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

Dom Hutchinson

April 19, 2020

Contents

Summary/Abstract

I.i Background

Graphs are a popular way of representing relationships between objects. They are particularly popular within the field of data science & with the recent movement of *Big Data* it is becoming common to want to deal with large graphs. Moreover, it is common to want to use graphs which cannot be loaded fully onto a computer's RAM and instead require the use of virtual memory. Having to use virtual memory drastically increases request time for data & thus negatively affects the runtime of algorithms which manipulate these large graphs. Often, in practice, the fastest algorithms for analysing graphs are those which are more space efficient.

Graph streams have been introduced to reduce the space required to store a graph since they require $O(|E|)$ space, rather than $O(|V|^2)$ space which a traditional matrix representation requires. Since the number of edges in a graph is cannot be greater than, and is usually much lower than, the square of the number of vertices this is a big improvement. However many graphs still exist which would overflow RAM when loaded in full when in stream form. This is solved by reading the stream one entry at a time as this requires $O(1)$ space. Thus the most space-efficient algorithms for analysing graphs are streaming-algorithms which read instructions sequentially.

Graph streams can be used to show the evolution of a graph by insertion & deleting edges over some time period, in these implementations it is possible for a graph stream to require more space than a matrix representation but since only one instruction is every loaded into the RAM at one time this does not affect the total memory used by a streaming-algorithm.

High degree detection is a common and trivial problem in graph theory. The `High Degree Detection Problem` entails finding a vertex in a graph with sufficiently high degree. This can be trivially solved for graph streams by keeping counters for the number of edges connected to each vertex and updating the counters after each instruction from the graph stream.

Although it is useful to identify influential vertices in a graph, it is hard to fully utilise this knowledge without additional information about the neighbourhood of these vertices. Knowing the neighbourhood of a vertex, or at least a subset of it, allows you to make inferences about that node. This leads to the `Neighbourhood Detection Problem` which is formally stated as

Problem 1 `Neighbourhood Detection`.
Let $G = (A \cup B, E)$ be a bi-partite graph with vertex sets A, B, where $|A| = n$ and $|B| = \text{poly } n$, and edge-set E.
In `Neighbourhood Detection`(G, d, c) we are tasked with outputting a vertex from A with at least d/c of its neighbours in B. We can assume that G contains at least one node of degree d. Here $d \in \mathbb{N}$ is a threshold parameter & $c \in (0, 1]$ is an approximation parameter.

The `Neighbourhood Detection Problem` is defined for bi-partite graphs in order to generalise the problem. Note that, if we were to define the vertex sets A, B to be equivalent then then the problem holds for every type of graph.

Again, the `Neighbourhood Detection Problem` can be solved trivially by storing the neighbourhoods of every vertex in the graph & updating them after each instruction from the graph stream. This requires $O(|E|)$ space and can very quickly lead to overflowing the RAM which is undesirable since it means slower run-times for large graphs.

I.ii Motivating Applications

There are several real-world applications of the `Neighbourhood Detection Problem` which motivate investigating it. These are situations where simply knowing highly connected elements of a network is not sufficient.

- Given a list of connections within a social network, solving the problem would allow you to not only find *'social influencers'* but those they are connect to too. This allows you to verify that they are indeed a *'social influencers'*, and not someone connected to a lot of bot accounts, and to assess what demographics they have influence over.
  For a marketing agency this would mean they don't waste money on people who don't actually have any influence to sell a product, nor on people who have the wrong demographics for a certain product.

- Given a list of passengers on public transport you could find people who have come into contact with a lot of people. This is particular relevant in the wake of the COVID-19 outbreak as it could be used to identify people who are likely to have come into contact with infected people.

- Given a list of receipts in a shop you could find the items which are popular, along with which items they are commonly bought with. This information could be used to inform how shelves are stacked and which promotions are run by the store, in order to encourage sales.

- Given a log of traffic within a network you could identify which resources are being accessed most often, and by whom. This can be used to inform which resources should be upgraded & which connections should be prioritised.

I.iii Structure

In this paper I am going to implement the two algorithms proposed by Dr Christian Konrad in [1] for solving the `Neighbourhood Detection Problem` for graphs streams in a space-efficient manner. I will use this implementation to assess how practical these algorithms are and now to maximise their practicalities.

I shall structure this investigation as follows

- Insertion-Only Streams
  - Layout theory
  - Discuss features that need to be/can be adjusted in the implementation.
  - Implement - mentioning any other problems.
  - Evaluate & test different combinations of parameters to optimise the practical uses of the algorithm.
  - Discuss limitations of the implementation. (ie What graphs work)
- Repeat for Insertion Deletion.

I.iv Objectives

The objectives of this project are

- Implement Christian's algorithms and discuss any considerations that need to be taken in doing so.
- Tune parameters to make algorithms as practical as possible.
- How practical are the algorithms.

Chapter 1

Preliminaries

1.1 Definitions

A *Graph* is a data structure used to represent pairwise relationships between objects. A Graph is defined to have a vertex set, which holds its objects, and an edge-set, which holds the relationship between these objects. Graphs are traditionally visualised with circles for each vertex and lines between vertices that share an edge. There are variations on graphs where the edges are weighted and/or directional. In this project we are only required to deal with undirected, unweighted graphs.

A *Bipartite Graph* is a graph whose vertex set can be partitioned into two distinct sets in such a way that all edges in its edge-set run between these partitions & not within them.

The *Neighbourhood* of a vertex in a graph is the set of vertices which it shares an edge with. The *Degree* of a vertex is the size of its neighbourhood. A vertex is s *Sparse* if its degree is less than, or equal to, s.

A *Graph Stream* is a technique for storing graphs. A graph stream is made up of a series of instructions which are read sequentially and describe how to construct the graph. These instructions describe modifications to the edges of the graph. In this project we are only concerned with two types of graph stream.

- *Insertion-Only Streams* where each instruction adds a new edge to the graph.
- *Insertion-Deletion Streams* where each instruction either adds a new edge to the graph or removes an existing edge from the graph.

Neither of these allow for duplicates of the same edge in the graph, at the same time. Insertion-Deletion streams are used for graphs whose edge-set is dynamic over time, consider a social network where users are able to follow & un-follow each other. There are other versions of graph streams which all for changes to the weights of edges but these are out of scope for this project. It is noteworthy that insertion-only streams equivalent to insertion-deletion streams which have no deletions. This means that they can be implemented in exactly the same way.

Streaming Algorithms are algorithms which are designed to take a stream of sequential instructions as their input. Streaming algorithms can be very space-efficient since only one instruction is read at a time, requiring constant space. Typically streaming algorithms aim to only require a single pass of the input stream so that you can extend the input stream without having to restart the algorithm. Single-pass streaming algorithms are online algorithms.

An α-*Approximation Algorithm* is an algorithm which returns a result within an α factor to the optimal solution to the problem. Algorithms which solve the `Neighbourhood Detection Problem` are $\frac{1}{c}$-approximation algorithms since they only return d/c of the neighbours of a vertex which we know to have at least d neighbours.

1.2 Technologies

For the implementations in this project I chose to use C++. I initially considered using either python or C++, as they work in the object-orientated paradigm and I am proficient in them. I chose to go with C++ (specifically `C++11`) as it allows for more control with memory management and typically faster run-times. For this project the memory management is more important as I am looking to evaluate the space-efficiency of algorithms and being able to control their memory allocation means I can identify each time a change occurs to the amount of memory being used. The faster run-times is important when evaluating the practicalities of using these algorithms, as longer run-times make algorithms less practical.

I limited myself to using the *C++ Standard Library* (`std`) as the underlying implementations of this library are well document. This was particularly important for the abstract data types I used as I could check what the underlying implementations and adjust my evaluation accordingly. When implementing randomness I used the `<random>` module of `std` and seeded the generators with the current time so that each run would have a different generator.

1.3 Evaluation Techniques

When evaluating these algorithms there were three main parameters which needed to be tested for: success-rate; time-taken; and, space-used. These are all interconnected and come with their own trade-offs. Typically, the changes to space-used & time-taken when different strategies are implemented occur in the same direction but at different rates. Whereas, you expect success-rate to go in the opposite direction as the algorithm is using fewer resources. For these particular algorithms the success rate always remained high but for some setups it takes a lot longer to find a solution.

The evaluation process of each algorithm can be broken into two stages. First, I evaluated possible efficiencies to the underlying implementation. The changes tested at this stage did not affect the success-rate of the algorithms but did, in some cases, have a drastic effect on the amount of resources used by each algorithm. Second, I evaluated different set-ups for the parameters of the algorithm. This stage involved a lot more *tuning* of values & strategies as these affected the logic of the algorithm resulting in trade-offs between resource requirements & success-rate. The changes in the first were a lot more general & there was little variation in the results seen for different graph types, whereas in the later stage I had to consider different graph types.

I used a global variable to count the space used by different components of the algorithms (as well as the algorithm as a whole) and a timer to evaluate the run-time. For each variation I ran the tests multiple times, recording the results of each run. Typically for testing I set d to be the maximum degree of the graph as these algorithms are typically used to investigate the nodes of greatest degrees in a graph, and I would then vary the approximation factor c as seem reasonable for the given graph.

1.3.1 Generating Graphs

For this project I required a number of graphs to evaluate my programs with. By the nature of the algorithms these graphs had to be undirected & unweighted bi-partite graphs. The bi-partite requirement is unimportant as any graph can be treated as bi-partite by making the vertex set partitions copies of each other.

I implemented programs to generate graphs with specific properties which I wanted to test, namely graphs with high disparity in the degree of each node (such as star graphs) and graphs with no disparity between degrees (such as complete graphs). I implemented the following programs.

- `completeGraph.cpp` - Generates a graph stream for a complete graph.
- `randomGraph.cpp` - Uses a Bernoulli distribution to determine whether two vertices share an edge in an insertion-only stream, and a poisson distribution to determine how many insertions & deletions occur between each pair of vertices.
- `starGraph.cpp` - Generates a graph stream for a star graph.

All these programs take parameters for the number of vertices and `randomGraph.cpp` takes parameters for the proportion of edges. Additionally these, programs are able to generate insertion-only or insertion-deletion streams but the final graphs defined by the streams generated by `completeGraph.cpp` & `starGraph.cpp` are always of the appropriate shape.

I wanted some graphs generated from the real-world so that I could test the algorithms on the sort of data sets they are most likely to be used on. I found a collection of large graph streams, from different applications, in the `Stanford Network Analysis Project` (`SNAP`) datasets collection [5]. The vertices in the instructions of these graph streams were not ordered meaning that these graphs are not strictly bi-partite. This meant I needed to change my implementations slightly such that both ends of the edge are treated as A-vertices. This is equivalent to having a bi-partite graph where the A & B vertex sets are copies of each other, which is perfectly fine.

The graphs from `SNAP` were all insertion-only streams so I wrote a program which would generate an insertion-deletion stream from them by adding a defined proportion of deletion instructions throughout the stream. This program maintained the property that no edge was deleted before it was inserted.

I wrote `utility.cpp` to perform a number of tasks to these graphs. Most notably, it can produce a file which lists the final degree of every vertex in the graph stream and it can verify whether a given set is a subset of the neighbourhood of a given vertex. I used the later function to verify the outputs generated by implementation of Dr Konrad's algorithms.

Table 1: Details of graphs used during evaluation

Name	Source	Type	# Vertices	# Edges	# Instructions	Max Degree	File Size
facebook_small	SNAP	Insertion-Only	52	292	-	36	3 KB
facebook_small_deletion	Generated	Insertion-Deletion	52	258	326	33	5 KB
facebook	SNAP	Insertion-Only	747	60,050	-	586	587 KB
facebook_deletion	Generaetd	Insertion-Deletion	747	53,457	66,643	522	846 KB
gplus	SNAP	Insertion-Only	12,417	1,179,613	-	5,948	53 MB
gplus_deletion	Generated	Insertion-Deletion	12,417	1,049,309	1,309,917	4,998	60 MB
gplus_large	SNAP	Insertion-Only	102,100	30,238,035	-	104,947	1.3 GB
1000vertices	Generated	Insertion-Only	1,000	246,676	-	984	2 MB
doublepoisson05ID	Generated	Insertion-Deletion	10,000	8,639,534	24,452,736	4,977	305 MB
1000star	Generated	Insertion-Only	1,000	999	-	999	9 KB
1000starID	Generated	Insertion-Deletion	1,000	999	1,023	999	11 KB
1000complete	Generated	Insertion-Only	1,000	499,500	-	999	4 MB
1000completeID	Generated	Insertion-Deletion	1,000	499,500	509,524	999	5 MB

Chapter 2

Insertion-Only Streams

2.1 Degree-Based Reservoir Sampling

`Degree-Based Reservoir Sampling` is a technique for uniformly sampling a subset from the set of vertices whose degrees are greater than a specified minimum bound, while storing part of the neighbourhood for each of the sampled nodes. This sample is known as the *reservoir*.

`Degree-Based Reservoir Sampling` takes three parameters: the lower bound for the degree of vertices at which to start sampling, $d_1 \in \mathbb{N}$; the maximum neighbourhood size to store for each vertex, $d_2 \in \mathbb{N}$; and the maximum reservoir size, $s \in \mathbb{N}$. `Degree-Based Reservoir Sampling` succeeds if it finds a neighbourhood of size d_2.

Three data stores are used in `Degree-Based Reservoir Sampling`: a map of the degree of every node in the graph; a set for the reservoir; and a set for edges incident to the vertices in the reservoir.

The reservoir has an invariant that at any point in time it contains a uniform sample of the vertices whose degrees are known to be greater than d_1 at that time. This invariant is maintained by controlling how vertices are added to the reservoir. The first time the degree counter for a vertex v surpasses d_1 we consider whether to add v to the *reservoir*. There are two cases

- If the reservoir is <u>not</u> full (ie its size is less than s) then we add v to the reservoir.
- Otherwise, if the reservoir is already full then we use a Bernoulli random variable with probability $p = \frac{x}{s}$, where x is the size of the reservoir. If this random variable succeeds then we pick, uniformly at random, a vertex u currently in the reservoir and replace it with v. We remove all the edges in the edge set which are incident to u and no other vertices in the reservoir.

If v is not added to the reservoir at this time, it never will be in the future. Note that in the case that the reservoir is full & v is inserted into it, v is replacing an element that was in the reservoir and any progress which has been made towards finding a neighbourhood for that vertex is annulled in doing so. The amount of progress is not taken into account & thus if there is a high turn-over of elements in a reservoir then less progress is made towards finding a neighbourhood for each of them. This is something I will look into during the evaluation.

Whenever an edge is encounter which is in incident to a vertex in the reservoir then we add it to the set of edges. The sampler runs until the graph stream is exhausted at which point it returns, uniformly at random, one of the neighbourhoods of sufficient size.

`Degree-Based Reservoir Sampling` only works on insertion-only streams as it does not account for when the degree of a vertex falls below the threshold, after being sampled. This means the reservoir is not necessarily a uniform random sample of the vertices with degree $\geq d_1$.

For this project we are interested in how `Degree-Based Reservoir Sampling` can be implemented for bi-partite graphs, specifically we want to sample from the A-vertices of the

graph. For a bi-partite graph with n A-vertices `Degree-Based Reservoir Sampling` requires $O(n \log n + sd_2 \log n)$ space, assuming $O(\log n)$ space is required to store an edge or a vertex, since it stores a degree counter for every A-vertex and at most d_2 edges for each of the s vertices in the reservoir. **Algorithm 1** outlines pseudocode for `Degree-Based Reservoir Sampling` on a bi-partite graph.

Algorithm 1: `Degree-Based Resevoir Sampling`(d_1, d_2, s)

require: Lower degree bound $d_1 \in \mathbb{N}$, Upper neighbourhood bound $d_2 \in \mathbb{N}$, Reservoir size $s \in \mathbb{N}$, Insertion-only stream $\{(a_0, b_0), \dots, (a_n, b_n)\}$

```
D ← {}                                                    // Degree counter
R ← {}                                                    // Reservoir
S ← {}                                                    // Collected edges
x ← 0                                                     // # nodes of degree≥ d_1
for i ∈ [0, n] do
    D[a_i] ← D[a_i] + 1                                   // Increment degree counter
    if D[a_i] ≡ d_1 then
        // Consider inserting a_i to reservoir
        x ← x + 1
        if |R| < s then
            // Reservoir is not full
            R ← R ∪ {a_i}
        else
            // Reservoir is full
            if Bernoulli(s/x) then
                Let a' be a uniform random element of R       // Element to replace
                Delete edges in S incident to a'
                R ← (R\{a'}) ∪ {a}                            // Swap a_i and a'
    if a_i ∈ R then
        S ← S ∪ (a_i, b_i)                                    // Store edge
        if D[a_i] ≡ d_1 + d_2 then
            return edges in S incident to a_i                 // Success
return FAIL                                               // No sufficient neighbourhoods found
```

2.2 Proposed Algorithm

The `Neighbourhood-Detection Problem` can be solved using `degree-based reservoir sampling` sampling by setting $d_2 = \frac{d}{c}$. The question remains as to what to set d_1 and s to in order to get a high probability of success. In [1] it is proven that setting $s = \lceil \ln(n) \cdot n^{\frac{1}{c}} \rceil$ and using c `degree-based reservoir samplers` each with a different lower bound, incremented from $\frac{d}{c}$ to d stepping by $\frac{d}{c}$ each time, results in a high probability of success. The algorithm suggested by Dr. Konrad runs the samplers in this way, with s set to $\lceil \ln(n) \cdot n^{\frac{1}{c}} \rceil$. The proposed algorithm returns, uniformly at random, any of the neighbourhoods return by a successful sampling.

Each of the c samplers requires $O(n \log n + sd_2 \log n)$ space. Since the degree map can be shared between samplers the total space is $O(n \log n + c \cdot sd_2 \log n) = O(n \log n + c\lceil \log(n) n^{\frac{1}{c}} \rceil \frac{d}{c} \log n) = O(n \log n + n^{\frac{1}{c}} d \log^2 n)$. The proposed algorithm does require you to know the number of vertices in the graph stream before running the algorithm, or at least the log of it. If this is unknown then it is quick to run through the whole stream & build a set of vertices in the graph. This

requires $O(n \log n)$ space, assuming $O(\log n)$ space is required to store a vertex. In many real-world scenarios this number (or a good approximation) will be known due to it being important for other tasks.

Algorithm 2 is the algorithm proposed in [1] for solving the `Neighbourhood-Detection Problem` for insertion-only streams.

Algorithm 2: One-pass c-Approximation Insertion-Only Streaming Algorithm for `Neighbourhood Detection`

require: Space s, degree bound d.

$s \leftarrow \lceil \log(n) \cdot n^{\frac{1}{c}} \rceil$

for $i \in [0, c-1]$ **in parallel do**

  $(a_i, S_i) \leftarrow$ `Deg-Res-Sampling` $\left(\max\left\{1, i \cdot \frac{d}{c}\right\}, \frac{d}{c}, s\right)$

return Uniform random neighbourhood (a_i, S_i) from successful runs

2.3 Implementation

I centred my implementation around working for the graph streams from *SNAP* [5] as they came from real-world sources & thus are very important for testing the real-world practicalities of the proposed algorithm. These graph streams are not formatted in a strictly bi-partite way. I adjusted `degree-based reservoir sampling` to account for this by by repeating `lines 6-19` of **Algorithm 1** after the end of the `for` loop, with all occurrences of a_i replaced by b_i.

After an initial implementation I considered two alterations which could be made:

1. *Early Termination* - Returning the first neighbourhood of sufficient size, rather than uniformly sample from all which succeed; and,

2. *Shared Edge-Set* - Having all samplers share an edge-set, rather than each have one each.

Algorithm 2 states to run through the whole stream before sampling uniformly from the successfully returned neighbourhoods, but this is not necessary to fulfil `neighbourhood detection`. Returning the first neighbourhood of size $\frac{d}{c}$ is sufficient for solving the problem. Implementing this should reduce run-time as only part of the stream is being used and should reduce space used by the edge-sets in reservoir sampling as no data is stored after the first solution is found. As this change terminates at the first success there is no change to the success rate of the algorithm. One down side is that you will likely get less variety in the returned neighbourhoods as the algorithm will not encounter every vertex which meets the degree requirements, however this is not relevant to the problem.

I implemented this change and evaluated the implementation at different values of c. Space-requirements more than halved in every instance. For graphs with the least variation in degree distribution space-requirements decreased by more than 97% after early termination was introduced (as there are many vertices with valid neighbourhoods). Introducing early termination meant that run-times reduced as c increased, whereas before run-times increased with c. These results are shown in `Figure 1` & `Figure 2`. These are really good results, so I implemented this change going forward.

Lemma 2.1 If **Algorithm 2** terminates after the first sufficient neighbourhood it finds, no vertex occurs in multiple reservoirs at the same time.

Proof (by contradiction) Let v be a vertex in the graph stream. Suppose, without loss of generality, that v is a member of the reservoirs of both `Deg-Res-Sampling`$(\frac{d}{c}, \frac{d}{c}, s)$ & `Deg-Res-Sampling`$(\frac{2d}{c}, \frac{d}{c}, s)$. This means that v was sampled twice, after encountering the $\frac{d}{c}^{\text{th}}$ and $\frac{2d}{c}^{\text{th}}$ edge incident to v in the stream. After sampling v for the first time, encountered edges which are incident to v are stored. This means that by the time v is sampled for the second time $\frac{2d}{c} - \frac{d}{c} = \frac{d}{c}$ edges incident to v have been stored. This is a sufficient neighbourhood for **Algorithm 2** to terminate. This is a contradiction since termination occurs before the second sampling occurs. □

A consequence of **Lemma 2.1** is that an edge is only stored in multiple edge sets if its endpoints are sampled by different samplers. This means that introducing a shared edge-set for all samplers will not have a great effect on the space requirements of `Algorithm 2`.

However, the alterations to the implementation which would have to made to allow for a shared edge-set could introduce time savings. Namely, I would need a data-structure which tells me precisely which reservoir each vertex is in that so I know the appropriate value of d_1 in `Algorithm 1:Line 18`, otherwise improper termination could occur. An appropriate data-structure for this would be a `map` from each vertex to a bitstring whose indices indicate whether the vertex lies in an associated reservoir. With this implementation, inserting or deleting a vertex to/from a reservoir and querying whether a vertex is in a given reservoir becomes querying a map & then a known bit in a bitstring which both take $O(1)$ time. This `map` replaces the `ordered sets` currently used to store the reservoirs of each sampler and thus reduce these action times from $O(\log n)$ to $O(1)$ time. As these actions occur a lot this change should result in a reasonable reduction in run-time.

I implemented this map, along with a shared edge-set, and ran tests for values of $c \in [3, 100]$. These tests showed an 8% reduction in run-time when the map & shared edge-set where introduced. It should be noted that in a few cases the algorithm was slower but ¡1% which I think is a reasonable trade off. There were no gains in the space requirements when a shared edge-set was used, nor were there losses. The results can be seen in `Figure 3` & `Figure 4`.

Implementing a shared-edge set does not change the logic of the algorithm so does not affect its success rate. Due to the reduction in mean run-time & no noteworthy changes in space requirements, I recommend implementing this change.

2.4 Parameter Tuning

The changes discussed above have been to the implementation of `degree-bound reservoir samplers`, rather than to the proposed algorithm. This means these changes have had no affect on the success rate of the algorithm. I will now discuss potential changes to the proposed algorithm. These changes will have an effect on the success rate of the algorithm. It is possible that no change in the success-rate will be seen & rather the change will occur in execution time as more instructions have to be read before a solution is found. I am investigating the practicalities of the algorithm so keeping the run-times usable is a high priority. In `Algorithm 2` there are two parameters which can be played around with: the number of samplers; and, the size of the reservoirs, s.

The total sample size, $\left\lceil \ln(n) n^{\frac{1}{c}} \right\rceil \times \#$ Samplers, is the main factor effecting the space-used by the algorithm as this is the number of vertices it needs to store edges for. Noting that `Algorithm 2` sets the number of samplers to be c and that $cp \left\lceil \ln(n) n^{\frac{1}{c}} \right\rceil \leq c \left\lceil p \ln(n) n^{\frac{1}{c}} \right\rceil \ \forall\ p \in [0, 1]$ it is

apparent that reducing the number of samplers has a greater effect on total sample size, and thus space-used, than reducing sample size by the same proportion. For this reason I investigated the number of samplers first.

When changing the number of samplers we need to consider their lower bounds, d_1, too. If the range of vertex degrees which individual samplers sample, $[d_1, d_1 + d_2]$, at overlap then duplicate sampling of vertices can occur which offers no advantages & leads to a reservoir space being wasted. And, if there are gaps between these ranges then there are vertices which may not be sampled. Thus, the best set up is to have the sampling ranges be sequential with no gaps and so when varying the number of samplers we should only add or remove from the endpoints of these ranges. The samplers with lower lower-bounds, d_1, start sampling earlier & from a larger pool. I expect these early samplers to be more important wrt performance gains, than later samplers. It is not necessarily true that the first sampler is the most important as it is sampling from the largest pool and so has the greatest turn-over rate in its reservoir.

I ran the following tests to investigate what had a greater effect on performance, removing later samplers or earlier samplers.

1. Removing the first x samplers from `Algorithm 2` for $x \in [1, c)$, for $c \in [1, 20]$.
   i.e. Change the `for` loop in `Algorithm 2:Line 2` to run for $i \in [x, c-1]$.

2. Removing the last x samplers from `Algorithm 2` for $x \in [1, c)$, for $c \in [1, 20]$.
   i.e. Change the `for` loop in `Algorithm 2:Line 2` to run for $i \in [0, c-x]$.

These tests confirmed the hypothesis that the lower a sampler's lower-bound is, the greater the effect of its inclusion on performance. Time-taken increased as more early samplers were removed, and decreased as more late samplers were removed. Comparing the relative difference of these results shows that removing early samplers is more detrimental to performance than including later samplers. This is due to late samplers only starting to sample later & they may not have begun sampling by the time one of the previous samplers has succeed, terminating the algorithm. When the same proportion of samplers were removed for different values of the approximation factor, c, the relative performance was worse for greater values of c. This is to be expected as the real number of samplers removed increases with c. As expected the space-used decreased whenever more samplers were removed. During the second test some failures did occur when only the first sampler was used on larger graphs. The results to these tests on `gplus` can be seen in `Figure 5` & `Figure 6`, results for other graphs followed similar trends.

From these tests it is apparent that there are advantages to be found by removing late samplers, but not from removing the earliest. These tests did show that the failure rate does increase when too few of the early samplers are used. Thus I repeated the tests, this time testing when $\leq 25\%$ of the earliest samplers were used and ran more repetitions so I could evaluate the failure rate better.

This test showed that the success rate was less than 1 only when just the first sampler was used, even on the largest graphs. Even in these cases it was still above .9. Since failure only occurs when the whole stream is run through, runs which failed take a lot longer to run than those that succeed (5 minutes compared to 10 seconds for `gplus_large`). This meant that the implementations which only used the first two samplers were quicker on average.

The success rate when only the first sampler was used was lower for larger graphs, but was higher for greater values of the approximation factor. In all my tests, running just the first two samplers provided the best results. However, the largest graph I tested on only contains $100,000$ vertices so it is likely this may not hold for significantly larger graphs. I suggest using the first $\texttt{max}\left(2, \left\lceil \frac{1}{5} \ln n \right\rceil\right)$ samplers. As we don't want the sampling ranges to overlap we need to add a restriction that the number of samplers is never greater than the approximation factor, giving the number of samplers used as $\texttt{min}\left(c, \texttt{max}\left(2, \left\lceil \frac{1}{5} \ln n \right\rceil\right)\right)$.

When tested on `gplus_large` introducing this change halved run times for tight approximation factors ($c \in [3, 10]$), for looser approximation factors time-savings of over 90% were made. This is because the number of samplers removed increases as the value of the approximation factor increases. This implementation had little effect on space-usage when compared to the implementation before the number of samplers was reduced, whilst maintaining a perfect success rate. The lack of change in space-usage shows that these later samplers were not being used and were just adding wasted computation time. These results can be seen in `Figure 7` & `Figure 8`.

Removing later samplers had little effect on the space used by the algorithm. This is due to these samplers not getting the chance to sample very often as the algorithm often terminated before many vertices of sufficient degree were encountered. This means that reducing the size of those samplers would have had virtually no affect on the performance of the algorithm. This helps justify optimising the number of samplers before the sample sizes. The previous optimisation means that very few samplers are being run so it is unlikely that any great improvements will be found, if any, when reducing sample size.

Decreasing the sample size s means less edges are stored, reducing space requirements, but increases the turn-over rate of the elements in the reservoir as the probability a vertex is sampled is the same but the sample size is smaller. The greater turn-over rate is detrimental to the success of the algorithm as it more likely for a vertex to be removed when it is close to succeeding. For very low sample sizes it is likely that the success rate of the algorithm will remain high, but the run-times will be dramatically increased.

I tested having the sample size set to between 10% and 200% of the proposed size, at 10% increments, in order to get an overview of how sample size affects performance. This test showed a positive linear relationship between sample size & space-used. The only inference that could be drawn from sample-size & time-taken is that there is a point at which time-taken dramatically increases, this is the point where the algorithm fails to return a result. Overall, there is no clear strategy to improving the algorithm by varying the sample size. There may have been if we had not already reduced the number of samplers, but for reasons discussed earlier it is justified to optimise the number of samplers first. These results can be seen in `Figure 9` & `Figure 10`.

Here is **Algorithm 2** rewritten to include the proposed parameter optimisations

Algorithm 3: One-pass c-Approximation Insertion-Only Streaming Algorithm for `Neighbourhood Detection`

require: Space s, degree bound d.

$s \leftarrow \lceil \log(n) \cdot n^{\frac{1}{c}} \rceil$

$n_s \leftarrow \min\left(c, \max\left(2, \left\lceil \frac{1}{5}\ln(n) \right\rceil\right)\right)$

for $i \in [0, n_s]$ **in parallel do**

  $(a_i, S_i) \leftarrow$ `Deg-Res-Sampling` $\left(\max\left\{1, i \cdot \frac{d}{c}\right\}, \frac{d}{c}, s\right)$

return Uniform random neighbourhood (a_i, S_i) from successful runs

2.5 Evaluation

To evaluate my final implementation I compared its performance to that of different stages of the optimisation and to a naïve implementation of the algorithm. These different stages were

i) Initial Implementation - No early termination, shared edge-set or sampler removal.

ii) After Technical Optimisation - Implements early termination & a shared edge-set.

iii) After Algorithmic Optimisation - Implements early termination, shared edge-set and sampler removal.

The naïve algorithm I used is given in `Algorithmn 4`. This aproach records the neighbourhood of every encountered vertex & returns the first one which surpasses the $\frac{d}{c}$ threshold. This algorithm will always succeed as there is a requirement that at least one vertex in the graph is of degree, at least, d. As the naïve algorithm terminates at the first neighbourhood of sufficient size & uses all edges in the stream up to that point, it is guaranteed to find the first vertex with a sufficient neighbourhood. This means the naïve algorithm is very quick, provided its space-usage is within the memory allocated to the algorithm. In the worst case, where the last element is the only element of degree $\frac{d}{c}$ and all other $n-1$ elements have degree $\frac{d}{c}-1$, $\frac{d}{c}+(n-1)(\frac{d}{c}-1) \in O(n\frac{d}{c})$ space is required. The average case has the same complexity and the best case, where the first $\frac{d}{c}$ instructions are incident to the same vertex, has $O(\frac{d}{c})$ space-complexity.

Algorithm 4: Naïve Single-Pass Insertion-Only 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 $N[s_i]$ `// Insert neighbour`
  if size$(N[s_i]) \geq d$ **then**
    `// Sufficiently large neighbourhood found`
    $S \leftarrow \{N[s_i][0], \dots, N[s_i][\frac{d}{c}]\}$ `// First` $\frac{d}{c}$ `elements of neighbourhood`
    return (s_i, S)
return FAIL `// No neighbourhoods found`

The empirical results showed that the space usage for the naïve algorithm was significantly greater than that of any of the proposed implementations. This high usage of space could be seen in practice as the naïve algorithm did not return a solution when the approximation factor was 2 for `gplus_large`, even after 3 hours of computation. For all the graphs, except `1000star`, the implementations of the proper algorithm used less than the stream file size for all approximation values. For the largest graph tested, `gplus_large`, only 13.7% of the stream file size was required to find a 2-approximation using the initial implementation. After my optimisations this was reduced to 1.39%. The space-usage results for `gplus`, the largest file which the naïve algorithm was usable on, are given in `Figure 11` with the results normalised by the stream's size (51,839 KB).

As I supposed the naïve algorithm was generally the fasted algorithm, in the cases where it succeeded, as it always used the fewest number of instructions required. My final implementation, after algorithmic tuning, was between 50-100% slower than the naïve implementation in these cases. My final implementation is useably quick: when run on `gplus_large` it took 150s to find a solution when $c = 2$ and less than a minute for $c \geq 5$. The naïve algorithm being quicker is irrelevant for this evaluation as it fails to terminate for larger files due to its space inefficiency. The time taken results for `gplus`, the largest file tested, are shown in `Figure 12`.

2.6 Conclusion

During the optimisation process I made three changes which helped improve the performance of the algorithm: I made the algorithm return the first neighbourhood it encountered of sufficient degree. This reduced both time & space requirements as less of the graph stream was consumed; I set up the samplers to share an edge set. This was meant to reduce space requirements, but due

to early termination duplication of edges was now very rare. This change did reduce run times as the time complexity for operations on reservoirs was reduced from $O(\log n)$ to $O(1)$; And, I reduced the number of samplers being run by removing most of the late samplers which were very rarely used (due to early termination). This change had little effect on space requirements, but did reduce run-time substantially, especially for high values of the approximation factor. After all these changes the algorithm maintained a perfect success rate during all final tests.

I proposed changing the sample size of each reservoir. Preliminary tests presented no strategies which could consistently improve one performance metric without unreasonably affecting another. Thus this idea was not implemented. It is likely that if the number of samplers had not been reduced by as much as then varying the sample size could have had a positive effect on the algorithm, but the gains from reducing the number of samplers are so good they must be incorporated.

The final evaluation shows that the proposed algorithm is much more space-efficient that a naïve approach & even for close approximations of large streams my final implementation uses less than 6% of the file size and for $c \geq 5$ it uses $\leq 1\%$. This space efficiency increases the maximum file size at which the algorithm is usable, with without requiring virtual memory. The run-times of my final implementation are very usable with solutions being found in a couple of minutes for close approximations ($c \in [2, 4]$) and taking significantly less time for loser approximations. Whilst evaluating my final implementation no failures occurred. All these results show that the proposed algorithm for insertion-only streams is indeed practical in real-world scenarios.

Chapter 3

Insertion-Deletion Streams

3.1 L_0 Sampling

- L0 sampling uniformly samples from the non zero elements of a vector.
- In this setting we consider the edge vector of each vertex (indicates which vertices it shares an edge with).
- The formally proposed algorithm.

3.1.1 Implementation

- How to L0 sample a graph stream
- Hash functions are hard (trade off due to dodgy hash generation).

3.2 The Algorithm

- There are two algorithms.
- Scenarios when each algorithm is most suitable.
- L0 samplers in parallel (lots of pre-allocated space).

Algorithm 5: One-pass c-approximation Insertion-Deletion Streaming Algorithm for `Neighbourhood Detection`. (Vertex Sampling)

require: Space s, degree bound d.

Let $x = \max\left\{\frac{n}{c}, \sqrt{n}\right\}$

Sample a uniform random subset $A' \subseteq A$ of size $10\ x \ln n$ of vertices.

for $a \in A'$ **do**

  Run $10\frac{d}{c}\ln n$ l_0-samplers on the set of edges incident to a.

return Any neighbourhood of size $\frac{d}{c}$ among the stored edges, if there is one.

Algorithm 6: One-pass c-approximation Insertion-Deletion Streaming Algorithm for `Neighbourhood Detection`. (Edge Sampling)

require: Space s, degree bound d.

Let $x = \max\left\{\frac{n}{c}, \sqrt{n}\right\}$.

Run $10\frac{nd}{c}\left(\frac{1}{x} + \frac{1}{c}\right)\ln(nm)$ l_0-samplers on the set of edges incident to a.

return Any neighbourhood of size $\frac{d}{c}$ among the returned edges, if there is one.

3.3 Implementation

- L0 samplers in parallel
  - require lots of preallocated space (know how much space is used at start of algorithm)
- Require list of vertices before vertex sampling.

3.4 Parameter Tuning

- gamma & delta for L0 sampling
- vertex sample size & samplers per vertex
- num edge samplers

3.5 Evaluation

Algorithm 7: Naïve Single-Pass Insertion-Streaming Algorithm for Neighbourhood Detection

require: Stream $\{(w_0, s_0, t_0) \dots (w_n, s_n, t_n)\}$, degree bound d, precision bound c

$N \leftarrow \{\{\}\}$ {neighbourhoods}
for $i = 0 \dots n$ **do**
  if $w_i \equiv 1$ **then**
    append t_i to $N[s_i]$ `// Insertion instruction`
  else if $w_i \equiv -1$ **then**
    remove t_i from $N[s_i]$ `// Deletion Instruction`
`// Look for a sufficiently large neighbourhood`
for $s_i \in N.keys$ **do**
  if size$(N[s_i]) \geq d$ **then**
    `// Sufficiently large neighbourhood found`
    $S \leftarrow \{N[s_i][0], \dots, N[s_i][\frac{d}{c}]\}$ `// First` $\frac{d}{c}$ `elements of neighbourhood`
    return (s_i, S)
`// No neighbourhoods found`
return FAIL

3.6 Conclusion

Chapter 4

Conclusions

4.1 Future Work

Chapter 5

Related Works

Chapter 6

Appendix

THESE ARE ALL PLACEHOLDER

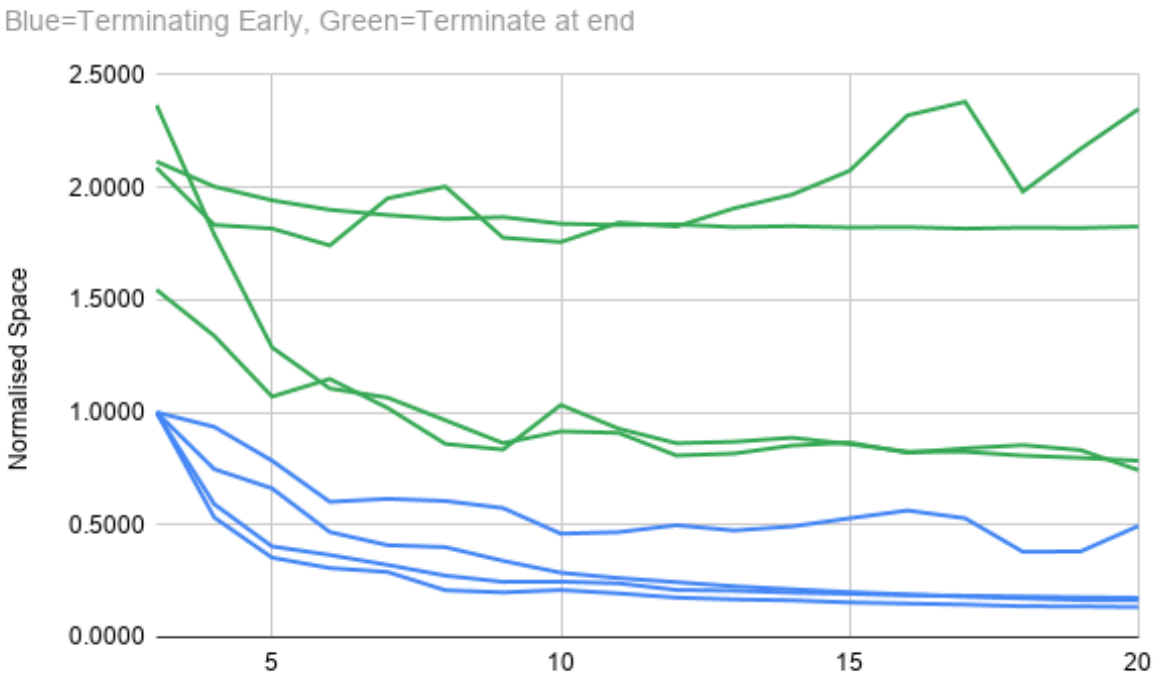

Figure 1: Normalised mean space used for different termination strategies

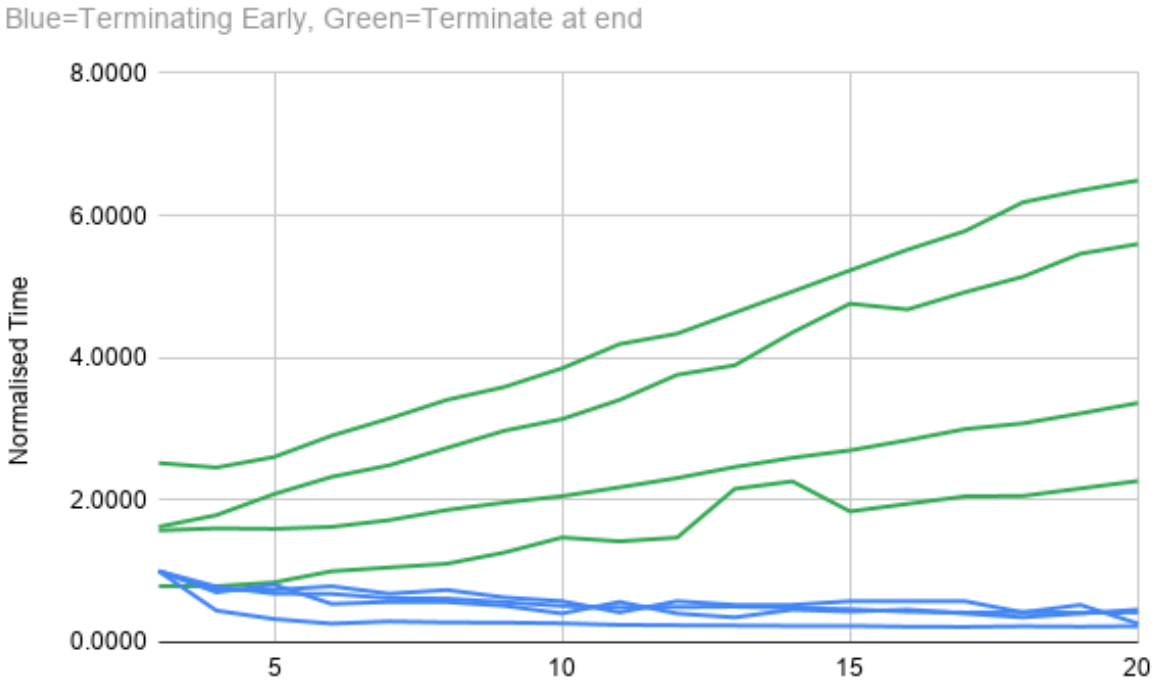

Figure 2: Normalised mean time taken for different termination strategies

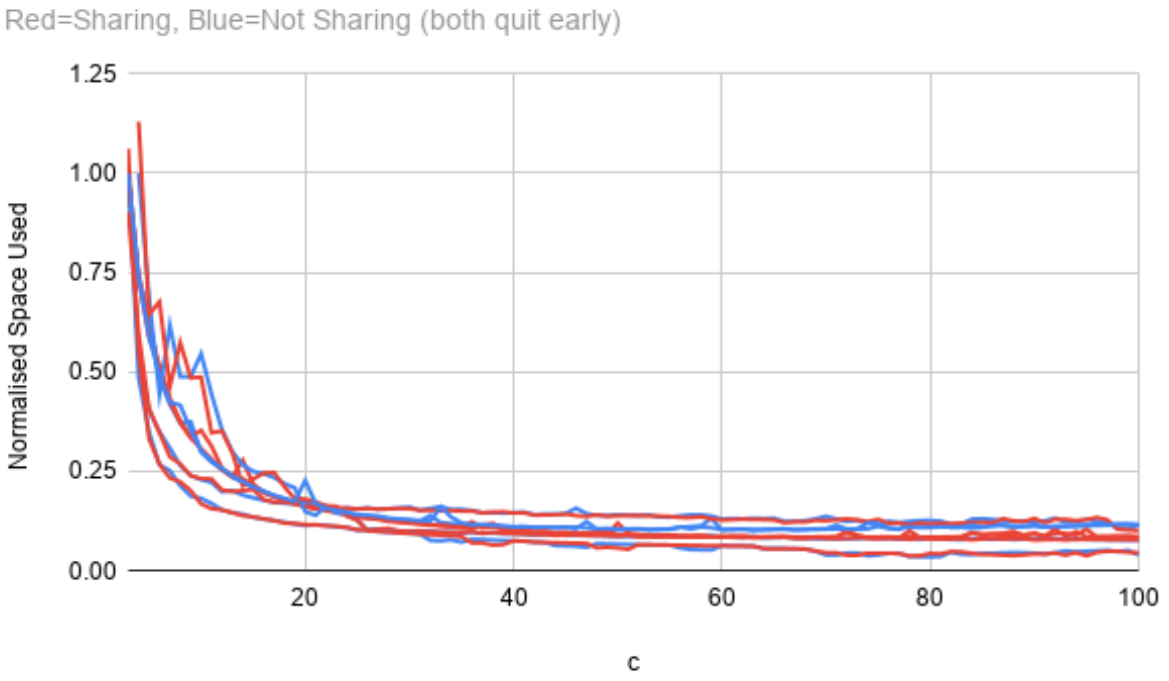

Figure 3: Normalised mean space used for different edge storage strategies

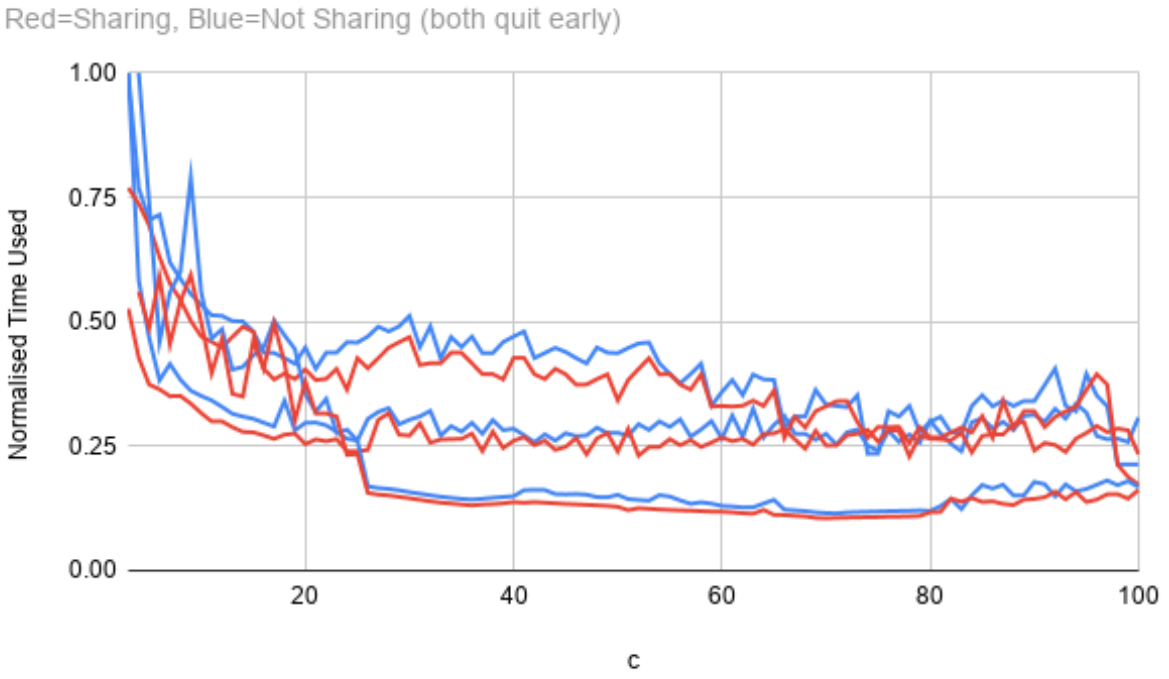

Figure 4: Normalised mean time taken for different edge storage strategies

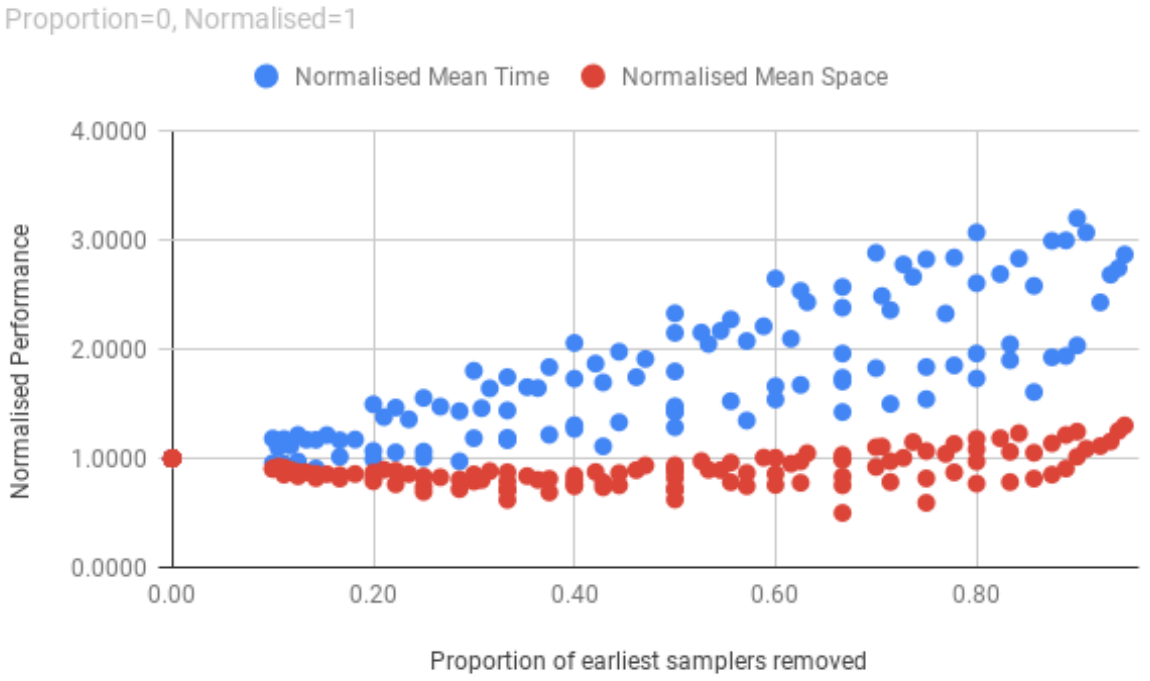

Figure 5: Normalised mean time-taken & space-used when removing different proportions of the early samplers on graph `gplus`

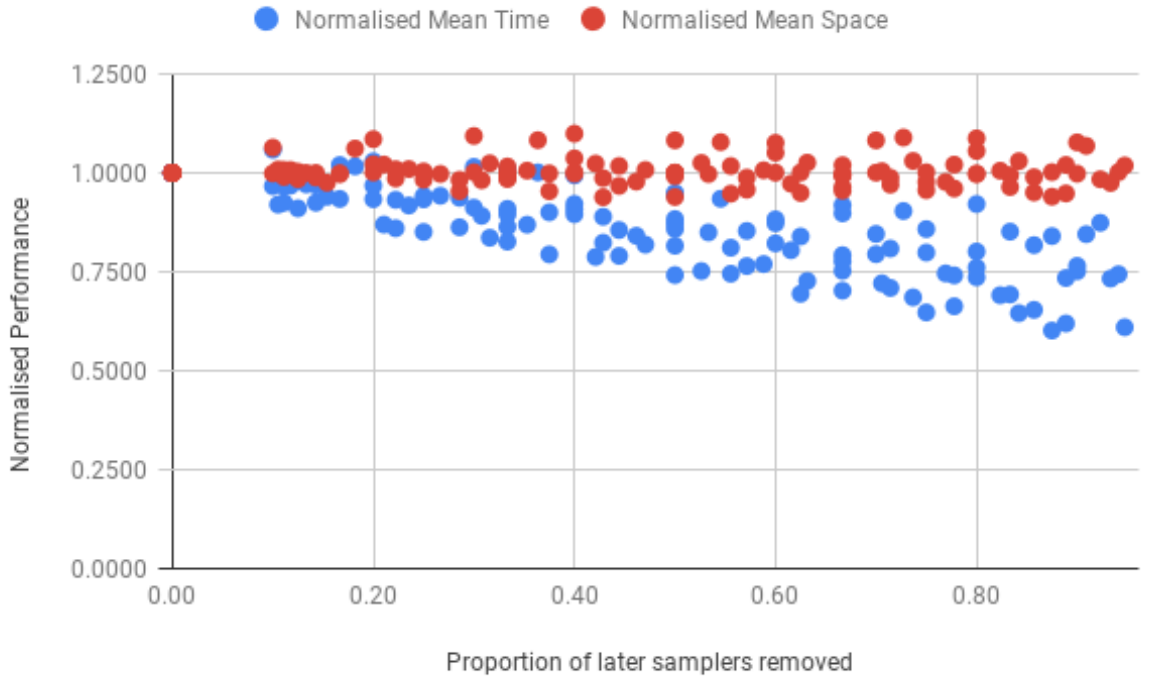

Figure 6: Normalised mean time-taken & space-used when removing different proportions of the late samplers on graph `gplus`

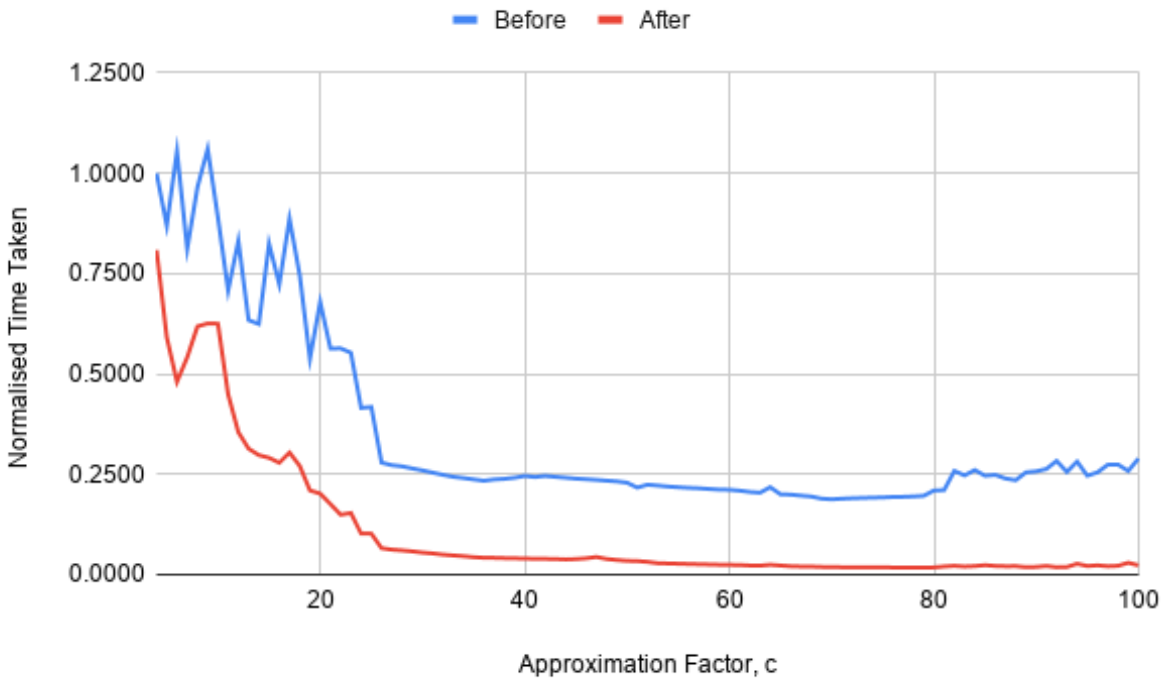

Figure 7: Time-taken before & after removing late samplers from `gplus_large`

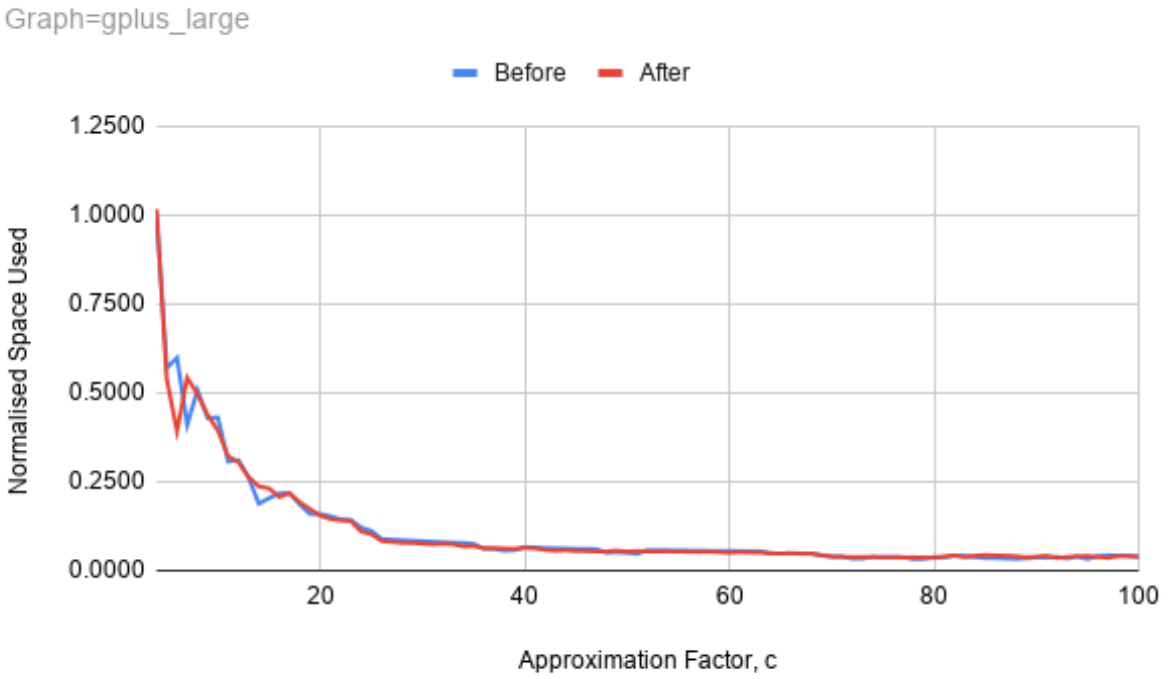

Figure 8: Space-used before & after removing late samplers from `gplus_large`

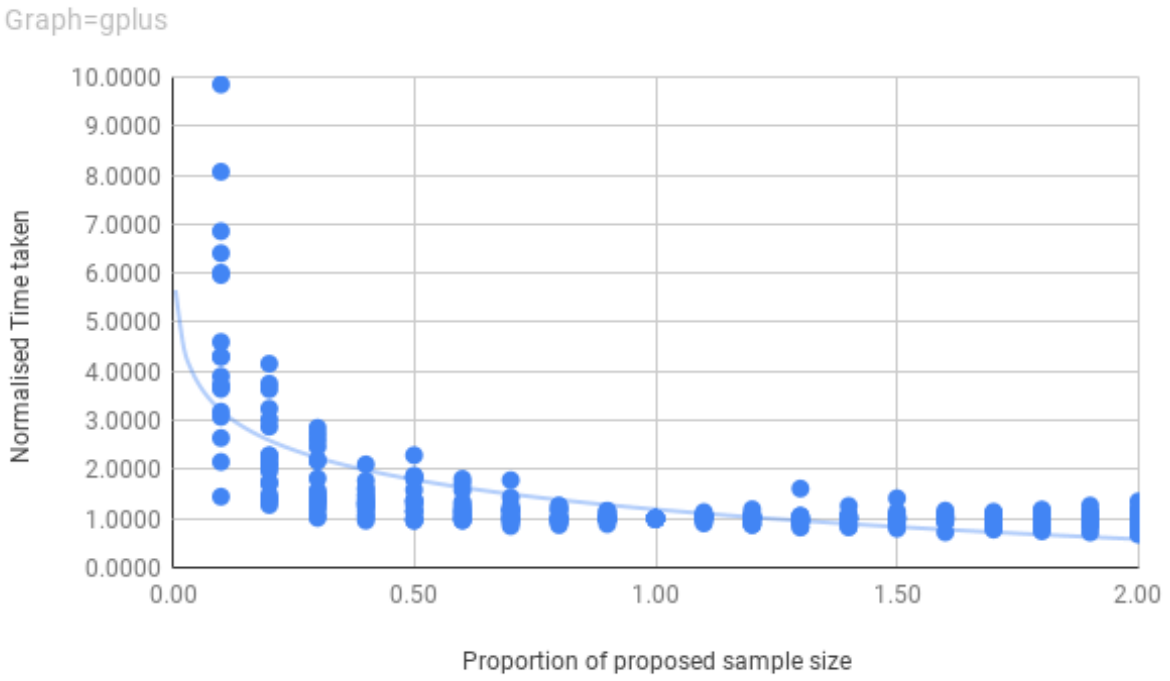

Figure 9: Time-taken when varying the reservoir sample size.

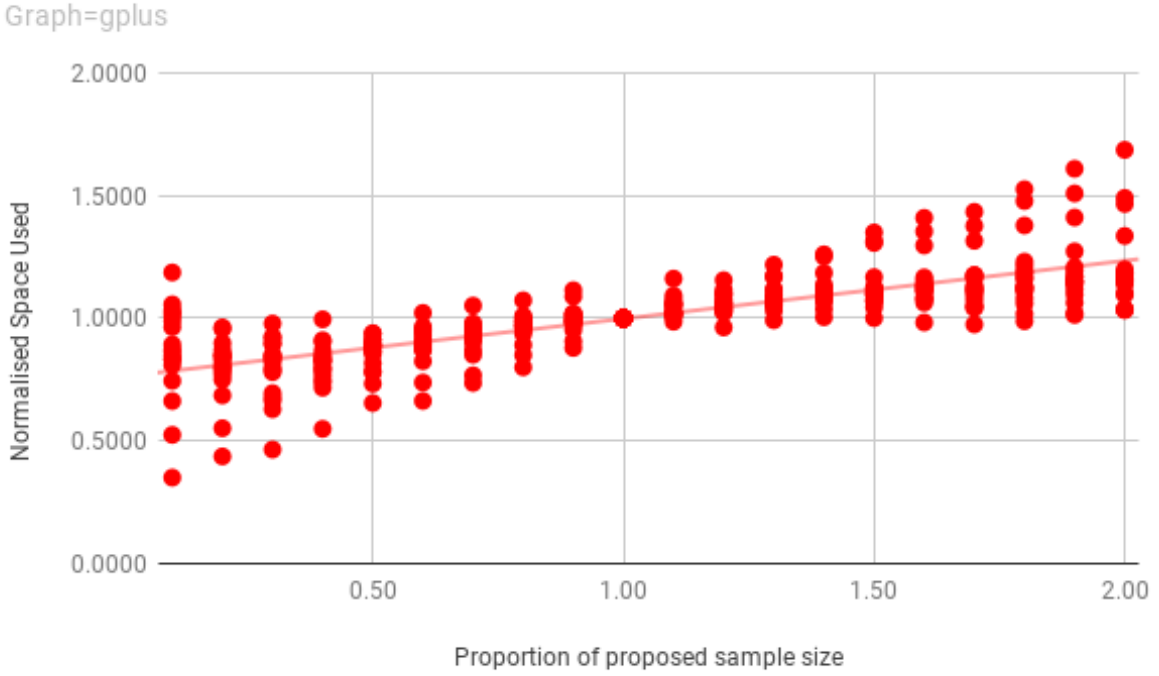

Figure 10: Space-used when varying the reservoir sample size.

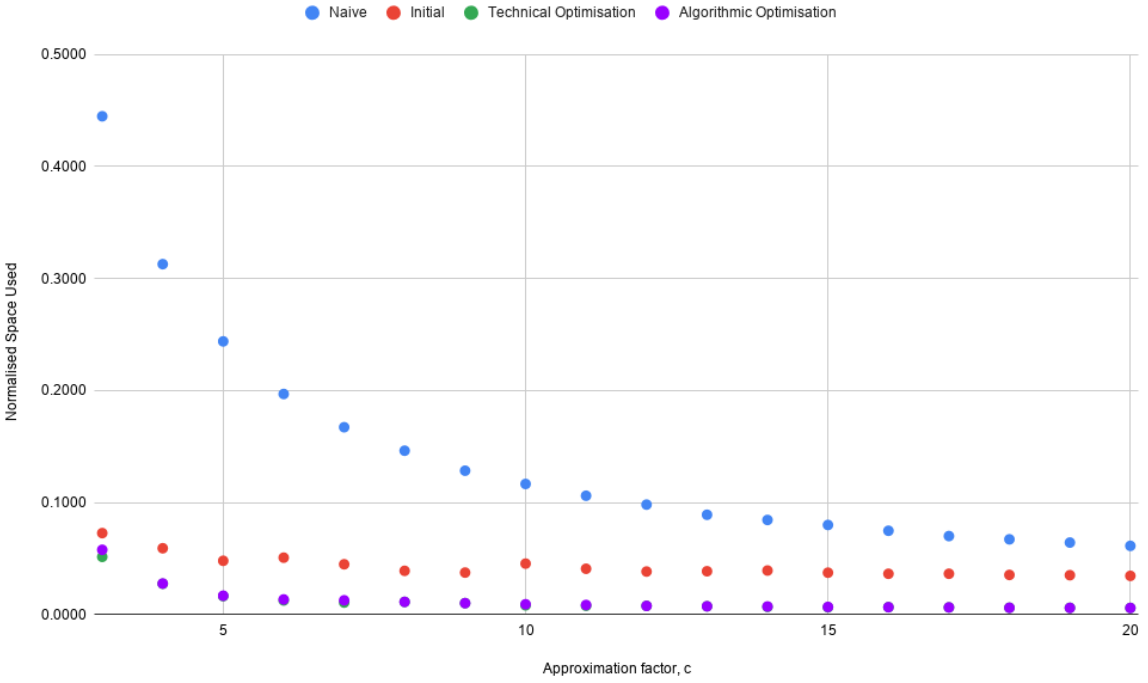

Figure 11: Space-used by different implementations of the algorithm for insertion-only streams when tested on `gplus`, normalised by stream size 51,839 KB

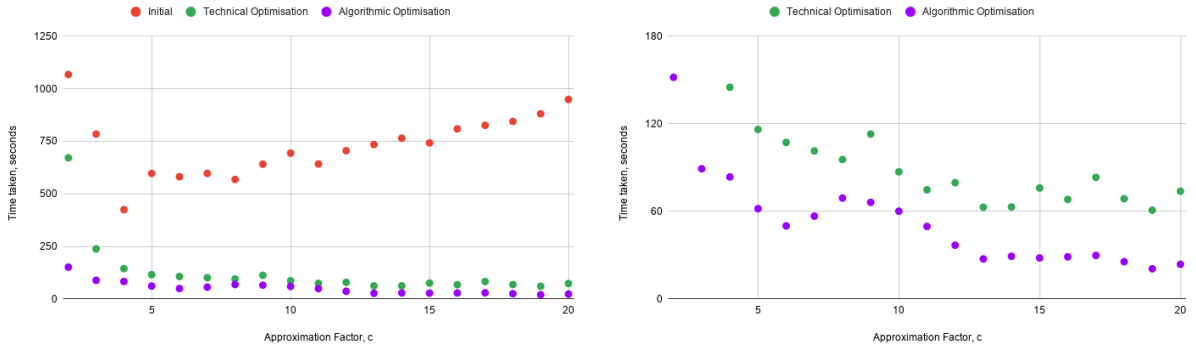

Figure 12: Time-taken by different implementations of the algorithm for insertion-only streams when tested on `gplus_large`

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/