mst))



    # plot side by side
    plt.figure(1)
    plt.subplot(121)
    plotGraph(g)

    plt.subplot(122)
    plotGraph(g, MST=True)
    plt.show()

```