Homework 3: Mining Data Streams

Evita Stenqvist
Linnea Fredriksson

November 2021

Streaming Triangles on web-NotreDame

We chose to implement the data stream algorithm presented in *A Space-Efficient Streaming Algorithm for Estimating Transitivity and Triangle Counts Using the Birthday Paradox* [JSP15].

NOTE: Reservoir sampling is baked in to this algorithm, we will point it out in the attached code of this report.

We applied the algorithm on both undirected and directed datasets taken directly from *SNAP* [LK14], even though the algorithm states that it can only be applied on undirected datasets. Interestingly, in the section *4. Experimental results* [JSP15] several directed datasets are mentioned with the comment "each graph is converted into a stream by taking a random ordering of the edges" - implying that simply treating the edges as unordered suffices. At least, we assumed so.

For brevity, our tested dataset is *web-NotreDame* [R A99]. Setting the two parameters to the algorithm, $s_e = s_w = 20k$, we managed to get decent results approximating the number of triangles and transitivty for the dataset.

True values provided by SNAP page [LK14]:

```
Average clustering coefficient 0.2346
Number of triangles 8910005
```

Approximated values by our implementation:

```
Approx clustering coefficient 0.2142
Number of triangles 8464102
```

The number of triangles are just above a 5% error rate, and the clustering coefficient within 10% error rate. Due to the difficulties we faced, mentioned in the coming subsection, we did not have time to run the algorithm several times to find a mean and deviation.

Optional task

What were the challenges you have faced when implementing the algorithm?

Initialisation. We spent days trying to understand how to initialise the algorithm. Lines like "A set of uniform random edges can be maintained by reservoir sampling [Vitter 1985]. From these edges, we generate a random wedge by doing a second level of reservoir sampling." [JSP15] had us uncertain if the authors had excluded parts of the pseudocode for brevity (maybe "second level of reservoir sampling" is so obvious it needn't be shown further?) before we released that this step was indeed included in the pseudocode. Mostly, what tripped us up was the idea that reservoir sampling wouldn't be so simple as what the pseudocode presented and so we over-complicated in foresight. Secondly, we had no idea what decent values for s_e and s_w would be and felt that the presented experiments left this detail uncommented.

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

Maybe, we think, if we don't need to think about continuously approximating (i.e. only interested in a bounded stream and it's end result) as long as locking of some sort is implemented on our two main data-structures $edge_reservoir$ and $wedge_reservoir$ such that racing conditions don't occur when changing out items. Otherwise, batching *update* should work.

Does the algorithm work for unbounded graph streams? Explain.

Yes, the algorithm is effectively an "online" method. It's two data structures are continuously updated provided the data it sees and it approximates transitivity and number of triangles based on this. Intuitively, the more edges we've seen, the more triangles we see while transitivity adjusts.

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

No, it does not support edge deletion. Seeing as the memory footprint is so small, it would be difficult to modify such that a backtrace can be done to "undo" the found triangles. Potentially keeping another data-structure indicating a starting point to start a-new from if an edge is deleted... possibly hashing values to a order of magnitude smaller buckets and then depending on bucket going back. This doesn't really work for true streams thou, as the information is discarded after processing as opposed to a file where we can set the pointer back.

Implementation

```
def __print_stats(is_closed: list[bool], s_e: int, t: int, tot_wedges: int):
    # Let p be the fraction of entries in isClosed set to true
    p = sum(is_closed) / len(is_closed)
    # set k_t = 3p
    k_t = 3 * p
    # k_t is the transitivity at t
    print(f"k_{t}: {k_t}")
    # set T_t = [pt^2/s_e(s_e-1)] x tot_wedges
    T_t = ((p * (t ** 2)) / (s_e * (s_e - 1))) * tot_wedges
    # T_t is the number of triangles at t
```

```
    print(f"T_{t}: {T_t} \n")

def main(open_file, s_e: int, s_w: int):
    """ The main part of the algorithm which calls
        update on every edge from the stream"""

    # Initialise our data structures
    edge_res = [None] * s_e
    wedge_res = [None] * s_w
    is_closed = [False] * s_w
    tot_wedges = 0

    # Iterate over every vertice in our file
    t = 1
    while True:
        line = open_file.readline()
        if not line or line == b"\n":
            break
        if line.startswith(b"#"):
            # Don't bother with comments
            continue
        edge = Edge.from_line(line)
        tot_wedges = update(
            edge, t, s_e, s_w, edge_res, wedge_res, tot_wedges, is_closed
        )
        t += 1

        if t % 1000 == 0:
            __print_stats(is_closed, s_e, t, tot_wedges)

    __print_stats(is_closed, s_e, t, tot_wedges)


def update(
    edge: Edge,
    t: int,
    s_e: int,
    s_w: int,
    edge_res: list[Edge],
    wedge_res: list[Wedge],
    tot_wedges: int,
    is_closed: list[bool],
):
    """ The update stage of the algorithm, the "steps"
        comments are directly taken from the paper"""

    # Steps 1-3 determines all the wedges in the wedge reservoir
    # that are closed by e_t and updates is_closed accordingly.
    for i in range(s_w):
        if wedge_res[i] and wedge_res[i].closed_by(edge):
```

```
            is_closed[i] = True

    # Steps 4-7 we perform reservoir sampling on edge_res. This involved
    # replacing each entry by e_t with probability 1/t, remaining steps
    # are executed iff this leads to any changes in edge_res
    updated = False
    for i in range(s_e):
        x = random.random()
        if x <= 1 / t:
            updated = True
            edge_res[i] = edge

    # We perform some updates to tot_wedges and determine the
    # new wedges Nt (new_wedges_t). Finally, in Steps 11-16,
    # we perform reservoir sampling on wedge_res, where each entry is
    # randomly replaced with some wedge in Nt. Note that we may remove
    # wedges that have already closed.
    if updated:
        tot_wedges = get_wedges(edge_res)
        new_wedges = get_new_wedges(edge, edge_res)
        if not new_wedges:
            return tot_wedges
        for i in range(s_w):
            x = random.random()
            if x <= len(new_wedges) / tot_wedges:
                index = random.randint(0, len(new_wedges) - 1)
                wedge_res[i] = new_wedges[index]
                is_closed[i] = False
    return tot_wedges
```

References

[JSP15] Madhav Jha, C Seshadhri, and Ali Pinar. "A space-efficient streaming algorithm for estimating transitivity and triangle counts using the birthday paradox". In: *ACM Transactions on Knowledge Discovery from Data (TKDD)* 9.3 (2015), pp. 1–21.

[LK14] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. `http://snap.stanford.edu/data`. June 2014.

[R A99] A.-L. Barabasi R. Albert H. Jeong. *Web Notre Dame*. `https://snap.stanford.edu/data/web-NotreDame.html`. 1999.