Data mining ID 2222

Short report on homework 3

Mining data streams

Sofie Schnitzer and Adam Åström

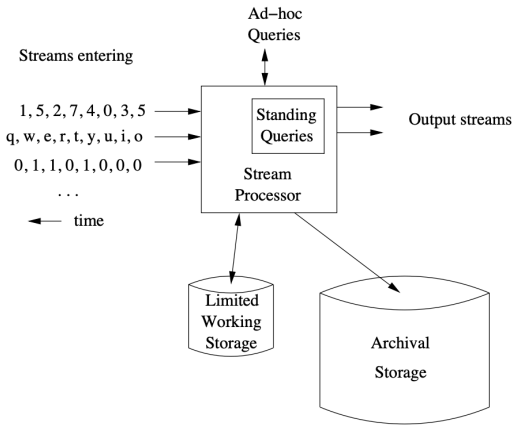
November 25, 2024

Introduction

This week's assignment focused on developing an algorithm to process streaming graphs, which are graphs that evolve dynamically as data arrives continuously. A streaming processor is designed to handle data that is not available all at once but arrives as a steady flow at varying rates and in different formats. A realistic scenario can be image data stream processing which needs to handle:

- **Satellite data streams** often provide high-resolution imagery but at a slower pace.
- **Surveillance camera feeds**, on the other hand, generate low-resolution video streams at a high frequency.

These diverse characteristics of data streams highlight a need for algorithms that can process and analyze dynamic data while balancing resource constraints like memory.



The task is to process streaming graphs and estimate:

1. **Transitivity** : A measure of how interconnected a graph is, indicating the probability that two neighbors of a node are also connected (i.e., form a triangle).
2. **Triangle Counts** : The number of triangles (closed triplets) in the graph.

We achieve this using the STREAMING-TRIANGLES algorithm, which uses reservoir sampling to keep track of a fixed-size sample of edges and wedges (open triplets). This allows us to estimate graph metrics efficiently without storing the entire graph in memory.

Metrics for this solution:

1. **Triangles (T):** The total number of triangles in the graph.
2. **Wedges (W):** The total number of wedges. A wedge can be part of a triangle or remain open (not closed by a third edge).
3. **Transitivity (κ):** Measures how many wedges are closed into triangles.

$$\kappa_t = 3 \cdot \frac{\text{Number of Closed Wedges}}{\text{Total Wedges in Reservoir}}$$

The birthday paradox:

This is a probabilistic result that shows likelihood of people in a group sharing birthdays. As you add more people, you're comparing every new person's birthday with everyone else already in the group. The number of comparisons grows rapidly, increasing the chance of a match. This paradox is applicable to the algorithm as it explains how a small amount of edges can still lead to wedges (just like people having the same birthdays).

Basic Implementation

The first function reads the graph file and returns a list of edges, the function iterates over each line in the file and adds all edges as tuples to a set.

To initialize the algorithm, we give it two parameters, the edge reservoir, which is the amount of edges to store, and the wedge reservoir which is the amount of wedges to store. These control how much memory we are allowing the algorithm to store. For our test, these were both set to 1000.

- The adjacency list is updated to include the new edge (u,v). This allows efficient lookups of neighbors for future computations.
- For each wedge (a,b) in the wedge_reservoir, the function checks if adding the new edge (u,v) completes the wedge into a triangle.
- If (u,v) forms a triangle with (a,b), the corresponding wedge in is_closed is marked as True.

```
        for i, wedge in enumerate(self.wedge_reservoir):
            a, b = wedge
            if (a == u and v in self.adjacency_list[b]) or \
            (b == u and v in self.adjacency_list[a]) or \
            (a == v and u in self.adjacency_list[b]) or \
            (b == v and u in self.adjacency_list[a]):
                self.is_closed[i] = True
```

We want to update the state of the algorithm for all coming lookups of new edges.

- The edge_reservoir is updated using reservoir sampling. If the reservoir has space, the new edge is added directly and if the reservoir is full, a random edge is replaced with the new edge, ensuring a random sample of edges from the stream.

```
        if len(self.edge_reservoir) < self.se:
            self.edge_reservoir.append(edge)
        else:
            replace_index = random.randint(0, len(self.edge_reservoir))
            if replace_index < self.se:
                self.edge_reservoir[replace_index] = edge
```

- New wedges involving the new edge (u,v) are generated by pairing u and v with their neighbors. These wedges are candidates for future triangle checks.

```
        new_wedges = []
        for node in self.adjacency_list[u]:
            if node != v:
                new_wedges.append((node, v))
        for node in self.adjacency_list[v]:
            if node != u:
                new_wedges.append((node, u))
```

- The wedge_reservoir is updated using reservoir sampling. Works the same as when sampling edges.

- The total number of wedges observed (total_wedges) is incremented by the number of newly generated wedges.

Once the update function has been run for each edge in the data, we run a function for estimating the transitivity and number of triangles.

- The transitivity takes the closed wedges * 3 and divide that by the length of the wedge reservoir. The reasoning behind multiplying the closed wedges count by 3 is that each closed wedge (triangle) closes 3 wedges.

- For the triangle count, we take the result we get when calculating the transitivity, multiply it with the total counted wedges during the run and divide by 3. Converts the total closed wedges into the number of triangles, since each triangle is counted three times in the closed wedges.

```
   def estimate(self):
       closed_wedges = sum(self.is_closed)
       transitivity = (3 * closed_wedges / len(self.wedge_reservoir)) if
self.wedge_reservoir else 0
       estimated_triangles = (transitivity * self.total_wedges / 3) if
self.total_wedges > 0 else 0
       return transitivity, estimated_triangles
```

Result

Since we are using reservoir sampling, we can only ever get an estimation of the metrics, and we will get different results each time we run. Two examples of running the same data file with the same max reservoir size:

```
Estimated Transitivity (κ_t): 0.123
Estimated Triangles (T_t): 631286.553
Runtime: 11.9537 seconds
```

```
Estimated Transitivity (κ_t): 0.129
Estimated Triangles (T_t): 662081.019
Runtime: 14.2732 seconds
```

Discussion (bonus)

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

For testing purposes, and the fact that the data we test on is not too big, this is not too big of a challenge, since we will not need to deal with very long runtimes and complex memory balancing. However, thinking realistically about implementing this solution for ex. The surveillance data streams mentioned in the introduction, it would be very hard to determine what fixed size sample data would be best to use. These challenges are particularly evident when determining the appropriate sample sizes for reservoirs while balancing accuracy and resource constraints. A fixed reservoir size might work well for one stream (ex, satellite data) but perform poorly for another (ex surveillance feeds).

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

Perhaps it could, but it would be challenging and require changes. One reason for this is that the edge and wedge reservoirs are constantly updated, which for parallelization would require them to be stored in another way and it would be hard to avoid and manage inconsistencies when

different threads modify the reservoirs simultaneously. If this problem was solved, and the updates across the threads were synchronized it could work. However, this would lead to a lot of overhead which might cancel out any improved performance of parallelizing the algorithm in the first place. So would it be easy? Probably not.

3. Does the algorithm work for unbounded graph streams? Explain.

As the algorithm is designed it could handle unbound graph streams, and reservoir sampling is a method used for handling these types of streams. There is still one negative aspect with letting these fixed size reservoirs be used in an unbound graph stream - as the graph grows indefinitely, the reservoirs represent a smaller and smaller fraction of the total edges and wedges in the graph. Also, the dictionary (adjacency_list) used to store all neighboring edges will grow very big, which could lead to memory issues for graphs with many nodes.

4. Does the algorithm support edge deletions? If not, what modification would it need? Explain.

No, if we were to delete an edge in the graph, it would still remain in the reservoir and this would lead to the approximations of transitivity and triangles being inaccurate. Similarly, wedges in the wedge_reservoir that rely on the deleted edge are no longer valid but are not removed or updated.

The adjacency list does not account for deletions. If an edge (u,v) is removed, u and v will still appear as neighbors in the adjacency list, causing incorrect triangle detection. And the same goes for the is_closed list where triangles closed containing the deleted edge would stay closed.

All these above states places where edges would remain would need to be updated and recalculated in order to make the algorithm handle deletions. I would work similar to the update function but in reverse, detecting all the places where the edge occurs and removing it.