Project 1:
Managing and Mining Complex Networks

George Parnalis Palantzidis - John Vitalis - Eleni Trachanidou

Data and Web Science
Aristotle University of Thessaloniki

Introduction

Objective

In this project, we focus on exact and approximate triangle counting algorithms. You are going to implement and compare some triangle counting algorithms. More specifically, the exact algorithms are: 1) all triplets, 2) node iterator, 3) compact forward. In addition, you will implement the DOULION algorithm which sparsifies the graph and then we can use any of the previous algorithms to approximately count the triangles. Finally, you will implement a streaming algorithm that reads the edges of the graph incrementally. You are going to use the version of the TRIEST algorithm for edge insertions only (you will not implement edge deletions). Below you will find the necessary material.

Overview

In this Project, we developed various methods to calculate the number of total triangles in a graph.
Initially, we provide three different algorithms that calculate the number of triangles in a graph. All three algorithms are exact in their calculation.
The algorithms that were used are:

- **Triplets**: Brute force approach. For every triplet of nodes in the graph, it checks whether the edges that connect them exist or not. Not scalable for large graphs.

- **Node Iterator**: Checks for every node of the graph if it forms any triangles with its neighbours. It iterates through every node in the graph, hence the name.
- **Compact Forward**: A refined version of edge iterator, where a dynamic data structure is used to store the neighbours.

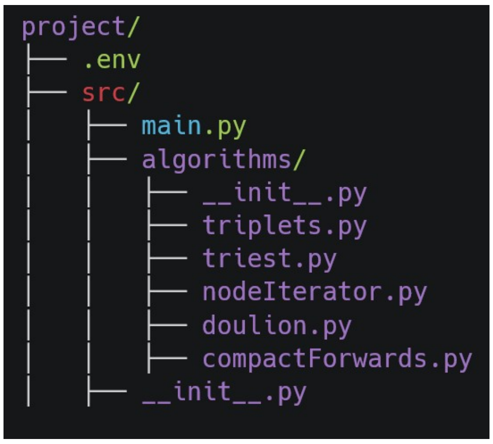
We also use the **DOULION** algorithm to sparsify the initial graph. After the sparsification, any of the previously mentioned algorithms can be applied to the new graph.
Finally, we also use the base **Triest-Base** streaming algorithm, in case we have an issue with how the graph can fit into the main memory. The Triest algorithm gives us the flexibility to count the triangles of a large graph that could not normally fit in memory.

The Triplets algorithm was tested only in a small Erdős–Rényi random graph, due to the high computation time that is required in its brute force approach.

Methodology

Overview

To approach the solution of the problem in Project 1, we structured our Python code into a modular and organized format to ensure clarity, scalability, and ease of maintenance. The project directory is structured as follows:



```
project/
├── .env
├── src/
│   ├── main.py
│   ├── algorithms/
│   │   ├── __init__.py
│   │   ├── triplets.py
│   │   ├── triest.py
│   │   ├── nodeIterator.py
│   │   ├── doulion.py
│   │   ├── compactForwards.py
│   ├── __init__.py
```

Folder Structure:

- The algorithms folder contains all the individual files for the implemented triangle counting algorithms, such as **triplets**, **triest**, **nodeIterator**, **doulion**, and **compactForwards**. Each algorithm is encapsulated in its own Python file for modularity and ease of testing.
- The main.py file serves as the entry point for the project. It orchestrates the execution of the algorithms based on user-defined configurations.
- The .env file is used for environment configuration. It allows the user to specify which algorithms to run (via Boolean flags) and which dataset to use, providing flexibility without requiring code changes.

Tools and Libraries

To implement and solve the problem effectively, the following tools and libraries were utilized:

- **Python 3.x:**
  The project was implemented using Python 3.x, leveraging its simplicity and versatility for algorithm implementation and data processing.
- **NetworkX Library**:
  The NetworkX library was used to manage and process graph data structures. It provided built-in functions for handling nodes, edges, and graph operations, simplifying the implementation of triangle counting algorithms.
- **Input Graph in Edge List Format**:
  The input graphs were provided in edge list format, a standard representation where each line specifies an edge between two nodes. This format was used for easy parsing and loading into the program

Data sets

The two data sets used for our experiments were the *High Energy Physics - Theory collaboration network* and the *Enron email network*, both provided by the Stanford Network Analysis Project.

Implementation

Code Structure

algorithms/triplets.py

Key Steps:

1. **Preprocessing**: Create hash tables of node neighbors for O(1) edge lookups.
2. **Node Filtering**: Only consider nodes with edges to reduce iterations.
3. **Edge Iteration**: For each node u, consider neighbors v where v>u (to avoid duplicate triangles).

Triangle Counting: For each v, check neighbors w where w>v > and verify if w is also a neighbor of v.

```
adj = {node: set(graph.neighbors(node)) for node in graph.nodes()}
for u in nodes:
    u_neighbors = {v for v in adj[u] if v > u}
    for v in u_neighbors:
        w_candidates = {w for w in adj[u] if w > v}
        triangles += sum(1 for w in w_candidates if w in adj[v])
```

algorithms/triest.py

Key Steps:

1. **Initialization**: Set reservoir size M, initialize counters.
2. **Edge Sampling**:

   If t≤M : Add all edges to the reservoir.
   If t>M: Sample edges with probability M/t. Replace a random edge in the reservoir if sampled.
3. **Triangle Updates**:

For each edge (u,v) find common neighbors of u and v in the reservoir.

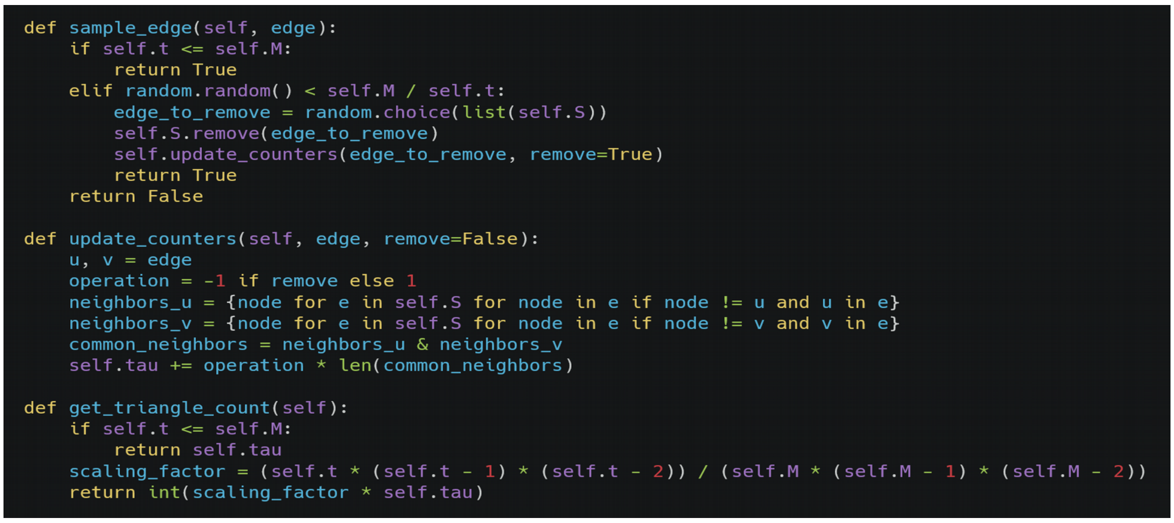
Update the global triangle count (tau) and per-node counts (tau_c).

4. **Scaling Factor**:

   Adjust the triangle count for unsampled edges using the scaling factor:

   Scaling Factor = *t (t−1)(t−2) / M (M−1)(M−2)*

   Final triangle estimate: tau × Scaling Factor



```
def sample_edge(self, edge):
    if self.t <= self.M:
        return True
    elif random.random() < self.M / self.t:
        edge_to_remove = random.choice(list(self.S))
        self.S.remove(edge_to_remove)
        self.update_counters(edge_to_remove, remove=True)
        return True
    return False

def update_counters(self, edge, remove=False):
    u, v = edge
    operation = -1 if remove else 1
    neighbors_u = {node for e in self.S for node in e if node != u and u in e}
    neighbors_v = {node for e in self.S for node in e if node != v and v in e}
    common_neighbors = neighbors_u & neighbors_v
    self.tau += operation * len(common_neighbors)

def get_triangle_count(self):
    if self.t <= self.M:
        return self.tau
    scaling_factor = (self.t * (self.t - 1) * (self.t - 2)) / (self.M * (self.M - 1) * (self.M - 2))
    return int(scaling_factor * self.tau)
```

algorithms/nodeIterator.py

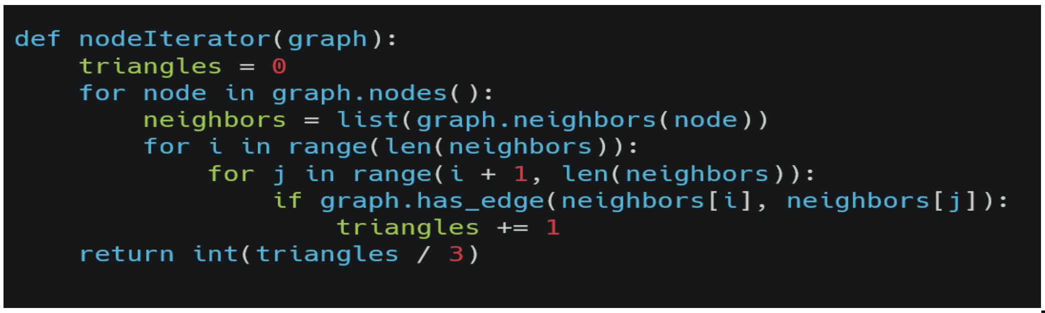
Key Steps:

1. **Neighbor Retrieval**: For each node, retrieve its neighbors for efficient O(1) lookups.
2. **Pairwise Checking**:

   Examine all pairs of neighbors to check if they are connected by an edge. Only consider pairs (i,j) where j>i to avoid duplicate checks.
3. **Triangle Counting:**

   Increment the triangle count for each valid triplet (node and its two connected neighbors).

At the end, divide the total count by 3 to account for overcounting (each triangle is counted once per vertex).



```
def nodeIterator(graph):
    triangles = 0
    for node in graph.nodes():
        neighbors = list(graph.neighbors(node))
        for i in range(len(neighbors)):
            for j in range(i + 1, len(neighbors)):
                if graph.has_edge(neighbors[i], neighbors[j]):
                    triangles += 1
    return int(triangles / 3)
```

algorithms/doulion.py

Key Steps:

1. **Graph Sparsification:**

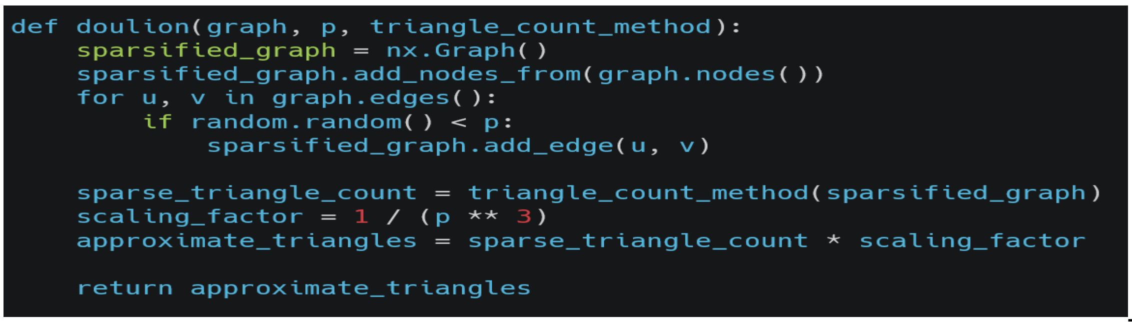
   Randomly sample edges from the original graph with probability p, creating a sparsified graph. This reduces the number of edges and accelerates triangle counting.

2. **Triangle Counting on Sparsified Graph:**

   Use the provided triangle counting method (e.g., Node Iterator, Compact Forward) to count triangles in the sparsified graph.

3. **Scaling :**

   Scale the triangle count by 1/p3 to compensate for the reduction in edges due to sparsification. Each triangle requires 3 edges, and each edge is kept with probability p.



```
def doulion(graph, p, triangle_count_method):
    sparsified_graph = nx.Graph()
    sparsified_graph.add_nodes_from(graph.nodes())
    for u, v in graph.edges():
        if random.random() < p:
            sparsified_graph.add_edge(u, v)

    sparse_triangle_count = triangle_count_method(sparsified_graph)
    scaling_factor = 1 / (p ** 3)
    approximate_triangles = sparse_triangle_count * scaling_factor

    return approximate_triangles
```

algorithms/compactForwards.py

Key Steps:

1. **Node Sorting:**

   Sort nodes by their degree in ascending order. This helps optimize the search for triangles by focusing on lower-degree nodes first.

2. **Re-labeling:**

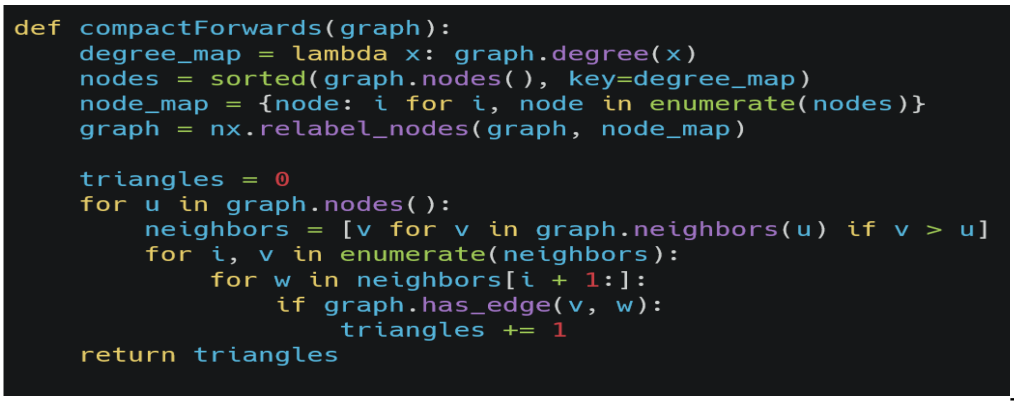
   Re-label nodes to ensure neighbors with higher IDs are considered first. This avoids counting the same triangle multiple times.

3. **Neighbor Filtering:**

   For each node u, filter its neighbors to only include those with higher IDs than u.

4. **Triangle Detection:**

   For each pair of neighbors v and w of u, check if an edge exists between v and w. If so, a triangle is detected.



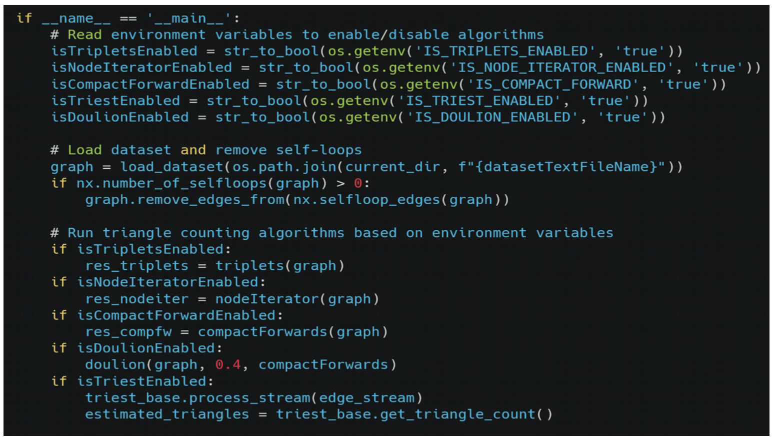
```python
def compactForwards(graph):
    degree_map = lambda x: graph.degree(x)
    nodes = sorted(graph.nodes(), key=degree_map)
    node_map = {node: i for i, node in enumerate(nodes)}
    graph = nx.relabel_nodes(graph, node_map)

    triangles = 0
    for u in graph.nodes():
        neighbors = [v for v in graph.neighbors(u) if v > u]
        for i, v in enumerate(neighbors):
            for w in neighbors[i + 1:]:
                if graph.has_edge(v, w):
                    triangles += 1
    return triangles
```

main.py

Key Steps:

1. **Environment Variable Setup:**
   Read environment variables to enable or disable specific triangle counting algorithms (Triplets, Node Iterator, Compact Forward, DOULION, TRIEST).
2. **Dataset Loading**:
   Load the graph dataset (e.g., "email-Enron.txt") using NetworkX's read_edgelist function.
3. **Self-loop Removal:**
   Remove self-loops from the graph to ensure accurate triangle counting
4. **Actual Triangle Count:**
   Calculate the actual triangle count using NetworkX's triangles function for comparison
5. **Algorithm Execution:**
   Conditionally run the selected algorithms (Triplets, Node Iterator, Compact Forward) and/or the DOULION algorithm for approximate counting.
   If DOULION is enabled, apply it with different triangle counting methods (e.g., Compact Forward, Node Iterator, Triplets).
6. **TRIEST Streaming Algorithm:**
   For the TRIEST algorithm, process the edge stream and estimate the triangle count based on the reservoir sampling technique.



```python
if __name__ == '__main__':
    # Read environment variables to enable/disable algorithms
    isTripletsEnabled = str_to_bool(os.getenv('IS_TRIPLETS_ENABLED', 'true'))
    isNodeIteratorEnabled = str_to_bool(os.getenv('IS_NODE_ITERATOR_ENABLED', 'true'))
    isCompactForwardEnabled = str_to_bool(os.getenv('IS_COMPACT_FORWARD', 'true'))
    isTriestEnabled = str_to_bool(os.getenv('IS_TRIEST_ENABLED', 'true'))
    isDoulionEnabled = str_to_bool(os.getenv('IS_DOULION_ENABLED', 'true'))

    # Load dataset and remove self-loops
    graph = load_dataset(os.path.join(current_dir, f"{datasetTextFileName}"))
    if nx.number_of_selfloops(graph) > 0:
        graph.remove_edges_from(nx.selfloop_edges(graph))

    # Run triangle counting algorithms based on environment variables
    if isTripletsEnabled:
        res_triplets = triplets(graph)
    if isNodeIteratorEnabled:
        res_nodeiter = nodeIterator(graph)
    if isCompactForwardEnabled:
        res_compfw = compactForwards(graph)
    if isDoulionEnabled:
        doulion(graph, 0.4, compactForwards)
    if isTriestEnabled:
        triest_base.process_stream(edge_stream)
        estimated_triangles = triest_base.get_triangle_count()
```

Results

High Energy Physics - Theory collaboration network

Dataset statistics

Nodes: 9877
Edges: 25998
Number of Triangles: 28339

Results on a run

ca-HepTh	Time (s)	Triangle Count	Accuracy
Node Iterator	0.06	28339	100.00
Compact Forward	0.06	28339	100.00
DOULION (p=0.4) Node Iterator	0.01	28609	99.05
DOULION (p=0.4) Compact Forward	0.04	28124	99.24
TRIEST	58.96	29255	96.77

Enron email network

Dataset statistics

Nodes: 36692
Edges: 183831
Number of Triangles: 727044

Results on a run

Email-enron	Time (s)	Triangle Count	Accuracy
Node Iterator	3.60	727044	100.00
Compact Forward	0.52	727044	100.00
DOULION (p=0.4) Node Iterator	0.59	724562	99.66
DOULION (p=0.4) Compact Forward	0.22	731703	99.36
TRIEST	171.69	662945	91.18