

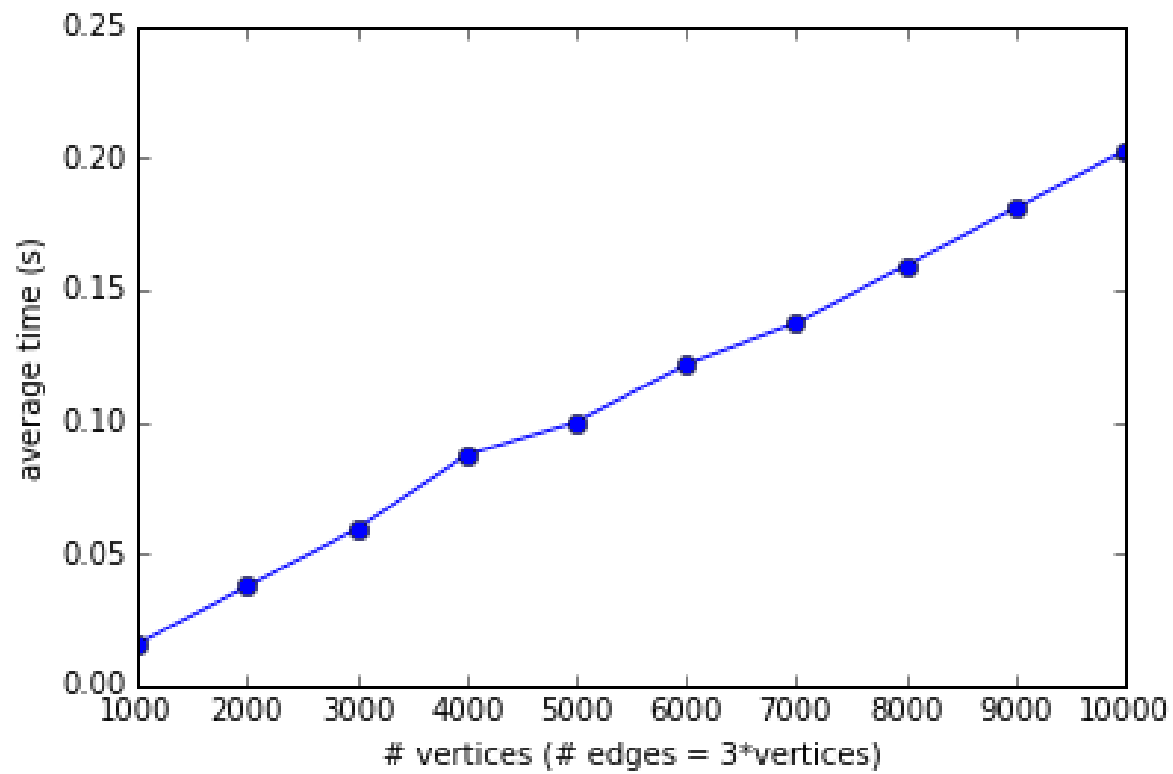
## Assignment 3 – Steven Lin

### Exercise 1

See code

### Exercise 2

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



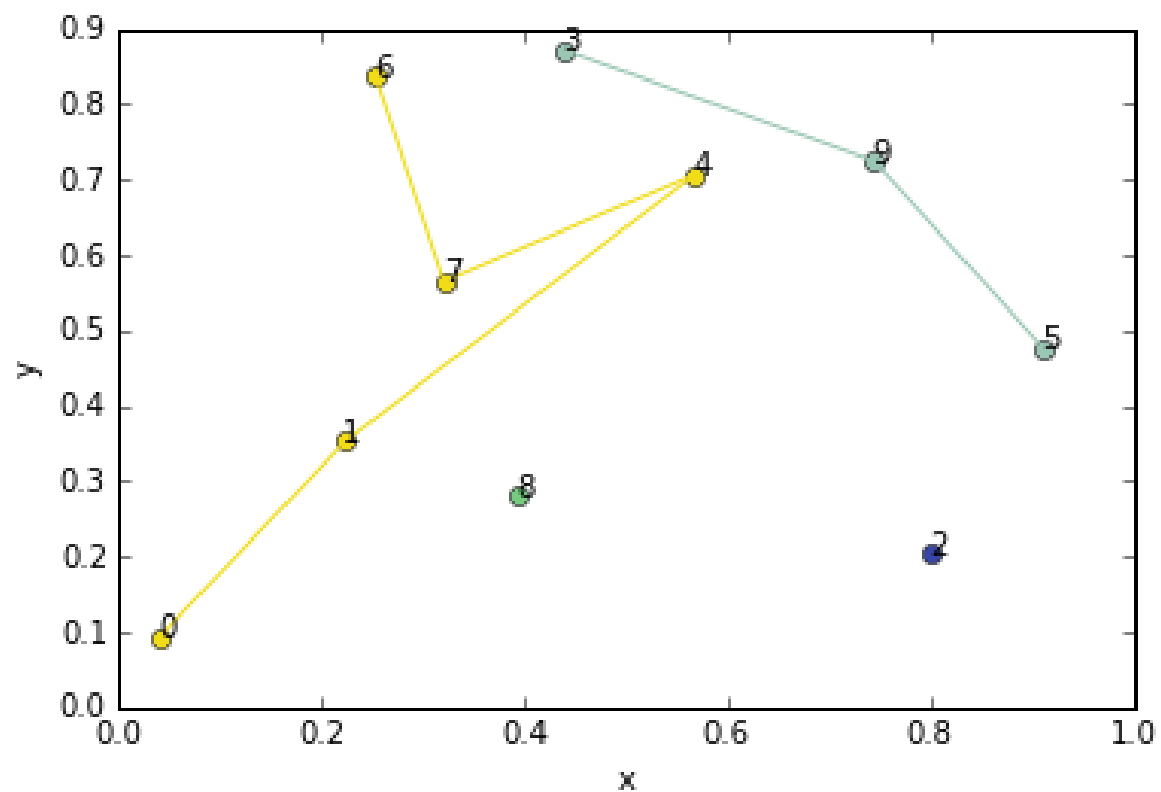
### Exercise 3

See Code

Test:

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

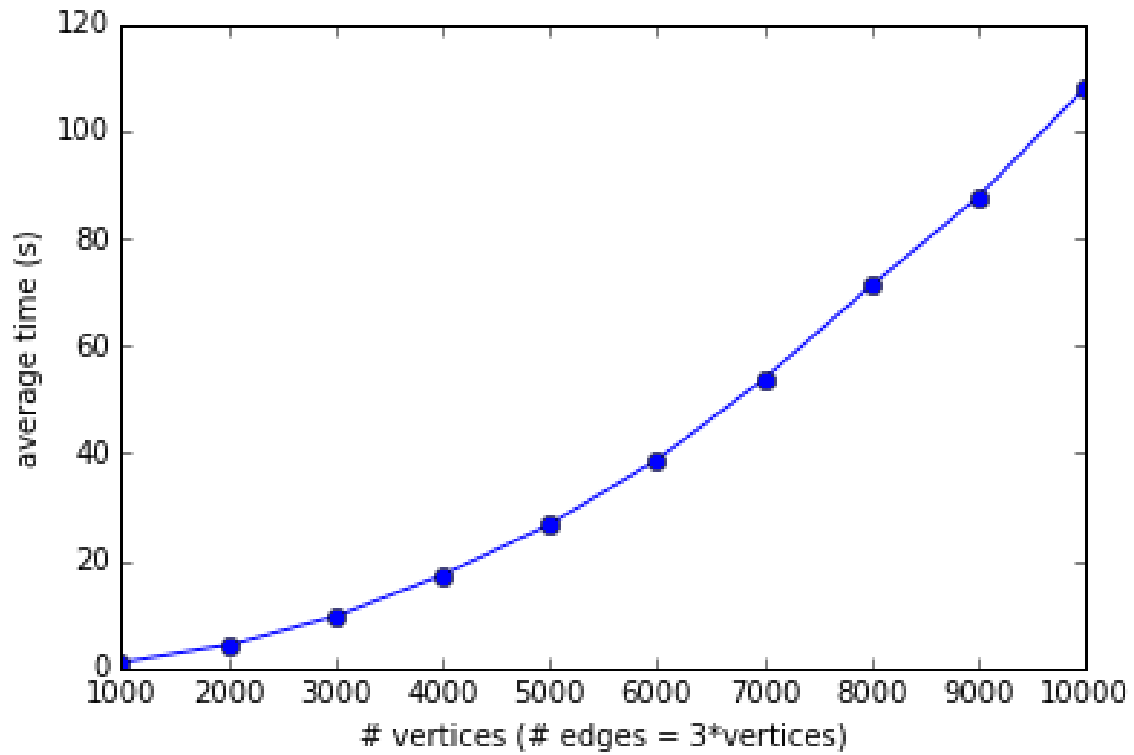
In [273]:



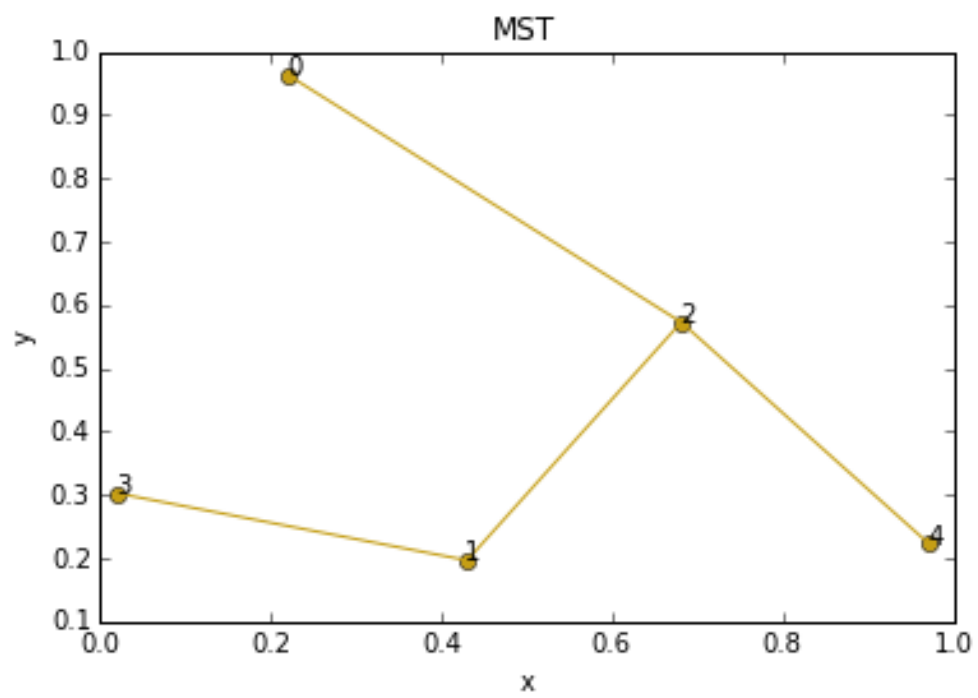
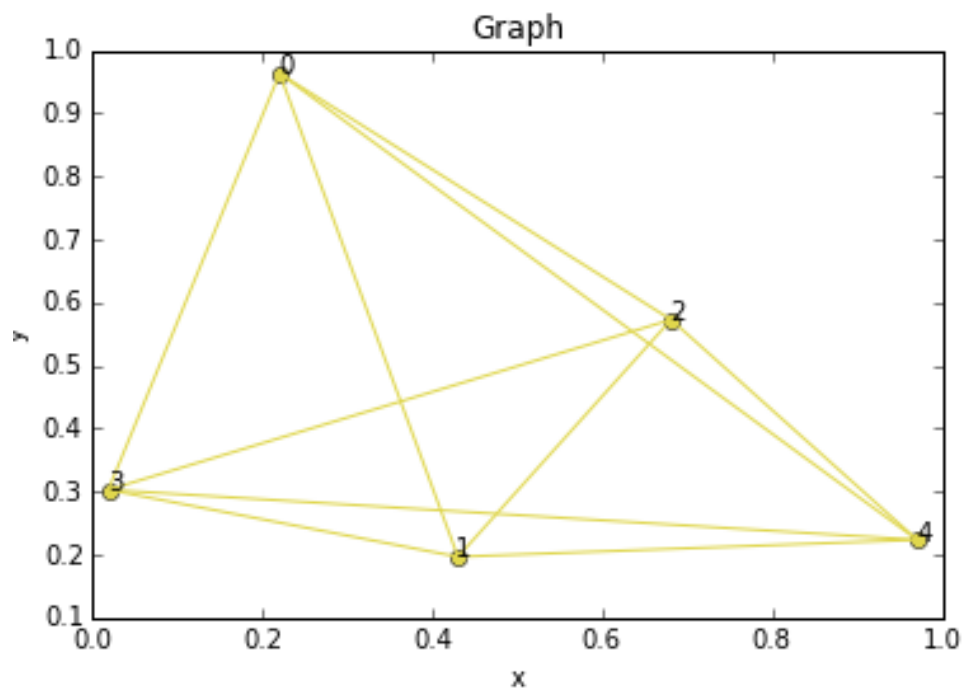
#### Exercise 4

See Code

Complexity:  $O(E+V) \cdot O(\log V) = O((E+V) \cdot \log V) = O(E \cdot \log V)$  where  $E = \#$  edges,  $V = \#$  vertices, for a connected graph,  $V = O(E)$



### Exercise 5



## Code

```
1. # -*- coding: utf-8 -*-
2. """
3. Assignment 3
4. Steven Lin
5. 4/20/15
6.
7. Python: 3.4
8. Editor: Spyder
9. """
10.
11. import random
12. import queue
13. import time # I prefer using time.perf_counter rather than timeit as shown below
14. import matplotlib.pyplot as plt
15. import numpy as np
16. from copy import deepcopy
17.
18. %% Classes #####
19.
20. class Vertex:
21.     """
22.     Vertex class with attributes id and location
23.     Methods: set location
24.     """
25.
26.     def __init__(self, id):
27.         self.id = id
28.         self.location = (None, None) # x,y coordinates
29.
30.     def setLocation(self, x, y):
31.         self.location = (x, y)
32.
33. class Edge:
34.     """
35.     Edge class with attributes id, pair of vertices, weight (default = 1)
36.     """
37.
38.     def __init__(self, id, V1, V2, weight = 1):
39.         self.id = id
40.         self.vertices = (V1, V2) # Vertices objects
41.         self.weight = weight # default value of if not given
42.
43. class Graph:
44.     """
45.     Edge graph with attributes id
46.     Contains adjacency list (key: (Vertex id, Vertex id), value: weight)
47.     dictionary for vertices and edges where key is id and value object
48.     """
49.     def __init__(self, id=None):
50.
51.         self.id = id
52.         self.adjacencyList = dict() # key: (Vertex id, Vertex id) value= wt
53.         self.vertexList = dict() # key: id vertex, # value: vertex object
54.         self.edgeList = dict() # key: id edge, # value: edge object
55.         #self.connected = False
56.
57.     def createAdjacencyList2(self):
```

```

58.         """
59.
60.         Method creates adjcacency list (dictionary) where
61.         #key: (vertex id), value: list adj vertex
62.
63.         Takes no input and does not return anything
64.
65.
66.         """
67.         self.adjacencyList2 = dict()
68.         # note this list does not include vertices that are not
69.         # connected to any vertices
70.
71.         pairs = list(self.adjacencyList.keys())
72.
73.         for v1,v2 in pairs:
74.
75.             # does not exist, so add key and value
76.             if v1 not in self.adjacencyList2:
77.                 self.adjacencyList2[v1] = [v2]
78.
79.             # already exists, so update list of the key
80.             else:
81.                 self.adjacencyList2[v1].append(v2)
82.
83.             # does not exist, so add key and value
84.             if v2 not in self.adjacencyList2:
85.                 self.adjacencyList2[v2] = [v1]
86.
87.             # already exists, so update list of the key
88.             else:
89.                 self.adjacencyList2[v2].append(v1)
90.
91.     %% Functions #####
92.     def findDistance(s,t, p=2):
93.         """
94.         returns the p-norm distance between two points
95.
96.         @param s: coordinates of point 1 (x1,y1,z1...)
97.         @param t: coordiantes of point 2 (x2,y2,z2,..)
98.         @param p: p-norm optional (p = p-norm, p = inf infinity norm, default =2)
99.
100.        """
101.
102.        points = zip(s,t)
103.        dist = 0
104.
105.        # for each corresponding pair coordintates compute distance and sum up
106.
107.        if p == 'inf':
108.            for p1,p2 in points:
109.                dist = max(dist,abs(p1-p2))
110.
111.        else:
112.            for p1,p2 in points:
113.                dist += abs(p1-p2)**p
114.            return dist**(1/p)
115.
116.    def randomPair(s, order=True):
117.        """
118.        Pick random pair of elements from a set

```

```

119.
120.         @ param ls: set of elements (this implies unique elements in the sets)
121.         @ param order: T = order matters, F = doesn't
122.         @ return: tuple with random pair
123.         if order doesn't matter, sort tuple so that return (smallest, largest)
124.
125.         Raise Exception: if the number is less than 2
126.
127.         """
128.         ls = list(s)
129.         if len(ls)<2:
130.             raise RuntimeError("Number of items has to be >=2")
131.         v1 = random.choice(ls) # random value
132.         v2 = random.choice(ls) # random value
133.
134.         # this will not be an infinit loop because unique elements and at least 2
135.         while(v1 == v2 ):
136.             v2 = random.choice(ls)
137.
138.         if order:
139.             return (v1,v2)
140.
141.         return tuple(sorted((v1,v2)))
142.
143.
144.     %% Exercise 1 #####
145.
146.     def randomGraph(v,e, graphID = None, connected = False):
147.         """
148.         Generate a graph object with random locations from (0,1) for (x,y) vertices
149.
150.         @param v: number of vertices
151.         @param e: number of edges
152.         @param graphID: optional ID for graph (Default = None)
153.         @param connected: True = connected graph, False = not required to be
154.         connected (Default)
155.
156.         @return: graph object
157.
158.         """
159.         graph = Graph(graphID)
160.         graph.connected = connected
161.
162.         # check edges does not exceed max number possible
163.         if e > v*(v-1)/2:
164.             raise RuntimeError("e: number of edges has to be <= v*(v-1)/2 ")
165.
166.         # check for connected graph have the min number of edges
167.         if connected and e < v-1:
168.
169.             raise RuntimeError("e: number of edges has to be at least >= v-1")
170.
171.         # create v vertices with id = i
172.         for vertexID in range(v):
173.
174.             # generate random location
175.             x = random.random()
176.             y = random.random()
177.
178.             # create instance and set location
179.             V = Vertex(vertexID)

```

```

180.         V.setLocation(x,y)
181.
182.         # store object key = id, value = vertex object
183.         graph.vertexList[vertexID]=V
184.
185.
186.         allVertices = set(range(v))
187.         assignedVertices = set()
188.         remainingVertices = v
189.
190.         # pick random start vertex
191.         assignedVertices.add(random.choice(list(allVertices)))
192.
193.         # iterate until create e edges:
194.         edgeID = 0
195.         while (edgeID<e):
196.
197.             # random pair of vertices (order in pair of vertices doesn't matter)
198.             # (v1,v2) is the same as (v2,v1), but need to store as one
199.             # unique key for dictionary, so get (smallest, largest) to be
200.             # consistent
201.
202.             if (remainingVertices == 0 or not connected ):
203.                 pair = randomPair(allVertices, order=False)
204.                 v1,v2 = pair
205.
206.             else:
207.                 v1 = random.choice(list(assignedVertices))
208.                 v2 = random.choice(list(allVertices.difference(assignedVertices)))
209.
210.                 pair = tuple(sorted((v1,v2)))
211.
212.             # generate a random pair of vertices until pair does not
213.             # exist in the adjacencyList (as a key in the dictionary)
214.             if (pair not in graph.adjacencyList):
215.
216.                 # get vertex objects and compute weight
217.
218.                 V1 = graph.vertexList[v1]
219.                 V2 = graph.vertexList[v2]
220.                 weight = findDistance(V1.location, V2.location)
221.
222.                 # create edge and add to edge list
223.                 edge = Edge(edgeID, V1,V2, weight)
224.                 graph.edgeList[edgeID] = edge
225.
226.                 # add pair as key and value = euclidean distance
227.                 graph.adjacencyList[pair]= edge.weight
228.
229.                 # remove v1 and v2 from set unassignedVertices if present.
230.                 # no error if not present (might be the case that the pair
231.                 # does not exist, but one of the vertices might have already
232.                 # been removed because was used in a different edge)
233.
234.                 # only do this for connected graph
235.                 if (connected):
236.                     assignedVertices.add(v2)
237.
238.                 remainingVertices = len(allVertices.difference(assignedVertices))
239.         )

```



```

240.             edgeID+= 1 # go to next edge
241.
242.         return graph
243.     ## test not connected ##
244.     g = randomGraph(5,3)
245.     g.createAdjacencyList2()
246.
247.     print(g.vertexList)
248.     print(g.edgeList)
249.
250.     print(" ***** Adjacency List")
251.     print(g.adjacencyList)
252.     print(" ***** Adjacency List2")
253.     print(g.adjacencyList2)
254.
255.     print(" ***** Vertex List")
256.     for V in g.vertexList.values():
257.         print("ID: " + str(V.id) + ", " + "location: " + str(V.location))
258.
259.     print(" ***** Edge List")
260.     for E in g.edgeList.values():
261.         print("ID: " + str(E.id) + ", " + "vertices: " +
262.             str(E.vertices[0].id) + "," + str(E.vertices[1].id) + ", " +
263.             "weight: " + str(E.weight))
264.
265.     v1 = g.edgeList[0].vertices[0].id
266.     v2 = g.edgeList[0].vertices[1].id
267.
268.     print("***** Distance vertex " + str(v1) + " and " + str(v2))
269.     findDistance(g.vertexList[v1].location,g.vertexList[v2].location )
270.
271.     ## test connected ##
272.     g = randomGraph(5,5,connected=True )
273.     g.createAdjacencyList2()
274.
275.     print(g.vertexList)
276.     print(g.edgeList)
277.
278.     print(" ***** Adjacency List")
279.     print(g.adjacencyList)
280.     print(" ***** Adjacency List2")
281.     print(g.adjacencyList2)
282.
283.     print(" ***** Vertex List")
284.     for V in g.vertexList.values():
285.         print("ID: " + str(V.id) + ", " + "location: " + str(V.location))
286.
287.     print(" ***** Edge List")
288.     for E in g.edgeList.values():
289.         print("ID: " + str(E.id) + ", " + "vertices: " +
290.             str(E.vertices[0].id) + "," + str(E.vertices[1].id) + ", " +
291.             "weight: " + str(E.weight))
292.
293.     v1 = g.edgeList[0].vertices[0].id
294.     v2 = g.edgeList[0].vertices[1].id
295.
296.     print("***** Distance vertex " + str(v1) + " and " + str(v2))
297.     findDistance(g.vertexList[v1].location,g.vertexList[v2].location )
298.
299.

```

```

300.      %% Exercise 2 #####
301.      %% Functions #####
302.
303.      ....
304.      connected_componentList = dict()
305.      # or dict.fromkeys(keys, None)
306.      unassignedList = {key: None for key in g.vertexList.keys()}
307.      currentGroup = dict()
308.
309.      # loop until unassigned list is empty
310.      groupID = 0
311.      while(bool(unassignedList)):
312.
313.          # pick a vertex
314.          v = list(unassignedList.keys())[0]
315.
316.          # holds vertices that need to be checked
317.          #q = queue.Queue()
318.          #q.put(v)
319.
320.          # check vertex exists in current group
321.          #if (v not in currentGroup):
322.
323.              #while (not q.empty()):
324.
325.                  pendingList = dict()
326.                  pendingList[v]=None
327.
328.                  while(bool(pendingList)):
329.
330.                      #v = q.get(v) # pick vertex in queue and delete from queue
331.
332.                      # pick a vertex
333.                      v = list(pendingList.keys())[0]
334.
335.                      currentGroup[v] = groupID # add vertex to current group
336.                      del unassignedList[v] # delete from unassigned list
337.                      del pendingList[v] # delete from pendingList
338.
339.                      print("processing" + str(v))
340.
341.                      # for all adjacent vertices that have not been assigned and
342.                      # are not in the pending list
343.
344.                      if v in g.adjacencyList2:
345.                          for i in g.adjacencyList2[v]:
346.                              if ((i not in pendingList) and (i in unassignedList)):
347.                                  print("adding to queue"+ str(i))
348.                                  pendingList[i] = None
349.                                  #q.put(i)
350.
351.                      print("finished group" + str(groupID))
352.
353.                      groupID +=1
354.                      # check if adjacent vertices are in currentGroup and add to queue
355.
356.              ...
357.
358.
359.      def findConnected_Components(graph):

```

```

360.         """
361.         Adds an integer property connected_component to each vertex so that
362.         vertexes in the same connected component have the same value and vertexes
363.         in different connected components have different values
364.
365.         @return: None
366.
367.         """
368.
369.
370.     unassignedSet = set(graph.vertexList.keys()) # set with all vertices not in
group
371.     foundVertices = set() # queue OR assigned vertices included
372.     # loop until unassigned set is empty
373.     groupID = 0
374.     while(bool(unassignedSet)):
375.
376.         v = unassignedSet.pop() # pick an arbitrary vertex (removevs it so...)
377.
378.         unassignedSet.add(v) # add it back in
379.
380.         q = queue.Queue() # holds vertices that need to be checked
381.         q.put(v) # add vertex to queue
382.         #foundVertices[v] =None # add vertex to found list
383.         foundVertices.add(v)
384.
385.         # loop until no items in queue
386.         while (not q.empty()):
387.
388.             v = q.get(v) # pick vertex in queue and delete from queue
389.
390.             graph.vertexList[v].connected_component = groupID # assign group
391.             unassignedSet.remove(v) # remove from unassigned set
392.
393.             ##print("processing: " + str(v))
394.
395.             # for all adjacent vertices that have not been assigned and
396.             # are not in the pending list
397.
398.             # get adjacent vertices, if none then only 1 vertex in group
399.             # so move to another item in the unassigned set
400.
401.             if v in graph.adjacencyList2:
402.
403.                 # check every adjacent vertex
404.                 for i in graph.adjacencyList2[v]:
405.
406.                     # only add to queue if vertex has not been found
407.                     if (i not in foundVertices):
408.                         q.put(i) # add vertex to queue
409.                         foundVertices.add(i) # add vertex to found list
410.                         ##print("adding to queue: "+ str(i))
411.
412.             ##print("finished group: " + str(groupID))
413.             groupID +=1
414.
415. def createConnectedComponentDict(graph):
416.     """
417.     Creates connectedComponent dictionary where key = vertexID, value = groupID
418.
419.     @ param: graph with connected_components attribute

```

```

419.         @ return: dictionary key = vertexID, value = groupID
420.
421.         """
422.         graph.connected_components = {}
423.
424.         for V in graph.vertexList.values():
425.             graph.connected_components[V.id] = V.connected_component
426.
427.         graph.connected_components2 = invertDictionary(graph.connected_components)
428.
429.
430.
431.     def invertDictionary(dic):
432.         """
433.         Invers a dictionary k,v to v,k
434.         Function works for cases when values in dictionary not unique
435.
436.         @param: dictionary
437.         @return: inverted dictionary
438.         """
439.         inv_dic = {}
440.         for k,v in dic.items():
441.             # if value has not been added as a key in inverse dic, then
442.             # set the value in the inv dic an empty list and then append key
443.             # as value in the inverted dictionary
444.             inv_dic[v] = inv_dic.get(v, [])
445.             inv_dic[v].append(k)
446.
447.         return inv_dic
448.
449.     def createEdgeListPair(graph):
450.         """
451.         Add a dictionary key = (vertex ID, vertex ID), value = Edge Object
452.         to graph
453.
454.         @param: graph
455.         @return: none (add a dictionary to graph)
456.
457.         """
458.         g.edgeListPair = {}
459.
460.         for E in g.edgeList.values():
461.             pair = tuple(sorted((E.vertices[0].id, E.vertices[1].id)))
462.             g.edgeListPair[pair] = E
463.
464.
465.         ### test function
466.         g = randomGraph(7,4)
467.         g.createAdjacencyList2()
468.         findConnected_Components(g)
469.
470.         createConnectedComponentDict(g)
471.
472.         print(g.adjacencyList2)
473.         print(g.connected_components)
474.         print(g.connected_components2)
475.
476.         ### Scalability Function
477.
478.     def findTimes(f,vertices,nreps):
479.         """

```

```

480.         Plots the average time vs # vertices for function f by creating
481.         a random graph in each repetition
482.
483.         @param f: function to test with argument = graph
484.         @param vertices: number of vertices of graph (3*n edges)
485.         @param nreps: number of repetitions
486.
487.         @return: none (plots)
488.
489.         """
490.
491.         avg_time = []
492.
493.         # iterate for different size of graphs
494.         for v in vertices:
495.
496.             totalTime = 0
497.
498.             # repetitions random graphs
499.             for rep in range(0,nreps):
500.
501.                 # create random graph
502.                 g = randomGraph(v,3*v, connected=True)
503.                 g.createAdjacencyList2()
504.                 createEdgeListPair(g)
505.
506.                 # time findConnected_Components(g)
507.                 timeStamp = time.process_time() # get the current cpu time
508.                 f(g)
509.                 timeLapse = time.process_time() - timeStamp
510.                 totalTime += timeLapse
511.
512.                 print("rep: {}, time: {}".format(rep,timeLapse))
513.
514.             # store average time
515.             avg_time.append(totalTime/nreps)
516.
517.             print('p time: vertices[{0}]=1}'.format(v, totalTime/nreps))
518.
519.         return [vertices,avg_time]
520.
521.
522.
523.     %% Scalability Test
524.
525.     minsize = 1000
526.     maxsize = 10000+minsize
527.     stepsize = minsize
528.     vertices = list(range(minsize,maxsize,stepsize))
529.     # vertices = [10000]
530.     f = findConnected_Components
531.     nreps = 5
532.     times_list = findTimes(f,vertices,nreps)
533.
534.     # plot avg time vs # vertices
535.
536.     plt.plot(times_list[0],times_list[1],'-bo')
537.     # Place a legend above this legend, expanding itself to
538.     # fully use the given bounding box.
539.
540.     plt.xlabel("# vertices (# edges = 3*vertices)")

```

```

541.     plt.ylabel("average time (s)")
542.     plt.show()
543.
544.     # complexity of algorithm is linear ( $O(V)$ ) ,  $V$  = Number of Vertices
545.
546.     """ Exercise 3 """
547.     """ Function """
548.
549.     def plotGraph(g, MST=False):
550.         """
551.             Plots a graph with vertices and edges and colored connected components
552.             Plots the mst if argument is given (edges need to have mst attribute)
553.
554.             @param g: graph object
555.             @return: none (plots)
556.
557.             """
558.
559.             # get the x and y coordinates of vertices
560.             vertexLocations = []
561.
562.             for V in g.vertexList.values():
563.                 vertexLocations.append(V.location)
564.
565.             x,y = list(zip(*vertexLocations))
566.
567.             colors = deepcopy(g.connected_components2)
568.
569.             # assign a random color to each connected_component
570.             for k in colors:
571.                 colors[k] = (random.random(), random.random(), random.random())
572.
573.             #N = len(g.vertexList)
574.             #colors = np.random.rand(N)
575.             #colors = list(cc.values()) # colors = corresponding connected_components
576.
577.             # (r,g,b) where values between 0 and 1
578.
579.             #fig, ax = plt.subplots()
580.
581.             #plt.xlim(0,1.1)
582.             #plt.ylim(0,1.1)
583.             plt.xlabel("x")
584.             plt.ylabel("y")
585.
586.             if MST:
587.                 plt.title("MST")
588.             else:
589.                 plt.title("Graph")
590.             #ax.scatter(x, y, c=colors, s = area, alpha=0.9)
591.
592.             # plot points and labels
593.             for i in g.vertexList:
594.                 plt.plot(x[i],y[i], marker = 'o',
595.                         color = colors[g.connected_components[i]])
596.                 #print(i)
597.                 #print(x[i])
598.                 #print(y[i])
599.                 #print("***")
600.                 plt.annotate(i, (x[i],y[i]))
601.

```

```

602.         # plot edges
603.         for e in g.edgeList:
604.
605.             if MST:
606.                 # plot only components in mst
607.                 if g.edgeList[e].mst:
608.
609.                     # (x,y) locations of a pair points of edge
610.                     p1 = g.edgeList[e].vertices[0].location
611.                     p2 = g.edgeList[e].vertices[1].location
612.
613.                     # pick a point to get the color = connected_component
614.                     c = g.edgeList[e].vertices[0].connected_component
615.
616.                     # get all the x's and y's separate
617.                     x,y = zip(p1,p2) # or zip(*(p1,p2))
618.
619.                     #print(e)
620.                     #print(x)
621.                     #print(y)
622.                     #print("***")
623.                     plt.plot(x,y, linestyle = '-', color = colors[c])
624.             else:
625.                 # (x,y) locations of a pair points of edge
626.                 p1 = g.edgeList[e].vertices[0].location
627.                 p2 = g.edgeList[e].vertices[1].location
628.
629.                 # pick a point to get the color = connected_component
630.                 c = g.edgeList[e].vertices[0].connected_component
631.
632.                 # get all the x's and y's separate
633.                 x,y = zip(p1,p2) # or zip(*(p1,p2))
634.
635.                 #print(e)
636.                 #print(x)
637.                 #print(y)
638.                 #print("***")
639.                 plt.plot(x,y, linestyle = '-', color = colors[c])
640.
641.
642.         plt.show()
643.
644.
645.         ### Test
646.         g = randomGraph(10,5)
647.         g.createAdjacencyList2()
648.         findConnected_Components(g)
649.         createConnectedComponentDict(g)
650.
651.         print(g.adjacencyList2)
652.         print(g.connected_components)
653.         print("***Key: connected_component, value: list of vertices**")
654.         print(g.connected_components2)
655.         plotGraph(g)
656.
657.
658.         ### Exercise 4 #####
659.
660.         ### Function
661.
662.         def findMST(graph):

```

```

663.         """
664.         Given one (weighted) graph, function adds a boolean property mst to each
665.         Edge that is True if the edge is part of the mst and False if it is not.
666.
667.
668.         @ param: graph object
669.         @ return: none
670.
671.         Note that the concept of MST only applies to connected graphs.
672.         For disconnected graph, the concept of minimum spanning forest applies
673.         (union of minimum spanning trees for its connected components)
674.
675.         Thus, the function only applies to connected graphs
676.         """
677.
678.         if not graph.connected:
679.             raise RuntimeError("Cannot find MST for not connected graph")
680.
681.         V_all = set(range(len(graph.vertexList)))
682.         V_assigned = set()
683.         E_assigned = set()
684.
685.         # pick a random vertex to start
686.         V_assigned.add(random.choice(list(V_all)))
687.         V_unassigned = V_all.difference(V_assigned)
688.
689.         # create an adjacency set that gets updated as edges are added to mst
690.         # so that avoid scanning these edges. Also uses sets instead of
691.         # lists as value for search and removal efficiency
692.         # k: vertex, value: set of adjacent vertices
693.         adjacencySet = dict()
694.         for k,v in g.adjacencyList2.items():
695.             adjacencySet[k]= set(v)
696.
697.         # loop until all vertices have been assigned
698.         while (bool(V_unassigned)):
699.
700.             minWeight = float( "inf")
701.             minV = None
702.             minE = None
703.
704.             #print("Assigned:" + str(V_assigned))
705.             #print("UnAssigned:" + str(V_unassigned))
706.             #print("EAssigned:" + str(E_assigned))
707.
708.             # from all vertices in assigned set
709.             for v1 in V_assigned:
710.
711.                 #print("v1:" + str(v1))
712.
713.                 # look for connected edges
714.                 for v2 in adjacencySet[v1]:
715.
716.                     #print("v2:" + str(v2))
717.
718.                     # with vertex in unassigned set
719.                     if v2 in V_unassigned:
720.
721.                         edge = tuple(sorted((v1,v2)))
722.                         edgeWeight = graph.adjacencyList[edge]
723.

```



```

724.             #print("edge:" + str(edge))
725.             #print("edgeWeight:" + str(edgeWeight))
726.             #print("minweight:" + str(minWeight))
727.
728.             # update temporary min weight and min vertex
729.             if edgeWeight < minWeight:
730.
731.                 minWeight = edgeWeight
732.                 minV = v2
733.                 minE = edge
734.
735.                 #print("minweight:" + str(minWeight))
736.                 #print("minV:" + str(minV))
737.                 #print("edge:" + str(edge))
738.
739.             # after min edge found, update sets
740.             V_assigned.add(minV)
741.             V_unassigned.remove(minV)
742.             E_assigned.add(minE)
743.
744.             #print("minE: " + str(minE))
745.
746.             # remove edge from set
747.             #adjacencySet[minE[0]].discard(minE[1])
748.             #adjacencySet[minE[1]].discard(minE[0])
749.
750.             #print(adjacencySet)
751.
752.             #print("*****")
753.
754.             # set mst attribute of edge
755.
756.             for pair,E in graph.edgeListPair.items():
757.                 if pair in E_assigned:
758.                     E.mst = True
759.                 else:
760.                     E.mst = False
761.
762.             #print("MST: " + str(E_assigned) )
763.
764.
765.             ##% Test
766.
767.             g = randomGraph(7,7, connected=True)
768.             g.createAdjacencyList2()
769.             findConnected_Components(g)
770.             createConnectedComponentDict(g)
771.             createEdgeListPair(g)
772.
773.             print(g.adjacencyList2)
774.             print(g.connected_components)
775.             print("**Key: connected_component, value: list of vertices**")
776.             print(g.connected_components2)
777.             plotGraph(g)
778.
779.             findMST(g)
780.             print(g.edgeListPair.keys())
781.
782.             for pair, E in g.edgeListPair.items():
783.                 print (str(pair) + ": " + str(E.mst))
784.

```

```

785.     %% Scalability test
786.
787.
788.     """
789.     Plots the average time vs # vertices for function f by creating
790.     a random graph in each repetition
791.
792.     @param f: function to test with argument = graph
793.     @param vertices: number of vertices of graph (3*n edges)
794.     @param nreps: number of repetitions
795.
796.     @return: none (plots)
797.
798.     """
799.
800.     minsize = 1000
801.     maxsize = 10000+minsize
802.     stepsize = minsize
803.     vertices = list(range(minsize,maxsize,stepsize))
804.     f = findMST
805.     nreps = 1 # do one rep because time very consistent for large graphs
806.
807.     avg_time = []
808.
809.     # iterate for different size of graphs
810.     for v in vertices:
811.
812.         totalTime = 0
813.
814.         # repetitions random graphs
815.         for rep in range(0,nreps):
816.
817.             # create random graph
818.             g = randomGraph(v,3*v, connected=True)
819.             g.createAdjacencyList2()
820.             createEdgeListPair(g)
821.
822.             # time findConnected_Components(g)
823.             timeStamp = time.process_time() # get the current cpu time
824.             f(g)
825.             timeLapse = time.process_time() - timeStamp
826.             totalTime += timeLapse
827.
828.             print("rep: {}, time: {}".format(rep,timeLapse))
829.
830.         # store average time
831.         avg_time.append(totalTime/nreps)
832.
833.         print('p time: vertices[{}]={}'.format(v, totalTime/nreps))
834.
835.
836.
837.     times_list = [vertices,avg_time]
838.
839.     # plot avg time vs # vertices
840.
841.     plt.plot(times_list[0],times_list[1],'-bo')
842.     # Place a legend above this legend, expanding itself to
843.     # fully use the given bounding box.
844.
845.     plt.xlabel("# vertices (# edges = 3*vertices)")

```

```

846.     plt.ylabel("average time (s)")
847.     plt.show()
848.
849.     # complexity of algorithm is  $O(E+V)*O(\log V) = O((E+V)*\log V) = O(E*\log V)$ 
850.     # where E = # edges, V = # vertices, for a connected graph,  $V = O(E)$ 
851.
852.
853.     ### Exercise 5 #####
854.
855.     g = randomGraph(5,10, connected=True)
856.     g.createAdjacencyList2()
857.     findConnected_Components(g)
858.     createConnectedComponentDict(g)
859.     createEdgeListPair(g)
860.
861.     print(g.adjacencyList2)
862.     print(g.connected_components)
863.     print("***Key: connected_component, value: list of vertices**")
864.     print(g.connected_components2)
865.
866.     findMST(g)
867.     plotGraph(g,MST=False)
868.     plotGraph(g, MST= True)
869.     print(g.edgeListPair.keys())
870.
871.     for pair, E in g.edgeListPair.items():
872.         print (str(pair) + ": " + str(E.mst))
873.
874.
875.
876.     # plot side by side
877.     plt.figure(1)
878.     plt.subplot(121)
879.     plotGraph(g)
880.
881.     plt.subplot(122)
882.     plotGraph(g, MST=True)
883.     plt.show()
884.
885.

```