CS 330: Assignment 3

Due 7PM, Thursday, February 25

February 21, 2016

Collaborators	Kristel Tan (ktan@bu.edu), Johnson Lam (jlam17@bu.edu), & Andrew Lee (teayoon@bu.edu)

Question 4

Source Code

```python
import networkx as nx
import random
import numpy as np
from networkx.utils import UnionFind
from itertools import combinations

# completeGraph() creates a complete undirected graph with weighted edges
# and returns the graph

def completeGraph(n):

    # Create complete undirected graph with n nodes and (n choose 2)
    # edges using networkx complete_graph library function
    g = nx.complete_graph(n)

    # Iterate through all edges and assign each one a random weight
    # uniformly distributed from [0, 1]
    for (u, v) in g.edges():
        rand_weight = random.uniform(0, 1)
        g.add_edge(u, v, weight=rand_weight)

    return g

# squareGraph() returns a complete graph where the vertices are
# chosen uniformly inside the unit square and the edges have weights
# which are the Euclidean distance between its vertices.

def squareGraph(n):

    # Generate list of uniformly random coordinates inside a square
    coordinates = []
    for i in range(n):
        coordinates.append((random.uniform(0, 1), random.uniform(0, 1)))

    # Add the list of vertices to a Graph() object
    g = nx.Graph()
    g.add_nodes_from(coordinates)

```

```python
    # Creates edges between all vertices in the graph
    edges = combinations(coordinates, 2)
    g.add_edges_from(edges)

    # Calculates the edge weights
    for (c1,c2) in g.edges():
        u = np.array(c1)
        v = np.array(c2)
        g.add_edge(c1, c2, weight = np.linalg.norm(u-v))

    return g

# kruskal() returns the minimum spanning tree using networkx library
# functions and implements Kruskal's algorithm

def kruskal(g, weight='weight', data=True):

    # Initialize a UnionFind() object to store the spanning edges of the
    # MST
    mst = UnionFind()
    # Sort the edges in ascending order of their weights
    edges = sorted(g.edges(data = True), key = lambda t: t[2].get(weight,
                   1))

    # For each edge (u, v) by ordered weight, check if adding the edge
    # will create a cycle; if not, add to the mst, else discard the edge
    for (u, v, w) in edges:
        if mst[u] != mst[v]:
            yield (u, v, w)
        mst.union(u, v)

# calcSum() iterates through a given minimum spanning tree and
# calculates the sum of the weights in the tree

def calcSum(g, weight='weight', data=True):

    total_weight = 0
    for (u, v, w) in g.edges_iter(data=True):
        total_weight += w['weight']

    return total_weight


graph1 = completeGraph(4)
mst1 = nx.Graph(kruskal(graph1))

print(graph1.edges(data=True), '\n')
print(mst1.edges(data=True), '\n')
print(calcSum(mst1), '\n')

graph2 = squareGraph(4)
mst2 = nx.Graph(kruskal(graph2))

```

```
print(graph2.edges(data=True), '\n')
print(mst2.edges(data=True), '\n')
print(calcSum(mst2), '\n')
```

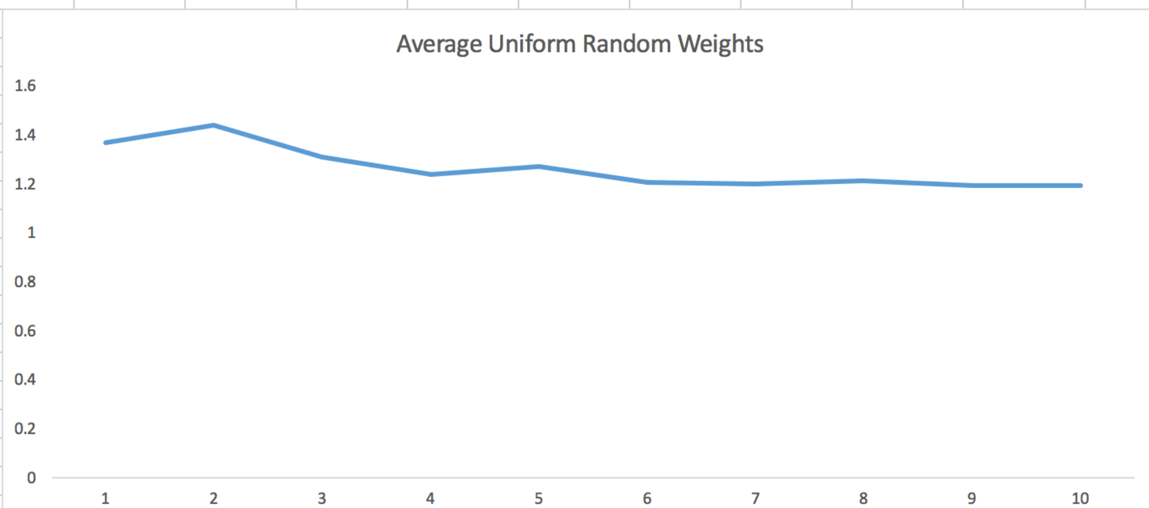
We used the NetworkX Python package to help us implement undirected, complete graphs as *Graph( )* objects. We used Python's *random* library to generate random uniformly distributed numbers for the edge weights and vertex coordinate points for each graph respectively. We also referred to NetworkX's documentation found at https://networkx.github.io/documentation/latest/ to guide us in implementing Kruskal's algorithm in conjunction with the package's data structures and appropriate functions.

The individual functions and calculations have been commented above for further detail about what each component is doing.

Experiment Results and Analysis

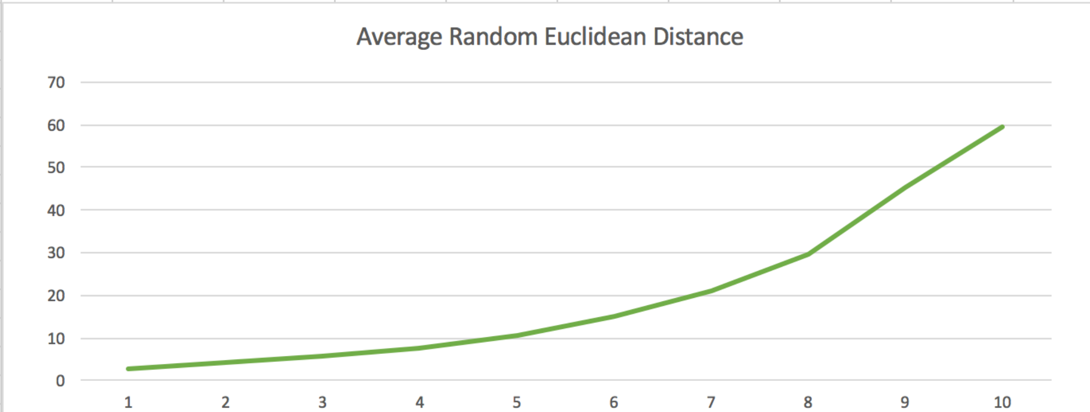
After running the program above for five random graph instances of n vertices = 16, 32, 64, 128, 256, 512, 1024, 2048, 4098, and 8192, we received the following average weights of the minimum spanning trees calculated and used Excel to graph the results. The x-axis represents one of the ten experiments for n vertices, respectively, and the y-axis represents the average total weight of the MST for five runs.

	16	32	64	128	256	512	1024	2048	4096	8192
1	1.4395115	1.5924065	1.3438607	1.4312954	1.352034	1.1400978	1.2041828	1.2069319	1.1868167	1.20818642
2	1.5243729	1.8151159	1.3941878	1.1647861	1.2461123	1.2522462	1.2090716	1.1969906	1.1715286	1.21758945
3	1.2772496	0.8469457	1.2141282	1.0852114	1.1795372	1.2607148	1.2128668	1.231208	1.21796112	1.1453665
4	0.8846516	1.5537813	1.2404677	1.3581879	1.2380589	1.0765875	1.1966877	1.1961496	1.16129985	1.18945204
5	1.6871337	1.3902903	1.3567802	1.1443449	1.3354267	1.2777303	1.1737405	1.2173276	1.2091734	1.20404798
	1.3625839	1.4397079	1.3098849	1.2367652	1.2702338	1.2014753	1.1993099	1.2097215	1.18935594	1.19292848



We estimate the function of this plot to be roughly $f = 1.26$, a constant linear function. If you notice, the last row of averages for each column are roughly the same. The plot for this randomly generated graph is reasonable because although we are continuously adding more edges, their respective weights are also getting smaller and evenly distributed throughout the graph. Therefore, the average weight of the edges in the MST does not change too drastically as more vertices are added.

	16	32	64	128	256	512	1024	2048	4096	8192
1	3.05358	4.1199837	5.3386956	7.5534567	10.567936	14.819342	21.213787	29.095146	59.4343459	59.1615533
2	2.8268835	3.9488072	5.8722542	7.746205	10.946655	14.962452	21.54881	29.81022	41.8923292	59.2957933
3	2.8470545	4.3170148	5.4656924	7.7654498	10.111322	14.994107	20.971972	29.812567	42.03585	59.2501022
4	2.6220298	4.0999752	5.6674555	8.1017107	10.376533	14.855821	21.086476	30.152009	41.8799509	59.3637036
5	2.5289106	3.9868159	5.6861651	7.6033293	10.630976	15.040696	20.849787	29.717173	41.5660227	59.3657146
	2.7756917	4.0945194	5.6060525	7.7540303	10.526685	14.934483	21.134166	29.717423	45.3616997	59.2873734



We estimate the function of this plot to be roughly $f(n) = n^2$, an exponentially increasing graph. Unlike the last plot, as we add more vertices to this graph, the total weight of the MST begins to increase. The plot for this randomly generated graph is also reasonable because instead of uniformly distributing the edge weights, we are uniformly distributing the coordinates of the vertices. Therefore, the weights of the edges between them will not necessarily be as evenly scaled as in the graph for the first experiment. Instead, their weights are determined by the distance between them.