Implementing & Evaluating Space Efficient Algorithms for Detecting Large Neighbourhoods in Graph Streams

Dom Hutchinson

February 26, 2020

Chapter 1

Background

1.1 Motivation

When designing algorithms time efficiency is generally viewed with primary importance, over space efficiency. However, if the memory used for an algorithm exceeds the RAM space allocated to it during execution then virtual memory would have to be used. Virtual memory has a much longer read/write time than RAM meaning that an algorithm's run-time will be much greater in this scenario. Thus, algorithms which are designed to work with large data sets space efficiency should be viewed as having primary importance.

In the early 2000s companies began to realise that data could be used as a commodity, and with this, the amount of data being collected has surged in the years since. In this same time period, we have not seen the same surge in the amount of RAM typical computers have. Thus people wishing to capitalise on this increase in available data need to give greater thought to the space efficiency of the algorithms they are using. Otherwise, run-times could become unusable.

Social networks are good sources of data about people. This data can be leveraged in many ways, notably within advertising. Advertisements which are targeted at people with certain observed behaviours have been shown to have more than twice the click-through rate of standard online advertisements [2]. This has motivated the development of techniques for assessing the behaviours of a person just from their online presence.

One technique is to look at who a person is connected to and then to use observations about these connections to infer the behaviours of said person. In turn, we have seen the rise of the "Social Media Influencer". These are people who are deemed to have a large online following and, moreover, a following which they are able to influence. Advertisers will approach influencers with deals by which the influencer will promote a product to their audience, hopefully driving sales.

The question for advertisers now becomes "Who is an influencer?". To answer this we need to indentify people with large neigbourhoods in a network & members of this network. We need to find the members of the neighbourhood (or at least a subset of them) in order to be able to assess what the demographics of the influencers audience are. This question can be generalised as the `Neighborhood Detection Problem`. Which is the problem I shall discuss in this paper.

Although in my description of the Neighbourhood Detection Problem I motivate its relevance by referring to finding social media influencers, it is applicable to many other scenarios.

- Which items are most commonly bought & which other items they are commonly bought with?

- Which resources in a network are being accessed most & by whom?

Answering these questions can be used to inform resource allocation in order to maximise a certain goal. Be that profit; or, network response speeds.

1.2 Neighbourhood Detection Problem

Graph Streams are a list of instructions for how to construct a graph and provide a dynamic way of representing graphs. In this paper, I shall discuss two types of graph streams: Insertion-Only Streams; and, Insertion-Deletion Streams. There are other types of graph streams that allow for additional operations such as updating vertex & edge values, but these are not relevant to the problem being considering in this paper.

Insertion-Only graph streams are a list of pairs of vertices, representing endpoints of unweighted-undirected edges. Each instruction describes a new edge to be inserted into the graph & potential up to two new nodes which are in the graph. There is no way to remove or update an edge in an Insertion-Only graph stream. We apply a limitation that no duplicate edges appear in an insertion-only stream and thus the order of instructions in an Inseriony-Only stream has no effect on the validity of the algorithms which shall be discussed.

Insertion-Deletion graph streams consist of a pair of vertices and a boolean. The vertices represent the endpoints of an unweighted-undirected edge & the boolean describes whether this edge is being inserted or deleted from the graph. If it is an insertion instruction then we are adding the described edge to the graph; if it is a deletion instruction then we are removing it. Again we add a limitation that no edges are repeated & that we will not receive a request to delete an edge from the graph which does not currently exist in the graph. This does, however, mean that the order of instructions in an Insertion-Deletion stream is important as the deletion of an edge must come after it's insertion.

The limitations placed here often occur naturally in systems. On social networks you are not able to follow someone you are already following, nor unfollow someone you are not following.

In spite of these limitations which are imposed it is still possible to construct almost any undirected-unweighted graph using the two graph streams described. Suppose you wish to analyse which devices in a network are making the most request. Here vertices represent devices & edges represent requests. For this scenario, you would want to allow for multiple edges (say $n > 2$) between the same pair of devices (say $\{x, y\}$). One solution would be to add n new vertices $\{v_{x,y,1}, \ldots, v_{x,y,n}\}$ each with a label stating they are just intermediate nodes and not full devices. Creating edges from x to each of $\{v_{x,y,1}, \ldots, v_{x,y,n}\}$ & from y to each of $\{v_{x,y,1}, \ldots, v_{x,y,n}\}$ allows for an interpretation of multiple occurrences of the edge (x, y).

If a graph stream is ordered by time (which is common in real-life applications) then it represents a dynamic-evolution of the graph over time. Allowing the user to roll back the graph should they wish to make assessments during particular time periods, rather than just assess the final graph.

The algorithms which shall be discussed in this paper are designed to work with just a single pass of a graph stream. This is not only time-efficient as it does not require multiple reads of the stream, but means that should more information be gained after the algorithm is run it is easy to incorporate it into the algorithm if we store the final state of the data structures used. Single-pass algorithms mean we don't necessarily have to store the graph stream in a local file, we could simply send instructions from the system being modeled straight into the algorithm. This is useful when a graph stream potentially could take up multiple terra-bytes, but is unnecessary in most applications and can mean it is harder to verify results later.

The `High Degree Detection Probem` is a problem in graph theory where we seek to find a vertex of sufficiently high degree, in a given graph. This node does not necessarily have the greatest degree in the graph, and often we cannot be sure whether it does as the greatest degree is unknown beforehand. This problem can be formally stated as

Problem 1 `High Degree Detection`.
Let $G = (A \cup B, E)$ be a bi-partite graph with vertex sets A & B, where $|A| = n$ and $|B| =$ poly n , and edge set E. Set a restriction that at least one vertex in A has degree, at least, d.
In `High Degree Detection`(G, d) we are tasked with outputting a vertex from A with degree, at least, d.

This problem can be solved efficiently for both insertion-only & insertion-

deletion graph streams with a fairly simple algorithm which simply runs through all the edges, incrementing or decrementing (depending on whether it is an insertion or deletion instruction) a count for the degree of each vertex as it goes. Returning the first vertex it finds with degree equal to d.

Algorithm 1: Single Pass High Degree Detection

require: Stream $\{(i_0, x_0, y_0) \dots (i_n, x_n, y_n)\}$, degree bond d

$D \leftarrow \{\{\}\}$ {degree map}
for $j \in [0 \dots n]$ **do**
  if $x_j \in D$ **then**
    if i_j **then** $D[x_j] += 1$
    else $D[x_j] -= 1$
  else $D[x_j] = 1$
  if $D[x_j] = d$ **then return** x_j
  repeat with y_j
return *FAIL*

In **Algorithm 1** i is a boolean which is true if the edge is an insertion edge, and false if it is a deletion edge. x & y are the two endpoints of the edge. The algorithm makes the reasonable assumption that you will never recieve a deletion-instruction for an edge which is not in the graph, nor an insertion-instruction for an edge which is already in the graph.

The `High Degree Detection Probem` is limited in its applications as it only returns a vertex with high degree & gives you no information as to its neighbourhood. Solving this problem would not be very helpful when it comes to identifying influencers as it would only return people who have large followings, but not who their followers are. Thus an advertisier would not be able to evaluate whether the returned person's followers represented the audience they wished to target.

The `Neighbourhood Detection Problem` is an extension of the `High Degree Detection Problem`. The task is to return a vertex with sufficiently high degree & a certain proportion that vertex's neighbourhood. This is formally stated as

Problem 2 `Neighbourhood Detection`.
Let $G = (A \cup B, E)$ be a bi-partite graph with vertex sets A, B, where $|A| = n$ and $|B| =$ poly n, and edge set E.
In `Neighbourhood Detection`(G, d, c) we are tasked with outputting a vertex from A and at least d/c of its neighbours in B.
Here d is a threshold parameter & c is an approximation parameter.

Solving the `Neighbourhood Detection Problem` allows for a more general assessment of the influence of the returned vertex. In the context of finding social media influencers the returned neighbourhood can be assessed to determine whether the returned person is suitable for a given advertising

campaign. When assessing the members of the neighbourhood you could look at the demographics of the members & how active the members are, among other features, in order to determine whether they fit the target audience of the campaign. For cases where $c \neq 1$ you only get a subset of a person's whole neighbourhood but for sufficiently high d you an implement statistical tools to still be able to draw meaningful inferences. Later it shall be shown that there are meaningful advantages to increasing the value of c.

1.3 Why Space Efficiency?

Simply solving the `Neighbourhood Detection Problem` is fairly trivial. **Algorithm 2** is an algorithm for solving the `Neighbourhood Detection Problem` for insertion-only graph streams by recording the vertices in the neighbourhood of every A-vertex and then returning the first encountered neighbourhood with $\frac{d}{c}$ members.

Algorithm 2: Naïve Single-Pass Insertion-Streaming Algorithm for Neighbourhood Detection

require: Stream $\{(s_0, t_0) \dots (s_n, t_n)\}$, degree bound d, precision bound c

$N \leftarrow \{\{\}\}$ {neighbourhoods}
for $i = 0 \dots n$ **do**
  append t_i to $N[s_i]$
  if *size*$(N[s_i]) = \frac{d}{c}$ **then return** (s_i,$N[s_i]$)
return *FAIL*

Algorithm 2 requires $O\left(n^2\right)$ space. Thus the space used to run the algorithm grows very quickly which could easily lead to the computer's RAM overflowing, increasing execution time. For solutions which are space inefficient run times become unmanagable for large graph streams. In 2014 in the UK, Facebook had 36.68 million users [3] each with an average of 155 friend connections [4]. In order for this to be represented in a bi-partite graph for `Neighbourhood Detection` we would need a graph stream with ∼5.6 billion edges. **Algorithm 2** would not be suitable for analysing this graph.

Instead we must look for much more space efficient algorithms for `Neighbourhood Detection`. In the rest of this paper I shall discuss & test an algorithm which can solve `Neighbourhood Detection` for insertion-only streams with space $O(n \log n + n^{\frac{1}{c}} d \log^2 n)$; and, two approaches to solving for `Neighbourhood Detection` for insertion-deletion streams with space $\tilde{O}\left(\frac{xd}{c}\right)$ & $\tilde{O}\left(\frac{nd}{c}\left(\frac{1}{x} + \frac{1}{x}\right)\right)$ respectively.

1.4 Plan

In this paper I shall implement and test the two algorithms presented by Dr Chrisian Konrad in [1]. These algorithms solve the `Neighbourhood Detection Problem` for insertion-only & insertion-deletion graph streams in a space-efficient manner. Both algorithms require only a single pass of the stream.

I shall use the naïve algorithm described in **Algorithm 2** as a benchmark for both time & space efficiency of these two algorithms. I shall use a four different graph streams, described in **Table 1.1**, for my tests. These graphs were acquired from the *Stanford Network Analysis Project*, [5].

The theoretical space requirements for these two algorithms both depend on the degree variable, d, and precision variable, c. I shall perform tests where I vary each of these in order to see what the space requirements are in practice, and then compare these results to the theory.

Table 1.1: Test Graph Streams

Name	# Edges	# Vertices	Max Degree	File Size, KB
`facebook_small`	292	52	36	3
`facebook`	60,050	747	586	587
`gplus`	1,179,613	12,417	5,948	52,839
`gplus_large`	30,238,035	120,100	104,947	1,328,820

Algorithm 3: Single l_0-Sampler for vector

require: Vector $\mathbf{a} = (a_1, \ldots, a_n)$

$j = 1$.
$\mathbf{a}_j = \mathbf{a}$.
while *true* **do**
    H=new hash function.
    $a_{j+1} = \mathbf{0}$.
    $j = j + 1$.
    `// Sampling`
    for $i \in [1, n]$ **do**
        $h = H(i)$.
        if $h \leq \frac{n^3}{2^j}$ **then** $a_{j+1,i} = a_{j,i}$
        else $a_{j+1,i} = 0$
    $\mathbf{a}_j = \mathbf{a}_{j+1}$
    `// Attempt Recover`
    $w_1 := \sum a_{j,i}$.
    $w_2 := \sum i a_{j,i}$.

    if $w_2 \equiv 0$ **then**
        return `FAIL` `//` $a_j \equiv \mathbf{0}$
    if $w_1 \equiv 1$ **then**
        return w_2 `// Index of non-zero value`

Hash functions should be k-wise independent and map from n to n^3 where n is the number of vertices in the graph (*i.e.* length of vector since it is from a graph matrix).

Algorithm 4: Single l_0-Sampler for a stream

require: Stream $\mathbf{s} = \{s_1, \ldots, s_n\}$ with n known, Hash Function H

sampled={}.
for $i \in [1, n\}$ **do**
    $h = H(i)$.
    if $h \leq \frac{n^3}{2^1}$ **then** sampled.append(s_i)
$j = 1$.
while *sampled.size()*> 1 **do**
    $j = j + 1$.
    new_sample={}.
    for $i \in [1,$ *sampled.size()*$]$ **do**
        $h = H(i)$. `// Consider changing hash function`
        if $h \leq \frac{n^3}{2^j}$ **then** new_sample.append(sampled[i])
    sampled=new_sample.
if *sampled.size()*$\equiv 1$ **then return** *Success,sampled[0]*
return *Fail,NULL*

Algorithm 5: m Parallel l_0-Samplers for a stream

require: Stream $\mathbf{s} = \{s_1, \ldots, s_n\}$ with n known, Hash Functions $\mathbf{H} = \{H_1, \ldots, H_m\}$

output={}.
sampled={{}}.
for $i \in [1, n]$ **do**
    `// Single pass of stream`
    for $j \in [1, m]$ **do**
        $h = H_j(i)$.
        if $h \leq \frac{n^3}{2^1}$ **then** sampled[j].append(s_i)
for $j \in [1, m]$ **do**
    $k = 1$.
    while *sampled[j].size()*> 1 **do**
        $k = k + 1$.
        new_sample={}.
        for $i \in [1,$ *sampled[j].size()*$]$ **do**
            $h = H_j(i)$. `// Consider changing hash function`
            if $h \leq \frac{n^3}{2^k}$ **then** new_sample[j].append(sampled[i])
        sampled[j]=new_sample.
    if *sampled[j].size()*$\equiv 1$ **then** output.append(sampled[j][0])
return *output*

Bibliography

[1] Christian Konrad *Streaming Frequent Items with Timestamps and Detecting Large Neighborhoods in Graph Streams.* November 2019

[2] Howard Beales *The Value of Behavioral Targeting.*

[3] https://www.statista.com/statistics/553538/predicted-number-of-facebook-users-in-the-united-kingdom-uk/

[4] https://www.telegraph.co.uk/news/science/science-news/12108412/Facebook-users-have-155-friends-but-would-trust-just-four-in-a-crisis.html

[5] http://snap.stanford.edu/data/