ID2222 Homework 3 Report

Mining Data Streams

Group 66: Tingyu Lei, Yiyao Zhang

Solution

We utilize Python and its built-in functions for this homework. We have implemented the algorithm in L. De Stefani, A. Epasto, M. Riondato, and E. Upfal, TRIÈST: Counting Local and Global Triangles in Fully-Dynamic Streams with Fixed Memory Size, KDD'16.

The implemented code is based on TRIÈST-IMPR. We process each graph edge once at a time to simulate a stream. The algorithm processes an arbitrary stream of edges using a fixed memory size M. To maintain a representative subgraph, it employs Reservoir Sampling.

Dataset

SNAP: Network datasets: Social circles

Core Functions

ReservoirSampler processes an arbitrary stream of edges using a fixed memory size `M`. To maintain a representative subgraph, it employs Reservoir Sampling. For the first `M` edges (time `t < M`), all edges are stored.For `t > M`, each new edge is included in the sample `S` with probability `M/t`. If included, a randomly selected existing edge is removed from `S` to maintain the fixed memory constraint.

```
class ReservoirSampler:
    """Reservoir Sampling algorithm - maintains a fixed-size random
sample"""

    def __init__(self, sample_size: int):

    def should_include(self) -> bool:
        """Decide whether the current element should be included in the
sample"""

    def add(self, edge: FrozenSet[int]):
```

```
        """Add edge to sample, randomly remove an edge if sample is
full"""
```

StreamingTriangleCounter has a run method that orchestrates the "Unconditional Update" strategy of `TRIÈST-IMPR`. The process for each edge in the stream is:

- WeightCalculation: The method `_get_estimation_factor` computes `eta(t)`.
- UnconditionalCounterUpdate: The `_update_triangles` method is called before the sampling decision. It searches for common neighbors in the current reservoir and increments the triangle count by `eta(t)` .
- Sample Update: Only after counting the algorithm decide whether to add the edge to the reservoir via `reservoir.should_include`.

```
class StreamingTriangleCounter:
    """Streaming triangle counting algorithm using Reservoir Sampling
(Triest)"""

    def __init__(self, file_path: str, memory_size: int):


    def find_neighbors(self, edge: FrozenSet[int]) -> Set[int]:
        """Find common neighbors of the two vertices in the edge"""

    def update_triangles(self, edge: FrozenSet[int], weight: float):
        """Update triangle count"""

    def get_estimation_factor(self) -> float:
        """Calculate estimation factor (Triest-IMPROVED version)"""

    def run(self) -> float:
        """Run algorithm to process stream data"""
```

Questions

1. What were the challenges you faced when implementing the algorithm?

   I think the most challenging part is to design the data structures, for example, we used `frozenset` to store edges and used `DefaultDict` to store `vertex_triangles` count.

2. Can the algorithm be easily parallelized? If yes, how? If not, why? Explain.

   No, the core algorithm cannot be easily parallelized for a single stream. The state of the reservoir (sample `S`) at time `t` strictly depends on the state at `t-1` and the current random decision.
3. Yes, because the algorithm is explicitly designed for streams of arbitrary length (unbounded) while using a fixed amount of memory `M` specified by the us. It uses Reservoir Sampling to ensure that no matter how large the stream grows (how high `t` gets), the memory usage remains constant (`O(M)`).
4. Does the algorithm support edge deletions? If not, what modification would it need? Explain.
   No, our code (TRIÈST-IMPR) does not support edge deletions. But according to the paper, to support deletions (fully-dynamic streams), we would need to implement the TRIÈST-FD algorithm.

How to run

1. Extract the homework folder.
2. Install Python3
3. Run `streaming_triest.py` in the current folder.
4. Enter memory_size.

Results

We tested the code for the memory size of 1000, 2000, 5000, 10000 and 20000, each for 5 rounds and recorded the results as below:

```
M=1000, Run 1: 1681449.63
M=1000, Run 2: 1831832.86
M=1000, Run 3: 1659017.61
M=1000, Run 4: 1519076.24
M=1000, Run 5: 1721116.25
M=1000, Average: 1682498.52

M=2000, Run 1: 1590100.40
M=2000, Run 2: 1621172.89
M=2000, Run 3: 1492087.34
M=2000, Run 4: 1608321.53
M=2000, Run 5: 1562161.75
M=2000, Average: 1574768.78

M=5000, Run 1: 1654881.06
M=5000, Run 2: 1532112.85
M=5000, Run 3: 1578575.18
M=5000, Run 4: 1619912.58
M=5000, Run 5: 1578411.17
M=5000, Average: 1592778.57

M=10000, Run 1: 1609531.31
M=10000, Run 2: 1592970.04
M=10000, Run 3: 1633420.22
M=10000, Run 4: 1603090.63
M=10000, Run 5: 1609732.21
M=10000, Average: 1609748.88

M=20000, Run 1: 1622132.04
M=20000, Run 2: 1645303.56
M=20000, Run 3: 1604481.81
M=20000, Run 4: 1578057.06
M=20000, Run 5: 1585226.23
M=20000, Average: 1607040.14

```

We can see from the results that increasing the sample size (`M`) significantly reduces the variance and error of the estimation. While `M=1,000` provides a rough ballpark, a memory size of at least `M=10,000` is recommended for this dataset to have reliable results.