cs630-hw3-coding

October 27, 2023

1 Instructions

- Please convert the notebook into pdf for final submission. If you have issue with directly downloading the file as pdf, you can use web browers's print >> save as pdf.
- If you use languages other than Python, you have to submit your executable code together with a report that contains all the results needed.

2 Triangle counting

A triangle is a graph structure with three-node all connecting to each other. Finding triangles in real world networks is important due to the homogenous meaning it carries. For example, in a facebook network where nodes represent users and edges represent "friend" relations, users connected by triangle structures are more likely to know each other in real life. However, counting triangles can be a computation-heavy task on large graphs, i.e., the best known deterministic algorithm has a complexity of $O(m^{3/2})$. In this homework, we will study a randomized approach that significantly accerlerate the computation, proposed in [1].

[1]. Charalampos E. Tsourakakis, U. Kang, Gary L. Miller, and Christos Faloutsos. 2009. DOULION: counting triangles in massive graphs with a coin. In Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '09). Association for Computing Machinery, New York, NY, USA, 837–846. https://doi.org/10.1145/1557019.1557111

2.1 Preliminaries

2.1.1 Networkx

Networkx is a Python library people commonly use when dealing with graphs. Please follow the instructions to install the package: https://networkx.org/documentation/stable/install.html. A tutorial for quick-start is also available on their webpage: https://networkx.org/documentation/stable/tutorial.html.

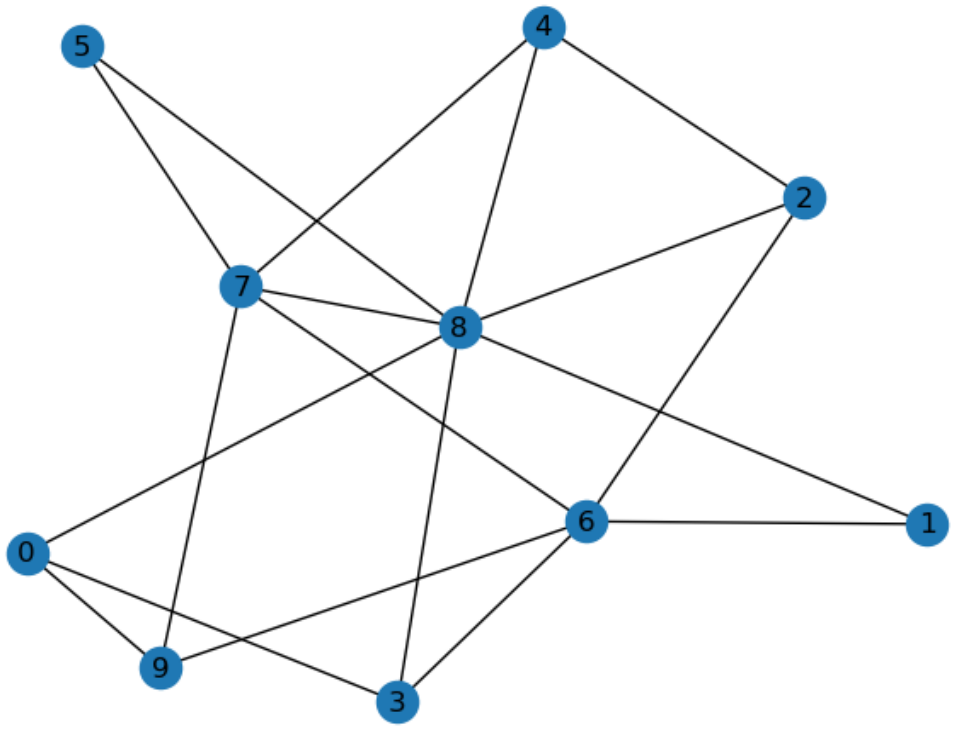
[1]:
```
import networkx as nx
import numpy as np
import pandas as pd
import random, time
import matplotlib.pyplot as plt
```

2.1.2 G(n, p)

$G(n, p)$ is a simple random graph model of the Erdős-Rényi graphs. It generates graphs with n nodes by independently sample the existance of each possible edge by probability p. Networkx provides a convenient way to generate random graphs using built-in functions for various commonly used graph models. To create an Erdős-Rényi graph, we can use the function nx.erdos_renyi_graph(n, p), where n is the number of nodes and p is the probability of an edge between two nodes. The function returns an nx.Graph object containing the nodes and edges of the generated graph.

[2]:
```
# Example: generate a graph from G(10, 0.3)
G_test = nx.erdos_renyi_graph(10, 0.4)
# Check how many nodes and edges it has.
print("the sampled graph contains ", G_test.number_of_nodes(), "nodes and",␣
 ↪G_test.number_of_edges(), "edges")
# Plot the graph
nx.draw(G_test, with_labels=True)
```

```
the sampled graph contains  10 nodes and 18 edges
```



2.1.3 Triangle counting in Networkx

Networkx also provides a function for triangle counting called nx.triangles(G). It takes a nx.Graph object G as input and returns a dictionary of (node id, number of triangles participated) key-value pairs. To calculate the total number of triangles in G, simply sum all the values in the dictionary and divide the result by 3 (since each triangle is counted three times in the dictionary).

[3]:
```
node_triangle_dict = nx.triangles(G_test)
print("Total number of triangles in the test graph:",␣
 ↪sum([node_triangle_dict[i] for i in node_triangle_dict])/3)
print("The triangle participation for each node: ", node_triangle_dict)
```

```
Total number of triangles in the test graph: 5.0
The triangle participation for each node:  {0: 1, 1: 0, 2: 1, 3: 1, 4: 2, 5: 1,
6: 1, 7: 3, 8: 4, 9: 1}
```

2.2 Doulion: a randomized algorithm for triangle counting

2.2.1 Algorithm

The algorithm works as follows. Given a graph G and a biased coin that shows head with probability q, we toss the coin for each edge, keep the edge if we get a head, and delete it otherwise. Count the number of triangles on the new graph G' and multiple by $1/q^3$.

2.2.2 Problem 1.

Assume the number of triangles in G is T, and T' is a random variable that represents the number of triangles in the new graph G'. Prove expected value of T' is $q^3 \cdot T$.

Let's say X_{ij} is an event of edge ij being in G'.

We know that, $Pr(X_{ij}) = q$

Let's say edges ab, bc, ac form a triangle. So,

Pr(Triangle abc existing in G')$ = Pr(X_{ab}, X_{bc}, X_{ac}) = Pr(X_{ab}) * Pr(X_{bc}) * Pr(X_{ac}) = q^3$

We know that T is the number triangles in G and T' is the number of triangles in G'.

$E(T') = Pr$(A triangle existing in G') * Total number of Triangles $ = q^3.T$

2.2.3 Problem 2.

Implement the algorithm in the cell below

[134]:
```
def doulion(G, q):
    '''
    G: nx.Graph.
    q: float, the probability of getting a head in the coin tosses of doulion.

    Return: int, the estimated number of triangles in G.
    '''
```

```
    G_new = nx.Graph()

    for edge in G.edges():
        if random.random() < q:
            G_new.add_edge(*edge)

    node_triangle_dict = nx.triangles(G_new)

    triangles_in_Gnew = sum([node_triangle_dict[i] for i in␣
↪node_triangle_dict])/3

    estimated_triangles = triangles_in_Gnew / (q ** 3)

    return int(estimated_triangles)

doulion(G_test, 0.7)
```

[134]: 5

2.2.4 Problem 3

Now we will test the function on $G(n,p)$ with q values range in $[0.1, 0.2, 0.4, 0.8]$. Since both the graph and the algorithm are randomized, for each q value, you are asked to repeat the following steps 10 times and report the average values of metrics.

1. Sample a graph from $G(n,p)$ with $n = 1000$ and $p = 0.05$.
2. Count the real number of triangles T in the sampled graph.
3. Run doulion to get an estimated number of triangles T'.
4. Return the following metrics: running time for both algorithms, and the accuracy of doulion. The accuracy here is defined as $1 - \frac{|T-T'|}{T}$.

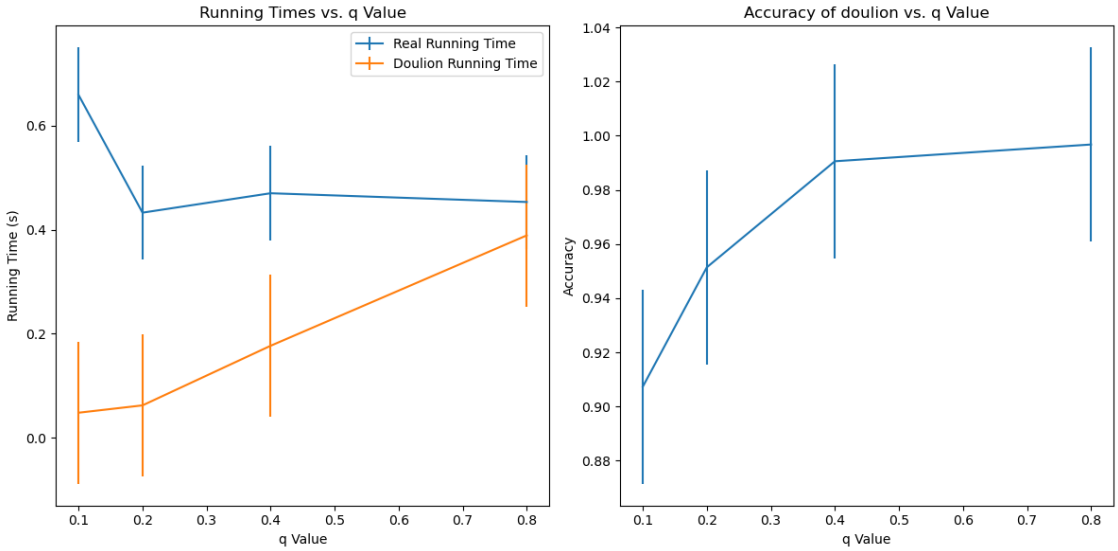
Plot the average values with standard deviation as error bars for different q values. You are asked to generate two plots with the q values being the x axis. The first one should show the change of running times for both methods v.s. q value, and the second one should show the change of accuracy for doulion.

[158]:
```
def generate_graph(n, p):
    return nx.erdos_renyi_graph(n, p)

def ret_real_triangles(G):
    node_triangle_dict = nx.triangles(G)
    return sum([node_triangle_dict[i] for i in node_triangle_dict])/3

def compare(n, p):
    q_values = [0.1, 0.2, 0.4, 0.8]
    real_times = []
    doulion_times = []
```

```
    accuracies = []

    for q in q_values:
        G = generate_graph(n,p)
        T = []
        T_ = []
        real_time = []
        doulion_time = []

        for i in range(10):
            start_time = time.time()
            T.append(ret_real_triangles(G))
            real_time.append(time.time() - start_time)

            start_time = time.time()
            T_.append(doulion(G, q))
            doulion_time.append(time.time() - start_time)

        real_times.append(np.mean(real_time))
        doulion_times.append(np.mean(doulion_time))
        accuracies.append(1-(np.abs(np.mean(T) - np.mean(T_))/np.mean(T)))

    print("Running times for finding Read number of triangles: ", real_times)
    print("Running times for finding Estimated number of triangles: ",␣
↪doulion_times)
    print("Accuracies: ", accuracies)

    plt.figure(figsize=(12, 6))
    plt.subplot(121)
    plt.errorbar(q_values, real_times, yerr=np.std(real_times), label='Real␣
↪Running Time')
    plt.errorbar(q_values, doulion_times, yerr=np.std(doulion_times),␣
↪label='Doulion Running Time')
    plt.xlabel('q Value')
    plt.ylabel('Running Time (s)')
    plt.title('Running Times vs. q Value')
    plt.legend()

    plt.subplot(122)
    plt.errorbar(q_values, accuracies, yerr=np.std(accuracies), label='Running␣
↪Time')
    plt.xlabel('q Value')
    plt.ylabel('Accuracy')
    plt.title('Accuracy of doulion vs. q Value')

    plt.tight_layout()
    plt.show()
```

```
compare(1000, 0.05)
```

```
Running times for finding Read number of triangles:  [0.659394907951355,
0.4327763557434082, 0.4700788974761963, 0.4532521963119507]
Running times for finding Estimated number of triangles:  [0.04815781116485596,
0.0622920036315918, 0.17685563564300538, 0.38905820846557615]
Accuracies:  [0.9073107049608355, 0.9514085556714861, 0.9905715497337795,
0.9967545337429392]
```

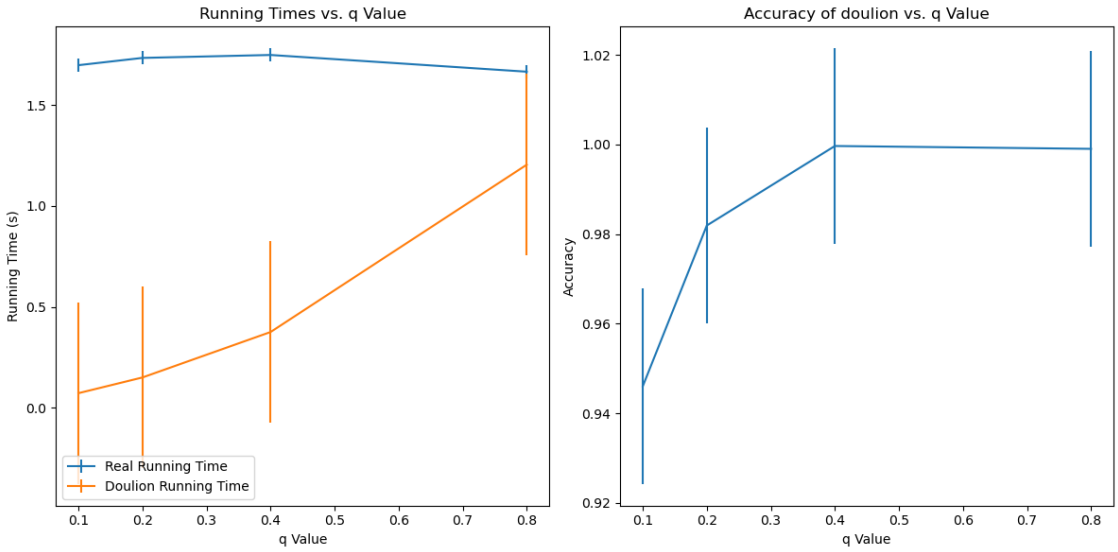


2.2.5 Problem 4

Repeat the experiment like previous, but use $p = 0.1$ for the random graph generation. What do you observe? Can you give an intuitive explaination?

[160]:
```
compare(1000, 0.1)
```

```
Running times for finding Read number of triangles:  [1.698171281814575,
1.73422212600708, 1.748772883415222, 1.6658950567245483]
Running times for finding Estimated number of triangles:  [0.07258572578430175,
0.1505105972290039, 0.37576286792755126, 1.204980731010437]
Accuracies:  [0.9460070593425988, 0.981906664832108, 0.9996549710390199,
0.9990192154918556]
```



Increasing the edge density with a higher value of p will make the graph denser, which makes it easier for the edge sampling method to capture triangles accurately. This leads to higher accuracy in estimations. However, it might come at the cost of longer running times, as there are more edges to consider during the sampling process.