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

Dom Hutchinson

May 3, 2020

Declaration

Contents

Abstract

The `Neighbourhood Detection Problem` is a problem in graph theory which tasks one with finding a vertex in the graph which has a certain number of neighbours and to then return a subset of these neighbours. This is a trivial problem to solve in theory, but in practice is hard to solve within a time frame which makes the implementation practical. A major reason for this is that many approaches to solving the problem require a lot of space and so quickly overflow a computer's RAM, slowing the execution significantly down. This common space inefficiency is becoming increasingly problematic since the move towards Big Data.

In this paper I discuss and implement the two algorithms presented in [1] which offer space efficient solutions for `neighbourhood detection` for two types of graph stream. During the implementation I discuss and evaluate several methods for improving performance of these algorithms, focusing on space-efficiency in order to maximise the potential size of graphs the implementations are effective on.

The final implementation of the proposed algorithm for insertion-only streams proves very promising with it able to find large neighbourhoods in a couple of minutes for a graph with 30 million edges. The algorithm for insertion-deletion streams proves much more difficult to evaluate due to restrictions incurred from the implementation of L_0 samplers.

Acknowledgments

Introduction

I.i Motivation

Graphs are used to represent relationships between objects and are popular in data science. A graph stream is a sequence of instructions which describe how to construct a graph. The sequential nature of a graph stream allows it to represent the evolution of a graph over time, which adjacency matrices cannot. This makes graph streams ideal for representing changing networks such as a social or computer network where objects can connect and then disconnect.

A basic problem in graph theory is `High Degree Detection` where you want to find the vertices with the greatest degree. This problem can be trivially and efficiently solved by counting the number of edges incident to each vertex and then returning any vertices of sufficient degree. Solving `High Degree Detection` allows for the identification of the most influential vertices in a graph, however you cannot make any inferences about the nature of this influence. In a social network it is possible for someone to have lots of followers (*i.e.* be of high degree), but for all these followers to be bot accounts meaning the account actually has zero influence.

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.

`Neighbourhood Detection` is the natural set up from `High Degree Detection`, it tasks you with finding vertices of high degree and a subset of their neighbours. Solving `Neighbourhood Detection` allows for inferences to be made about the influence a vertex has. Extending the social network example, `Neighbourhood Detection` allows for analysis of the followers of a popular account which can be used for targeted marketing campaigns.

`Neighbourhood Detection` can be solved trivially by storing the neighbourhood of every vertex and then returning any subset of size $\frac{d}{c}$. This trivial solution is very space inefficient and, with the movement towards Big Data, would quickly overflow a computer's RAM. This results in the use of virtual data which significantly slows down data access time and thus the whole algorithm. This motivates the need for space efficient solutions to `Neighbourhood Detection`.

I.ii Motivating Applications

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

- Given a list of connections in a social network, identify *social influencers* and determine the demographics of their audience in order to decide who should promote a particular product.

- Given a list of receipts, identify which items are commonly sold together and use this information to determine how to stack shelves in order to increase sales.

- Given a log of traffic within a network identify which resources are being accessed most often, and by whom. This information can be used to determine what upgrades should be made to the network and to identify potential attacks on the network.

I.iii Objectives

The objectives of this project are to:

- Implement and understand the algorithms proposed in [1].
- Discuss and test alterations to the logic and implementation of the proposed algorithms, and the subroutines they rely upon, in order to improve the performance of the proposed algorithm.
- Use large graphs from real-world scenarios to evaluate the practicalities of these implementations on real-world data sets.

I.iv Structure

Having provide some background to the problem, the structure of this project is as follows. In `Chapter 1` I discuss and define general ideas, technologies and techniques which are used throughout this project. In `Chapter 2` I discuss, implement and evaluate the algorithm proposed in [1] for insertion-only graph streams. In `Chapter 3` I discuss, implement and evaluate the algorithm proposed in [1] for insertion-deletion graph streams. And, in `Chapter 4` review what was achieved in this project and provide some ideas for possible future work to build upon those achievements.

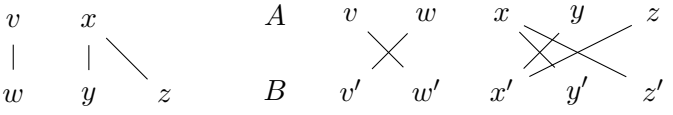
Chapter 1

Preliminaries

1.1 Definitions

A *Graph* is a data structure used to represent pairwise relationships between objects. A Graph $G = (V, E)$ is defined to have a vertex set V, which holds the objects, and an edge-set E, which holds the relationship between the objects. Graphs are traditionally visualised with circles for each vertex and lines between vertices that share an edge. There are variations on graphs which allow for edges to have direction and weight. In this project we are using undirected, unweighted graphs meaning edges can be represented by an unordered pair of vertices (u, v) for $u, v \in V$.

A graph $G = (V, E)$ is said to be a *Bipartite Graph* if its vertex set V can be partitioned into two disjoint subsets A, B and the every edges connects a vertex in A to a vertex in B. Formally, $\exists\ A, B \subseteq V$ st $A \cup B = V,\ A \cap B = \emptyset$ and $\forall\ (a, b) \in E$ we have that $a \in A, b \in B$. *Bipartite Graphs* are denoted as $G = (A \cup B, E)$.



(a) Graph (b) Bipartite Representation

Figure 1.1.1: Example of how all graphs have a bipartite representation

All graphs $G = (V, E)$ have a bipartite representation $G' = (A \cup B, E')$. Considering the example in `Figure 1`, defining $A = V$, B to a be a copy of V, and the edge set $E' = \{(u, v') : (u, v) \in E, u \in A, v' \in B\}$ generates a bipartite presentation of G. The bipartite representation has twice the number of edges since each edge (u, v) is now replaced by two edges (u, v') and (v, u').

The *Neighbourhood* of a vertex v is the set of vertices which share an edge with, $N_v = \{u : (u, v) \in E\}$. The *Degree* of a vertex v is the size of its neighbourhood, $\delta_v = |N_v|$. A vertex v is *s-sparse* if its degree is less than, or equal to, s.

The *Edge Vector* of a vertex v of length $|V|$ where each index gives the weight of the edge from v to the vertex associated with that index. Typically a weight of 0 is stored if no edge exists between v and a specific vertex. The graphs used in this project are unweighted so the values in the edge vector are booleans describing whether, or not, an edge exists. These edge vectors can be stored as bitstrings.

Graph Streams are an unbounded sequence of instructions $\{e_1, e_2, \dots\}$ which describe how to construct the graph. These instructions e_i describe modifications to the edges of a graph. The vertex set of the described graph is inferred from the instructions. In this project we are only concerned with two types of graph stream.

- *Insertion-Only Streams* where each instruction inserts a new edge to the graph. It is assumed that no edges are inserted multiple times. For the unweighted, undirected graphs used in this project insertion-only streams are a list of the edges in the graph and can be read in any order. Here the instructions have the form $e_i = (u, v)$ where $u, v \in V$ are the endpoints of the edge being inserted.

- *Insertion-Deletion Streams* where each instruction either adds a new edge to the graph or removes an existing edge from the graph. It is assumed that no edges are inserted multiple times, nor is an edge removed if it is not currently in the graph. For the unweighted, undirected graphs used in this project instructions of insertion-deletion streams take the form $e_i = (\Delta, u, v)$ where $u, v \in V$ are the endpoints of the edge and Δ is a boolean defining whether e_i is an insertion or deletion instruction. The order of instructions is clearly important for insertion-deletion streams.

Other versions of graph streams exist which allow for changes to the weights of edges and directed edges, but these are out of scope for this project.

Streaming Algorithms are algorithms which are designed to take a stream of sequential instructions as their input. Streaming algorithms are space-efficient since only one instruction is read at a time, requiring constant space. Streaming algorithms which only require a single pass of the instruction stream are called *Online Algorithms*. Online algorithms have the advantage of being able to take new instructions without having to recompute anything.

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.

A family of hash functions $\mathcal{H} = \{h : X \to Y\}$ is described as being *Pairwise Independent* if $\forall\ u, v \in X$ with $u \neq v$ and $\forall\ a, b \in Y$ if h is chosen uniformly at random from $\mathcal{H}$ then $\mathbb{P}(h(u) = a \wedge h(v) = b) = \frac{1}{|Y|^2}$. This means that the values u, v are hashed to are assigned uniformly at random and pairwise independently.

A family of hash functions $\mathcal{H} = \{h : X \to Y\}$ is described as begin *k-Wise Independent* if for any k distinct elements of X $\{v_1, \ldots, v_k\}$ and any, not necessarily distinct, elements of Y $\{a_1, \ldots, a_k\}$ if h is chosen uniformly at random from $\mathcal{H}$ then

$$\mathbb{P}(h(v_1) = a_1 \wedge \cdots \wedge h(v_k) = a_k) = \frac{1}{|Y|^2}$$

1.2 Technologies

For the implementations in this project I chose to use C++, specifically `C++11`. I initially considered using either python or C++, as they both work in the object-orientated paradigm and I am proficient in both. I chose C++ as it allows for more control with memory management and typically has faster run-times. For this project the memory management is more important as I am looking to evaluate the space-efficiency of algorithms. The faster run-times is important when evaluating the practicalities of the algorithms.

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 Approach

During evaluation three performance metrics were measured: 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. During most tests the success-rate was near perfect, but the other metrics were significantly higher as it took longer for a solution to be found.

In real world scenarios we are most interested analysing the neighbourhoods of the most influential vertices. For this reason d is set to the maximum degree in the graph during all tests. Further, the greater the proportion of a neighbourhood returned the better the inferences made are so I focussed testing on low values of c, namely $c \in [1, 20]$.

For this project I wanted some graphs generated from the real-world. I found a collection of large graph streams, from different applications, in the `Stanford Network Analysis Project` (`SNAP`) datasets collection [2]. I chose to use the graphs generated from social networks as I found that application most motivating, but this choice is essentially arbitrary. The graph streams from [2] were all insertion-only so I wrote a utility program which created insertion-deletion streams from them by using a Bernoulli random variable to decide whether to delete an edge immediately after inserting it. **Table 1** provides details about the graphs used during evaluation.

Name	Type	# Vertices	# Edges	# Instructions	Max Degree	File Size
facebook_small	Insertion-Only	52	146	-	18	3 KB
facebook_small_deletion	Insertion-Deletion	52	131	161	16	5 KB
facebook	Insertion-Only	747	30,025	-	293	587 KB
facebook_deletion	Insertion-Deletion	747	26,718	33,332	267	846 KB
gplus	Insertion-Only	12,417	1,179,613	-	5,948	12 MB
gplus_deletion	Insertion-Deletion	12,417	1,049,309	1,309,917	4,998	16 MB
gplus_large	Insertion-Only	102,100	30,238,035	-	104,947	1.3 GB

Table 1: Details of graphs used during evaluation

Chapter 2

Insertion-Only Streams

2.1 Degree-Based Reservoir Sampling

`Degree-Based Reservoir Sampling` is a technique for uniformly sampling a vertex from the set of vertices whose degrees are greater than a specified minimum bound d_1 along with d_2 of its neighbours. This is achieved by maintaining a sample of this set of vertices, known as the *reservoir*, and storing the first d_2 of the edges incident to a vertex after it is sampled. The size of the reservoir is a parameter of the algorithm. **Algorithm 1** outlines pseudocode for performing `Degree-Based Reservoir Sampling` on a bipartite graph using an insertion-only graph stream.

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

require: Degree bound $d_1 \in \mathbb{N}$, Neighbourhood bound $d_2 \in \mathbb{N}$, Reservoir size $s \in \mathbb{N}$, Insertion-only stream $\{(a_0, b_0), \ldots, (a_n, b_n)\}$

$D \leftarrow \{\}$ `// Degree counter`
$R \leftarrow \{\}$ `// Reservoir`
$E \leftarrow \{\}$ `// Collected edges`
$x \leftarrow 0$ `// # nodes of degree`$\geq d_1$
for $i \in [0, n]$ **do**
  $D[a_i] \leftarrow D[a_i] + 1$ `// Increment degree counter`
  if $D[a_i] \equiv d_1$ **then**
    `// Consider inserting` a_i `to reservoir`
    $x \leftarrow x + 1$
    if $|R| < s$ **then**
      `// Reservoir is not full`
      $R \leftarrow R \cup \{a_i\}$
    else
      `// Reservoir is full`
      if `Bernoulli`$\left(\frac{s}{x}\right)$ **then**
        Let a' be a uniform random element of R `// Element to replace`
        Delete edges in E incident to a'
        $R \leftarrow (R \backslash \{a'\}) \cup \{a\}$ `// Swap` a_i `and` a'
  if $a_i \in R$ **and** $D[a_i] < d_1 + d_2$ **then**
    $E \leftarrow E \cup (a_i, b_i)$ `// Store edge`
if $\exists\ a \in R$ with $D[a] \geq d_1 + d_2 - 1$ **then**
  return Uniform random vertex & neighbourhood from those of size d_2
else return `FAIL`

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

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. This invariant is proved as **Lemma 2.1**, first I shall describe how the reservoir is maintained by controlling how vertices are inserted. The first time the degree counter for a vertex v surpasses d_1, v is considered for insertion into the *reservoir*. There are two cases

- If the reservoir is not full (*i.e.* $|R| < s$) then v is inserted into the reservoir.
- Otherwise, if the reservoir is already full, a Bernoulli random variable with probability $p = \frac{x}{s}$ is used, where x is the number of vertices known to have degree at least d_1. If this random variable succeeds then pick, uniformly at random, a vertex u currently in the reservoir and replace it with v. All the edges in E which are incident to u, and no other vertex in the reservoir, are removed from E.

Whenever an edge (u, v) is encountered and u is in the reservoir and the number of edges stored in E incident to u is less than d_2 then (u, v) is inserted into (u, v).

Lemma 2.1 At any time t, R contains a uniform sample of the vertices whose degrees are known to be at least d_1.

Proof (by induction)
Let X_t be the set of vertices whose degrees are know to be greater than d_1 after t instructions have been consumed and $x_t := |X_t|$.
Base Case - $x_t \leq s$.
Trivially true since R is not yet full so $R = X_t$.
Inductive Case - $x_t > s$.
Assume the property holds for $x_t = z > s$, then the probability of any given element of X_t is in R is $\frac{1}{z}$. Without loss of generality, let a be the next element considered for sampling and b be any element of R. The probability a is sampled is $\frac{s}{z+1}$ and, given a is to be sampled, the probability b is chosen for removal is $\frac{1}{s}$. Thus, the probability that b is replaced by a is $\frac{s}{z+1} \cdot \frac{1}{s} = \frac{1}{z+1}$. Similarly, the probability b is not replaced by a is $1 - \frac{1}{z+1} = \frac{z}{z+1}$. By the inductive hypothesis, the probability that b was in R was $\frac{1}{z}$. Thus, the probability that b is still in R is $\frac{1}{z} \times \frac{z}{z+1} = \frac{1}{z+1}$.

Hence by mathematical induction, R contains a uniform sample of the vertices of sufficient degree at all points in time. □

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 and v is inserted into it, v is replacing an element that was in the reservoir meaning any progress made towards finding a neighbourhood for that vertex is annulled. 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 discussed further during evaluation.

`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 bipartite 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 $\mathcal{O}(n \log n + sd_2 \log n)$ 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.

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 achieve a high probability of success.

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.
1 $s \leftarrow \lceil \log(n) \cdot n^{\frac{1}{c}} \rceil$
2 **for** $i \in [0, c-1]$ **in parallel do**
3 $\quad (a_i, S_i) \leftarrow$ `Deg-Res-Sampling` $\left(\max\left\{1, i \cdot \frac{d}{c}\right\}, \frac{d}{c}, s\right)$
4 **return** Uniform random neighbourhood (a_i, S_i) from successful runs

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 samplers are run in parallel so that the algorithm only requires a single pass of the stream. Implementing parallel running of the samplers just requires passing the same instruction to each sample before fetching the next instruction.

Each of the c samplers requires $\mathcal{O}(n \log n + sd_2 \log n)$ space. Since the degree map can be shared between samplers the total space is $\mathcal{O}(n \log n + c \cdot sd_2 \log n) = \mathcal{O}(n \log n + c\lceil \log(n) n^{\frac{1}{c}} \rceil \frac{d}{c} \log n) = \mathcal{O}(n \log n + n^{\frac{1}{c}} d \log^2 n)$. 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 $\mathcal{O}(n \log n)$ space, assuming $O(\log n)$ space is required to store a vertex. However, in many real-world scenarios this number (or a good approximation) will be known due to it being important for other tasks.

2.3 Implementation

I centred my implementation around working for the graph streams from *SNAP* [2] 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 bipartite way. I adjusted `degree-based reservoir sampling` to account for this by by repeating `lines 6-17` (and the early termination clause) of **Algorithm 1** after the end of the `for` loop, with all occurrences of a_i replaced by b_i.

I tested changes to the implementation on multiple graphs, for brevity and succinctness I shall only show the results for `gplus`. `Figure 2.3.1` shows the results of testing the initial implementation of **Algorithm 2**. These results shows that the space requirements of **Algorithm 2** are significantly less than the size of file stream and that the space requirements decrease as the approximation factor increases but there is an undesirable result that the time taken increases as the approximation factor increases.

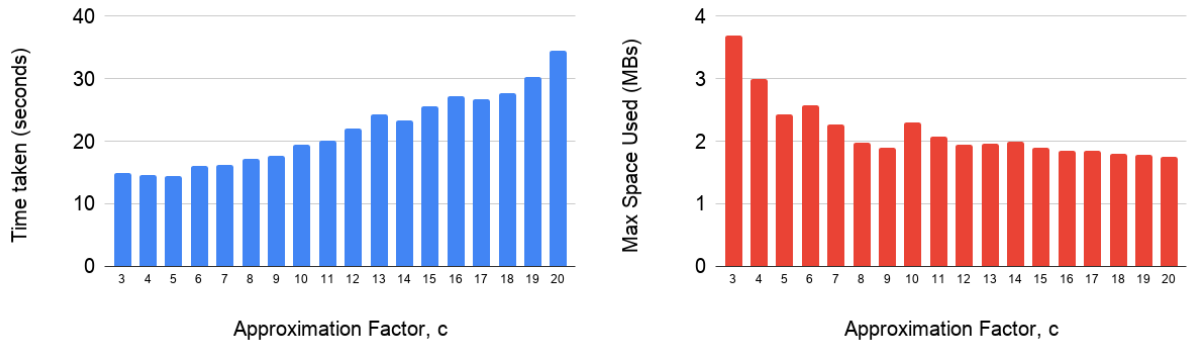


Figure 2.3.1: Results from testing initial implementation of **Algorithm 2** on `gplus` graph for different approximation factors

After this 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 1 states that the whole stream should be run through before the sampler returns a result uniformly from those which succeeded. This is unnecessary to fulfil `neighbourhood detection` as returning the first successful result would be sufficient. Similarly **Algorithm 2** states that all samplers should be run to completion and then a result is returned from those which succeeded. Again, this is unnecessary as return the first encountered neighbourhood of sufficient size would suffice for solving `neighbourhood detection`. These early terminations can be implemented by terminating the algorithm after it finds a vertex of degree $d_1 + d_2 - 1$ which is in a reservoir (remembering that d_1 is different for each reservoir).

Implementing early termination should reduce run-time as only part of the stream is being consumed 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.

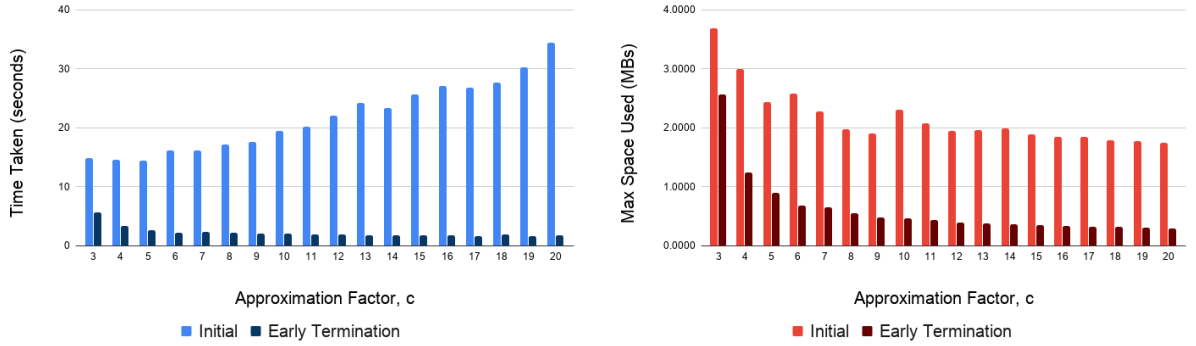


Figure 2.3.2: Results when using early termination, tested on `gplus` graph for different approximation factors. Against the results for the initial implementation.

`Figure 2.3.2` shows the results implementing early termination, against not implementing it, for graph `gplus` at different approximation factors. The time taken now decreases as c

increases and is more than halved in all cases. The space-requirements are down in every case and are down more than 90% for reasonable values of c. Both metrics now converge quickly, to $\sim 1\frac{1}{2}5$ seconds and 3.3MBs respectively, meaning we can make predictions on the performance of the algorithm for greater values of c. These are really good results, so I implemented this change going forward.

Lemma 2.2 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 2.2** 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 have little 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 $\mathcal{O}(1)$ 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 $\mathcal{O}(1)$ time. As these actions occur a lot this change should result in a reasonable reduction in run-time.

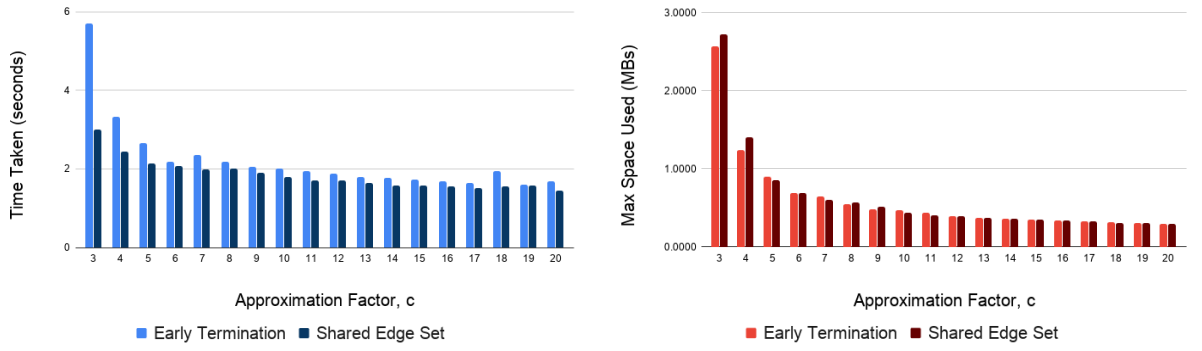


Figure 2.3.3: Results when using early termination and a shared edge set, tested on `gplus` graph for different approximation factors. Against the results just using early termination.

`Figure 2.3.3` shows the results when implementing both early termination and a shared edge set, against only using early terminiation. These tests showed a consistent reduction in run-time when both strategies are used. there is a slight increase in space usage for low values of c but this difference disappears very quickly so is of little concern. 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 technical implementation of the proposed algorithm, rather than to the its logic. This means these changes have had no affect on the success rate of the algorithm. I will now discuss potential changes to the logic of the proposed algorithm. These changes will affect 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 reservoir 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 reservoir size, and thus space-used, than reducing reservoir size by the same proportion. For this reason I investigated the number of samplers first.

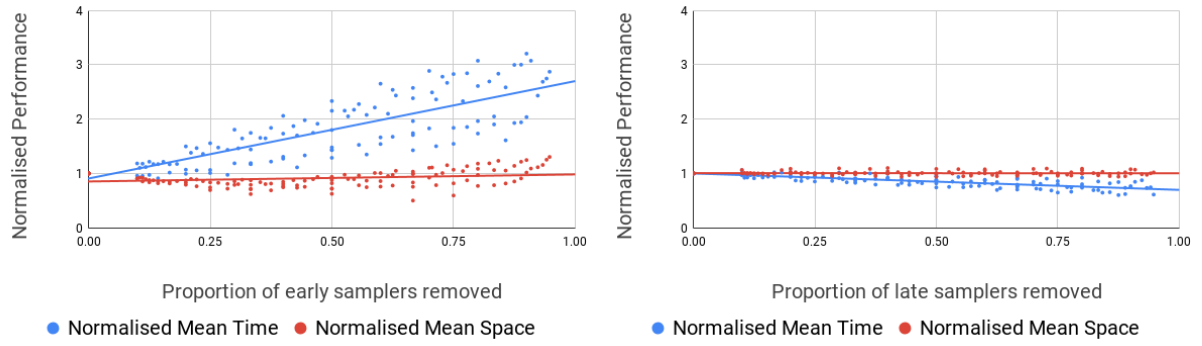
2.4.1 Number of Samplers

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 $i \in [0, c - x]$.



(a) Removing early samplers

(b) Removing late samplers

Figure 2.4.4: Results when removing different proportions of samplers, either from the start or end of the sampler range, for different values of c. These results are normalised with the values when no samplers are removed being 1 for that value of c. Results from graph `gplus`.

`Figure 2.4.4` shows the results of tests, with the results normalised to make them comparable. These results confirm 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. The space used was not affected much by the removal of samplers, although there is a slight increase in space usage when more early samplers were removed. Comparing the normalised results shows that removing early samplers is more detrimental to performance than including late 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 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.

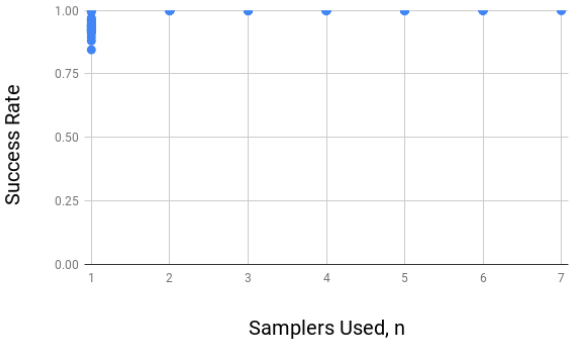
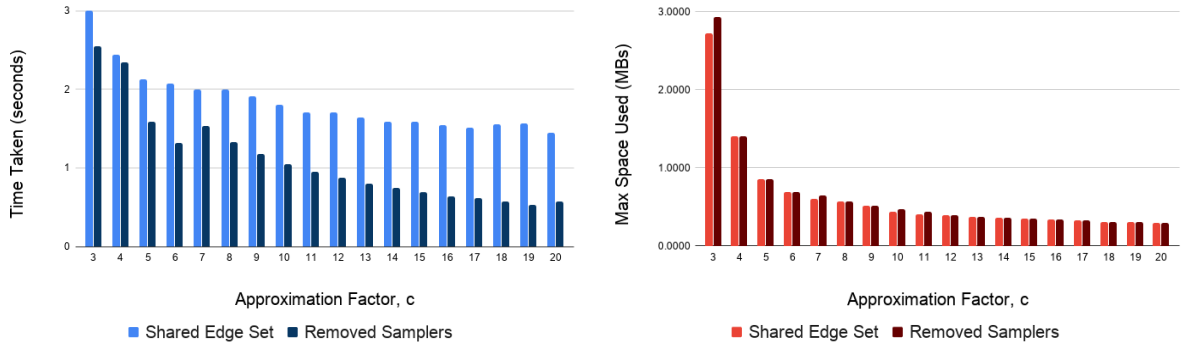


Figure 2.4.5: Success rate when only the first n samplers are run on `gplus`.

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 using the first n samplers for different values of c. `Figure 2.4.5` shows the results of this test and shows that the success rate dropped below 1 in the case that only the first sampler was used, in all other cases it was 1.

Running the first two samplers provided the best results. However, the largest graph I tested on only contains $100,000$ vertices so 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 $\texttt{min}\left(c, \texttt{max}\left(2, \left\lceil \frac{1}{5} \ln n \right\rceil\right)\right)$.



(a) Time Taken

(b) Space Used

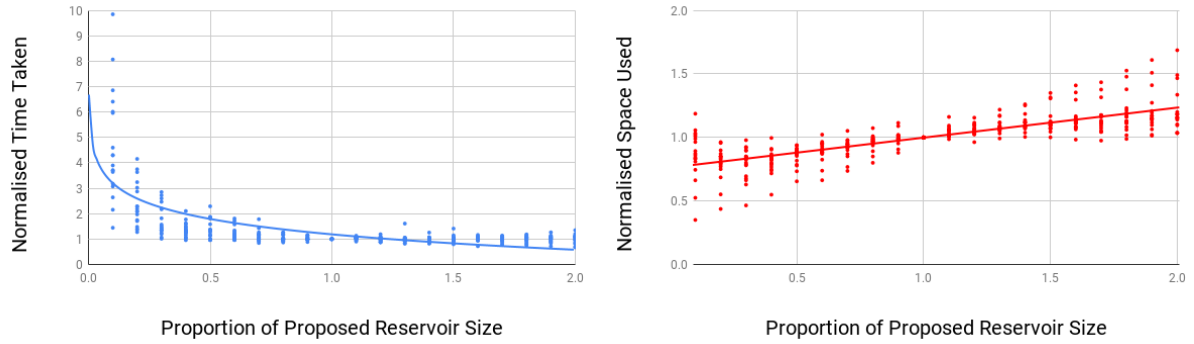
Figure 2.4.6: Results when using a reduced number of samplers and the technical improvements, tested on `gplus` graph for different approximation factors, against just using the technical improvements.

`Figure 2.4.6` shows the results when using the first $\texttt{min}\left(c, \texttt{max}\left(2, \left\lceil \frac{1}{5} \ln n \right\rceil\right)\right)$ samplers against just using the technical improvements discussed in `Section 2.3`. As seen in `Figure 2.4.4`, the space usage is unchanged while the time taken is down in all cases. Run time is reduced by over 50% for $c \geq 12$ and is below 1 second for larger values of c. A perfect success rate was maintained throughout these tests. Due to these good results, I implemented this change.

2.4.2 Reservoir Size

`Figure 2.4.4 (b)` showed that removing late 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 reservoir size.

Decreasing the reservoir 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 reservoir 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.



(a) Time Taken

(b) Space Used

Figure 2.4.7: Normalised results when varying the reservoir size used on `gplus` for different approximation values. Results normalised by the result when the proposed reservoir size was used for each approximation valuee. (Using changes from `Section 2.3` and a reduced number of samplers)

`Figure 2.4.7` shows the results when varying the reservoir size between 10% and 200% of the proposed size, for different values of c. The results are normalised to make them comparable across values of c. There is a positive linear relationship between space usage and reservoir size. The only inference that could be drawn from sample-size and time-taken is that there is a point at which time-taken dramatically increases, this is the point where the algorithm fails to find a sufficient neighbourhood. Overall, there is no clear strategy to improving the algorithm by varying the reservoir size.

Algorithm 3 gives **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.

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

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

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

Algorithm 4 is a naïve algorithm for solving `neighbourhood detection` for insertion-only graph streams. This approach records the neighbourhood of every encountered vertex and 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 and 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 all elements have degree $\frac{d}{c} - 1$ except for the last which has degree $\frac{d}{c}$, $\frac{d}{c} + (n-1)(\frac{d}{c} - 1) \in \mathcal{O}(n\frac{d}{c})$ 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

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`

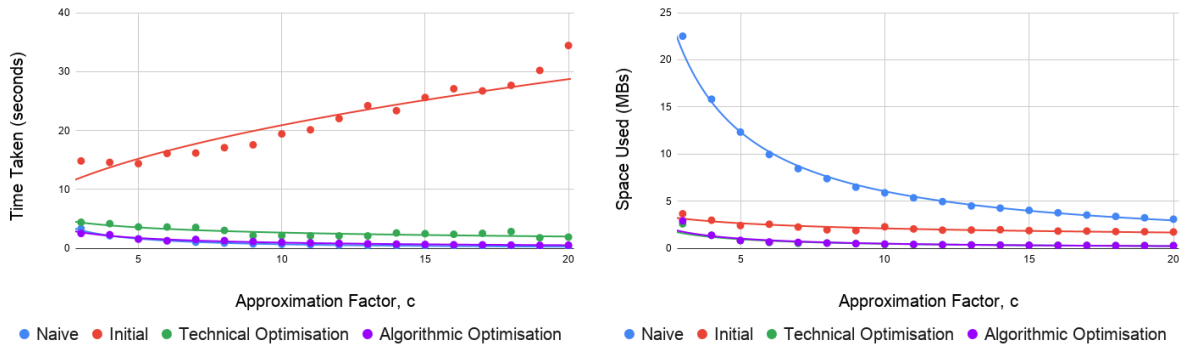
vertex, has $O(\frac{d}{c})$ space-complexity.

To evaluate my final implementation I compared its performance to that of different stages of the optimisation and to the naïve implementation given in **Algorithm 4**. 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.



(a) Time Taken

(b) Space Used

Figure 2.5.8: Time taken and space used by different implementations of the proposed algorithm for insertion-only streams when tested on `gplus`

`Figure 2.5.8` shows that the space usage was greatest for the naïve algorithm and lowest for the final implementation. The shortest run times came from the naive algorithm, followed by the final implementation. However, after three hours of computation the naïve algorithm had failed to return a result for the largest graph `gplus_large` due to its high space usage. For the largest graph tested, `gplus_large`, 13.7% of the stream file size was required to find a 2-approximation using the initial implementation and only 1.39% using the final implementation. This is a notable decrease in space requirements, meaning the algorithm can run efficiently on even large graphs.

The run time of the final implementation is useably quick: when run on `gplus_large` it took ∼150 seconds 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.

Chapter 3

Insertion-Deletion Streams

3.1 L_0 Sampling

L_0 `Sampling` is a technique for sampling, approximately uniformly, from the non-zero indices of a vector when given a stream of updates to the weights of the indices. Here I shall describe one space-efficient technique for L_0 `Sampling` which succeeds with probability $1 - \delta$ and requires $\mathcal{O}\left(\frac{1}{\delta}\log(1/\delta\gamma)\log^2(n)\right)$ space, this is significantly less than the $\mathcal{O}(n)$ space required to store the whole vector. This technique relies on two subroutines: `Perfect 1-Sparse Recovery`; and `Exact` s`-Sparse Recovery`.

3.1.1 Perfect 1-Sparse Recovery

`Perfect 1-Sparse Recovery` takes a stream of updates and if the described vector is exactly 1-sparse then it returns the index of the non-zero element, otherwise the routine fails. **Algorithm 5** provides an implementation `Perfect 1-Sparse Recovery` which works in cases where the weight of the non-negative element is non-negative. The non-negativity constraint is acceptable in this project as the graphs used as undirected and unweighted.

Algorithm 5: `Perfect 1-Sparse Recovery`

require: Update Stream $\{(i, \Delta)\}$

$\phi, \iota, \tau \leftarrow 0$
for all updates (i, Δ) **do**
    $\phi \leftarrow \phi + \Delta$.
    $\iota \leftarrow \iota + i \cdot \Delta$.
    $\tau \leftarrow \tau + i^2 \cdot \Delta$.
if $\iota^2 \equiv \phi \cdot \tau$ **then**
    `// Vector is 1-Sparse`
    return $\frac{\iota}{\phi}$ `// Index of non-zero entry`
else
    `// Vector is not 1-Sparse`
    return `FAIL`

Algorithm 5 uses three counters: sum of the weights $\phi = \sum \Delta$; sum of the weights weighted by their index $\iota = \sum i \cdot \Delta$; and, the sum of the weights weighted by the their index squared $\tau = \sum i^2 \cdot \Delta$. **Lemma 3.1** proves that $\iota^2 \equiv \phi \cdot \tau$ is a sufficient test for whether the described vector is exactly 1-sparse. If the vector is exactly 1-sparse then ι/ϕ is the index of the non-zero element & ϕ is its value. The only data stores are three counters each requiring $\mathcal{O}(\log n)$ space, thus `Perfect 1-Sparse Recovery` requires $\mathcal{O}(\log n)$ total space.

Lemma 3.1 $\iota^2 \equiv \phi \cdot \tau$ iff a vector is exactly 1-sparse.

Proof Consider a vector where i is the only non-zero index and that index stores value u. Then $\phi = u,\ \iota = i \cdot u$ and $\tau = i^2 \cdot u$. This means

$$\iota^2 = (iu)^2 = i^2u^2 = (i^2u) \cdot u = \phi \cdot \tau$$

Thus, if a vector is exactly 1-sparse then $\iota^2 \equiv \phi \cdot \tau$.
Consider an exactly 2-sparse vector with non-zero indices i, j each storing values u, v respectively. Then $\phi = u + v,\ \iota = i \cdot u + j \cdot v$ and $\tau = i^2 \cdot u + j^2 \cdot v$. This means

$$\begin{aligned} \iota^2 &= (iu + jv)^2 \\ &= i^2u^2 + 2ijuv + j^2v^2 \\ &\neq i^2u^2 + i^2ju + ij^2v + j^2v^2 \\ &= (u + v)(i^2u + j^2v) = \phi \cdot \tau \end{aligned}$$

Similar arguments show $\iota^2 \neq \phi \cdot \tau$ for greater levels of sparsity.
Thus, $\iota^2 \equiv \phi \cdot \tau$ iff a vector is exactly 1-sparse. □

3.1.2 Exact s-Sparse Recovery

`Exact s-Sparse Recovery` recovers a uniform sample of upto s of the non-zero indicies in a vector described by a stream with probability $1 - \gamma$. **Algorithm 6** defines a space efficient implementation of `Exact s-Sparse Recovery`.

Algorithm 6: `Exact s-Sparse Recovery`

require: Update Stream $\{(i, \Delta)\}$, Pairwise Independent Hash Functions $\mathcal{G}_2 : [n] \to [2s]$, Failure Rate γ

$r \leftarrow \log(s/\gamma)$ `// Number of rows`
$c \leftarrow 2s$ `// Number of columns`
$M \leftarrow r \times c$ matrix of `Perfect 1-Sparse Recoveries`
$\{g_1, \ldots, g_r\} \subseteq \mathcal{G}_2$ `// Hash function for each row`
for all updates (i, Δ) **do**
  for $y \in [1, r]$ **do**
    $v \leftarrow g_y(i)$
    Pass update (i, Δ) to $M[y][v]$
$A \leftarrow \{\}$ `// Set of recovered items`
for $x \in [1, c]$ **do**
  for $y \in [1, r]$ **do**
    $i \leftarrow$ recovered index of $M[x][y]$
    if $i \neq$ `FAIL` **then** $A \leftarrow A \cup \{i\}$
return A

Algorithm 6 uses a two-dimensional matrix M of `Perfect 1-Sparse Recoveries` and assigns a hash function g_r to each row. When an update (i, Δ) is received, it is passed to the $g_r(i)^{\text{th}}$ `1-Sparse Recovery` of each row. Once the updates have been exhausted the returned elements of the 1-sparse recoveries are collected into a set A and returned.

g_r maps, on average, $\frac{n}{c}$ indices to each 1-sparse recovery. These `1-Sparse Recoveries` fail iff more than one of these assigned indices is non-zero. It is proved in [3] that setting the number

of rows and columns to $\log(s/\gamma)$ and $2s$ respectively means **Algorithm 6** returns at most s indices with probability $1-\gamma$.

Algorithm 6 uses $2s \cdot \log(s/\gamma)$ `1-Sparse Recoveries` thus the total space requirement for this implementation of `Exact` s`-Sparse Recovery` is $\mathcal{O}(s\log(s/\gamma)\log(n))$.

3.1.3 L_0 Sampling

Algorithm 7: L_0 `Sampler`

require: Update Stream $\{(i,\Delta)\}$, k-wise independent hash function $h:[n]\to[n^3]$, L_0 Failure Rate δ, s-Sparse Failure Rate γ

$j \leftarrow \log_2(n)$ `// Number of Exact` s`-Sparse Recoveries`
$s \leftarrow \frac{1}{\delta}$ `// Sparsity of Exact` s`-Sparse Recoveries`
Initialise $S_1,\dots,S_j$ as `Exact` s`-Sparse Recoveries`
$r \leftarrow 0$ `// Estimate of sparsity`
for all updates (i,Δ) **do**
  $r \leftarrow r+\Delta$
  for $k \in [1,j]$ **do**
    $v \leftarrow h(y)$
    if $v \le \frac{n^3}{2^k}$ **then** Pass update (i,Δ_i) to S_y
$x \leftarrow \log_2(r)$ `//` s`-Sparse recovery to recovery & sample from`
$a \leftarrow$returned from S_x `// Set of returned non-zero indices`
if $|a| \in [1,s]$ **then**
  return Index in a with lowest hash value
return FAIL

The process for L_0 `Sampling` given in **Algorithm 7** is similar to the process for `Exact` s`-Sparse Recovery` in **Algorithm 6**. j `Exact` s`-Sparse Recoveries` are instantiated and a k-wise independent hash function $h(\cdot)$ (with $k \in \mathcal{O}(s)$) is used to determine which recovery to pass an update to. When an update (i,Δ) is received we update an estimate of the sparsity r of the vector being sampled and pass the update to all S_k where $h(i) \le \frac{n^3}{2^k}$. Once the update stream is consumed we consider the set returned from the $\log(r)^{\text{th}}$ recovery and return the index which produces the lowest hash-value. Since the hash-values are assigned uniformly at random, this is equivalent to uniformly sampling from the returned set.

Algorithm 7 uses $\log(n)$ `Exact` s`-Sparse Recoveries` thus the total space requirement for this implementation of L_0 sampling is $\mathcal{O}(s\log(s/\gamma)\log^2(n))$. During implementation it is shown that $\gamma,\delta \approx 0.1$ produce good results, in this case the algorithm requires $O(\log^2 n)$ space for reasonably n.

3.2 Proposed Algorithms

In [1] two single-pass algorithms are proposed for `neighbourhood detection` in an insertion-deletion stream, both of which use L_0 sampling to acquire edges from the described graph.

3.2.1 Vertex Sampling

Algorithm 8 takes a uniform sample from the vertex set and then runs multiple L_0 samplers on the edge vector of each sampled vertex. This is equivalent to sampling uniformly from the neighbourhood of each vertex multiple times. From these edges a neighbourhood can be found and if it is of sufficient degree then it may be returned by the algorithm. The vertex sample is

Algorithm 8: One-pass c-approximation Insertion-Deletion Streaming Algorithm for `Neighbourhood Detection`. (Vertex Sampling)

require: Degree bound d, Approximation factor c, Insertion-Deletion Stream S, List of Vertices A

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.

taken uniformly as we have no prior information about the degree of the vertices in the graph and thus assume each is equally likely to be of sufficient degree.

Assuming $\mathcal{O}(\log n)$ space is required to store a vertex, the vertex sample requires $\mathcal{O}(x \log^2(n))$ and a total of $100\frac{xd}{c}\ln^2(n)$ L_0 samplers are run each requiring $\mathcal{O}(\log^2(n))$ space. This means the total space requirement for **Algorithm 8** are $\mathcal{O}(\frac{xd}{c}\log^4(n))$.

As incident edges are sampled uniformly, using $10\frac{d}{c}\ln n \gg \frac{d}{c}$ samplers results in a high probability that a neighbourhood size $\frac{d}{c}$ is found for vertices of degree greater $\frac{d}{c}$. Thus the success of **Algorithm 8** relies on it sampling a a vertex of sufficient degree. This means **Algorithm 8** is not effective when there are very few vertices of sufficient degree. This scenario occurs for small values of c and for graphs with degree distributions similar to a star graph. In these cases sampling the edges directly has a greater success rate.

3.2.2 Edge Sampling

Algorithm 9: One-pass c-approximation Insertion-Deletion Streaming Algorithm for `Neighbourhood Detection`. (Edge Sampling)

require: Degree bound d, Approximation factor c, Insertion-Deletion Stream S

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.

Algorithm 9 runs L_0 samplers directly on the set of edges in order to acquire a uniform sample of the edges in the graph. This sample is then checked for any neighbourhoods of degree $\frac{d}{c}$. The sample of edges can be considered as an insertion-only graph stream, thus **Algorithm 2** can be used to find sufficient neighbourhoods or in cases where the number of sampled edges is very low the naïve **Algorithm 4** could be more appropriate due to its speed.

The edge sampling approach is most likely to succeed when there is a small set of vertices which the majority of edges are incident to. The total space complexity of **Algorithm 9** is $\mathcal{O}\left(\frac{nd}{c}\left(\frac{1}{x} + \frac{1}{c}\right)\log^3(n)\right)$ which is less than the total space complexity of **Algorithm 8**.

3.3 Implementation

During these implementations I made the assumption that vertices were labelled with unique values in $[1, n]$. This simplifies many parts of the implementation as each vertex aligns to an index in the edge vector. This assumption is reasonable as it places no restrictions on the use of these implementations as a `map` can be used to assign index locations to each vertex.

3.3.1 Perfect 1-Sparse Recovery

Lemma 3.2 For an unweighted insertion-deletion stream $\phi \equiv 1$ and ι is the non-zero index iff a vector is exactly 1-sparse.

Proof In an unweighted insertion-deletion streams $\Delta = -1$ for a deletion instruction and $\Delta = 1$ for an insertion instruction. In the case where a vector is exactly 0-sparse (*i.e.* is the zero vector) the result trivially fails since there is no non-zero index for ι to hold. Consider a vector v with non-zero index i. If v is exactly 1-sparse then i must have been insert exactly one more time than it was deleted, and all other indexes must have had the same number of insertions and deletions. Since each deletion cancels out an insertion there is only one insertion left. This means $\phi = 1$ and $\iota = 1 = i$, the result holds.
Suppose v is exactly s-sparse with $s > 1$. Then there exists $s - 1$ other indices $j_1, \ldots, j_{s-1} \neq i$ which are inserted exactly one more time than it is deleted. After letting the deletions cancel the insertions there are s insertion instructions left, this means $\phi = s$ and the result does not hold. Thus the result holds iff v is exactly 1-sparse. □

The insertion-deletion streams used in this project are unweighted. Thus, by **Lemma 3.2**, the exact 1-sparse recovery process describded in **Algorithm 5** can be simplified to the implementation described in **Algorithm 10**. This implementation removes the τ counter meaning the total space used by the algorithm is reduced by a third which is significant as lots of these processes are run during **Algorithm 8** and **Algorithm 9**. Implementing **Algorithm 10** is trivial.

Algorithm 10: `Case Specific Perfect 1-Sparse Recovery`

require: Update Stream $\{(i, \Delta)\}$
$\phi, \iota \leftarrow 0$
for all updates (i, Δ) **do**
  $\phi \leftarrow \phi + \Delta$.
  $\iota \leftarrow \iota + i \cdot \Delta$.
if $\phi \equiv 1$ **then return** ι
else return `FAIL`

3.3.2 Exact s-Sparse Recovery

When implementing `Exact` s-`Sparse recovery` the main decision is how to implement the family of pairwise hash functions efficiently. I used the set of functions

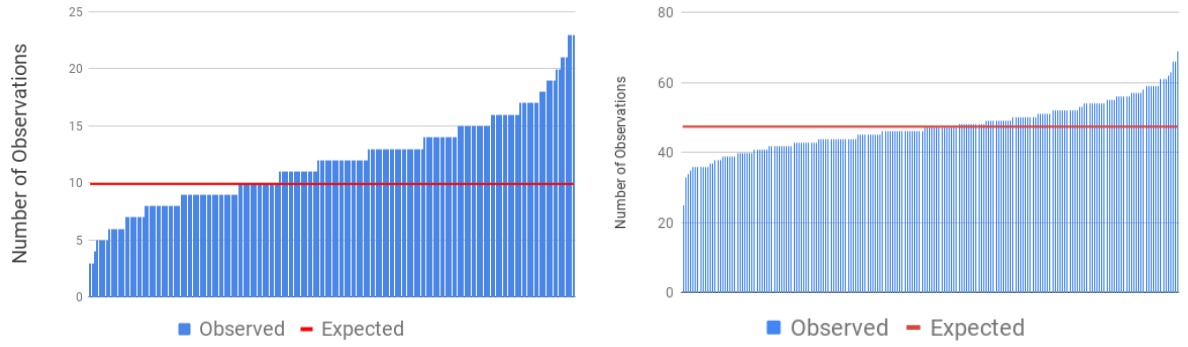
$$\mathcal{H}_2 := \{h_{a,b}(x) = ((ax + b) \bmod p) \bmod 2s | a, b \in [1, P - 1]\}$$

where P is a prime greater than $2s$. These hash functions map to the desired range $[0, 2s)$. This family of hash functions are ideal for this project as can be stored in $O(\log P)$ bits as only the values of a, b need to stored for each function. Similarly it is easy to generate a function in this family by sampling twice from $[1, P - 1]$.

Finding primes is hard so I hard coded $P = 1,073,741,789 > 2^{29}$. This is an upper bound on number of columns the s-`Sparse Recovery` algorithm can handle, which is equivalent to placing a lower bound on the acceptable failure rate δ of the L_0 `Sampling` algorithm. $2 \cdot \frac{1}{\delta} < P \implies \delta > \frac{2}{P} \approx 2 \times 10^{-9}$. It is shown during evaluation that this bound is inconsequential to the success and performance of L_0 `Sampling`.

3.3.3 L_0 Sampling

For this project L_0 sampling is used on two different vectors: the edge vector of a specified vertex; and the edge set of the whole graph. The edge set of a graph is often defined as an $n \times n$ adjacency matrix, to apply L_0 sampling to this matrix we flatten it into a n^2 dimensional vector by reading the rows sequentially. The graphs used in this project are unweighted so the values of the edge vector are booleans defining whether an edge exists or not. This is implement by setting $\Delta = 1$ for insertion instructions and $\Delta = -1$ for deletion instructions. This means the values in the vectors are 0 or 1 only.



(a) Pairwise Hash Function

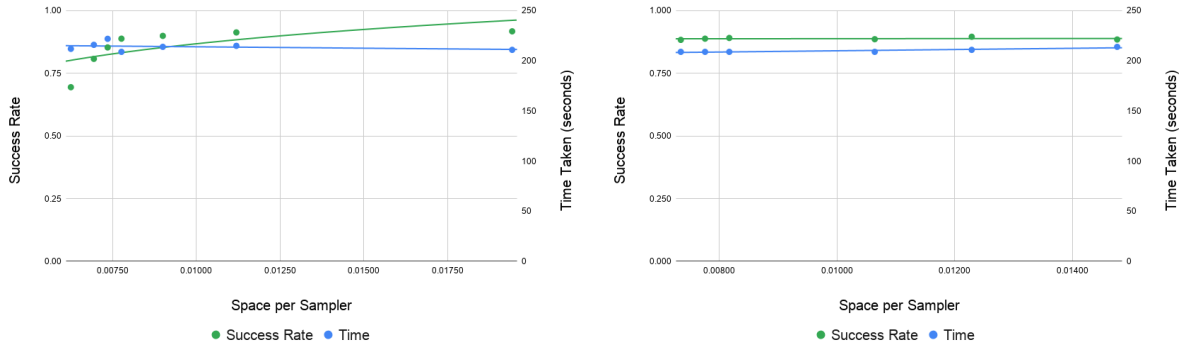
(b) Unique Hash Map

Figure 3.3.1: Occurrences of each value in a sample when different hashing methods are used for L_0 sampling, order by increasing number of occurrences.

When implementing L_0 `Sampling` the main problem is how to implement the k-wise independent hash function efficiently. I looked into libraries which provided such functions but none allowed for the range to be controlled and taking the modulus with respect to n^3 breaks the independence property. Using a pairwise independent hash function would be ideal as they are easy to implement. `Figure 3.3.1 a)` shows the results when testing using a pairwise hash function. These results fail a χ^2 test for a uniform distribution, indicating a pairwise independent hash function is not sufficient.

A `map` which assigns each value in $[n]$ to a distinct value in $[n^3]$, with values chosen uniformly at random, is equivalent to an n-wise independent hash function. The probability of a collision when assigning each value is less than $\frac{1}{n^2}$ meaning no collisions are expected when assigning values. This means that such a `map` is generated in $\mathcal{O}(n)$ time on average and requires $\mathcal{O}(n \log n)$ space. `Figure 3.3.1 b)` shows the results when such a `map` is implemented as the hash function h. These results pass a χ^2 test for a uniform distribution so I implemented this `map` as the k-wise independent hash function. As the vertices are assumed to be labelled in $[n]$ this `map` can be implemented as an array with the hash values stored in each index. Using an array is more space-efficient than a tradition `map`.

The maximum hash value is a constraint on the size of the domain and thus the number of vertices which can be handled. In theory this is not a problem, but in practice the limit of this value depends on the data type used. I implemented the hash array as an array of `uint64_t` (8 bytes) which has the greatest maximum value of the standard c++ numeric data type. The maximum domain is the cube root of this value, thus **Algorithm 8** will not work for graphs with more than $(2^{64} - 1)^{1/3} \approx 2.6 \times 10^6$ vertices and **Algorithm 9** will not work for graphs with more than $(2^{64} - 1)^{1/6} \approx 1,625$ vertices. This constraints can be reduced by reducing the relative range of the hash function.



(a) Varying δ ($\gamma = .1$)

(b) Varying γ ($\delta = .2$)

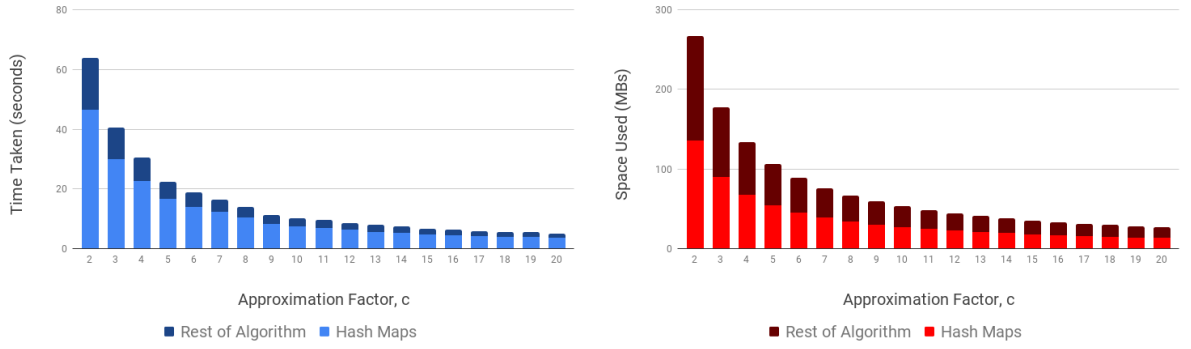
Figure 3.3.2: Space Usage and time taken when varying γ and δ in L_0 Sampling implementation. For graph `facebook_deletion`

When setting γ and δ there is a trade off between space-usage and the success rate. The value of δ affects the number of columns and rows in each s-sparse recovery, whereas γ only affects the number of rows. Thus optimising δ first has the greatest effect. `Figure 3.3.2 a)` plots success-rate and time taken against space used when $\gamma = .1$ and δ is varied. The time taken is constant throughout, but the success rate plateaus at $\sim 90\%$ when $d \geq .2$. `Figure 3.3.2 b)` shows that fixing $\delta = .2$ and varying γ has no effect on performance. Thus γ is set to .3 in order to minimise space usage.

3.3.4 Vertex Sampling Algorithm

The vertex sample A can be acquired by uniformly sampling from $[n]$ until a sufficient number of unique values is sampled. For a sample of size m, each time a vertex is sampled the probability of a collision is at most $\frac{m}{n}$ meaning the expected number of total collisions is at most m. Thus, in the worst case a sample is generated in $\mathcal{O}(mn)$ time.

The sampled edges form an insertion-only graph stream so a sufficient neighbourhood can be found among them using the algorithms discussed in **Chapter 2**. Due to the limitations from the L_0 sampler implementation I could not run this implementation on particular large graphs, this meant that the number of sampled edges is very small and thus the naive **Algorithm 4** provided the best results as it always succeeds. If the implementation was improved to run on larger graphs then **Algorithm 3** should be used for its overall performance improvements.



(a) Time Taken

(b) Space Used

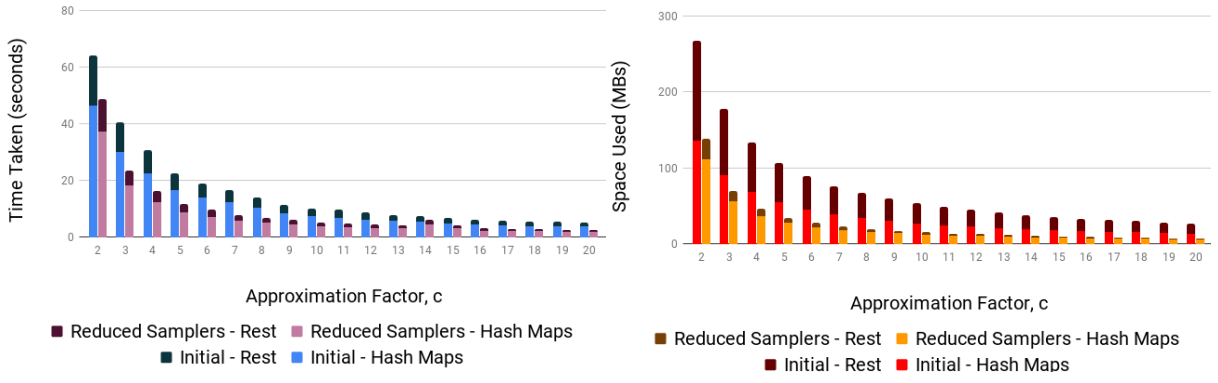
Figure 3.3.3: Space Usage and time taken for initial implementation of vertex sampling, for graph `facebook_deletion`, when varying approximation factor.

`Figure 3.3.3` shows the results of an initial implementation of vertex sampling. The proposed sample size would have been greater than n for `facebook_deletion` so $\sqrt{n}$ was used as a placeholder value until a more suitable sample size was tested for. This sample size proved too small for $c \leq 4$ and resulted in some failures (Success rate remained above 83% in all cases). These failures do not have a significant effect the performance of the algorithm as checking for sufficient neighbourhoods accounts for a small amount of the time and space usage. As expected, the hashing functions used in L_0 sampling account for the majority of space and time usage. Reducing the number of samplers will reduce the time and space usage of the implementation but will increase the failure rate if the number of unique edges sampled is reduced. The suggested number of samplers allows for a lot of redundancy as many edges are sampled multiple times.

Lemma 3.3 $\frac{\ln(1-p)}{\ln\left(1-\frac{1}{m}\right)}$ L_0 samplers are expected to find $p\delta$ unique neighbours of a vertex of degree δ.

Proof Consider a vertex v of degree δ and let i be a neighbour of v. Each L_0 sampler run on the edge vector of v does not sample i with probability $\frac{\delta-1}{\delta}$. As each sampler is independent the probability of i not being sampled after t L_0 samplers have been run is $\left(\frac{\delta-1}{\delta}\right)^t$. For the probability that i has been sampled to be at least p requires $\left(\frac{\delta-1}{\delta}\right)^t \leq 1-p$. This means $t \geq \frac{\ln(1-p)}{\ln\left(1-\frac{1}{m}\right)}$ samplers need to have been run. Since each neighbour is independent, p is also the expected proportion of the neighbours which are sampled.

Lemma 3.3 tells us the minimum number of L_0 samplers required for us to expect a sufficient neighbourhood to a vertex to be found and by tuning the value of p the number of samplers and achieved success rate can be optimised. When a vertex is sampled it is appropriate to assume it has at least degree $\frac{d}{c}$. The greater the actual degree of the vertex, the lower p needs to be for $\frac{d}{c}$ neighbours to be returned as collisions are less likely. So tuning p for vertices of degree close to $\frac{d}{c}$ is a good indicator of how vertices of greater degree will perform. Testing showed that setting $p = .9$ resulted in a sufficient neighbourhood being found 23% of the time for a vertex of degree $\frac{11d}{10c}$ and 97% of the time for a vertex of degree $\frac{12d}{10c}$. The implementation of L_0 samplers being used only successfully samples 90% of the time, to account for this the expected number of samplers is multiplied by $\frac{1}{.9}$.



(a) Time Taken

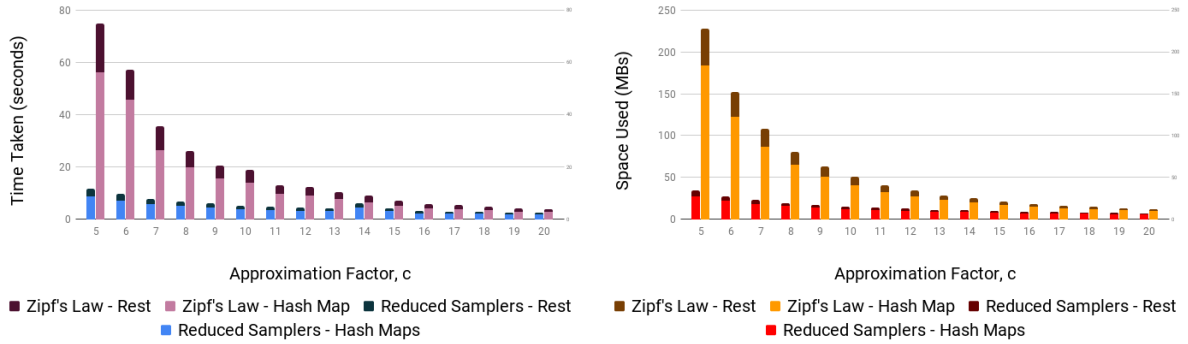
(b) Space Used

Figure 3.3.4: Time taken and space used when the number of L_0 samplers is optimised against the initial implementation, for graph `facebook_deletion`, when varying approximation factor.

`Figure 3.3.4` shows the results when $\frac{1}{.9} \cdot \frac{\ln(1-.9)}{\ln(1-\frac{c}{d})}$ samplers are run for each sampled vertex,

rather than the suggested $10\frac{d}{c}\ln(n)$. The total space usage more than halved in all cases, and the space not used by the hash maps reduced by over 80%. The time taken by the algorithm (excluding that spent producing the hash maps) decreased by over 40% in all cases. Both of these reductions are due to the total number of L_0 samplers being reduced. The success rate was unchanged, showing that there was a lot of redundancy in the number of samplers being run.

The optimal vertex sample size depends on the degree distribution of the graph as this describes how many vertices are of sufficient degree. Thus any prior knowledge of the degree distribution of the graph should be used. It is trivial to establish the degree distribution of a graph from a graph stream by counting the degree of each vertex although an exact distribution is not required as research has been done into typically degree distributions for common graphs from real world applications. In this project I have been using graphs from social networks which are theorised [4] to follow a Power-law distribution $f(x) = cx^{-\alpha}$. By Zipf's Law of Power-law distributions, the expected number of vertices of degree at least $\frac{d}{c}$ is c. Thus a uniform sample of $\frac{n}{c}$ vertices in V is expected to contain a vertex of sufficient degree. A larger sample should be taken as it is not guaranteed that the maximum neighbourhood will be found for all sufficient vertices. Using this as a basis for the vertex sample size should make the algorithm very successful but as $\frac{n}{c} \gg \sqrt{n}$ for the most interesting values of c its performance will be significantly worse.



(a) Time Taken

(b) Space Used

Figure 3.3.5: Space Usage and time taken when $\frac{12n}{10c}$ vertices are sampled against implementation of reduced number of sampelrs, for graph `facebook_deletion`, when varying approximation factor.

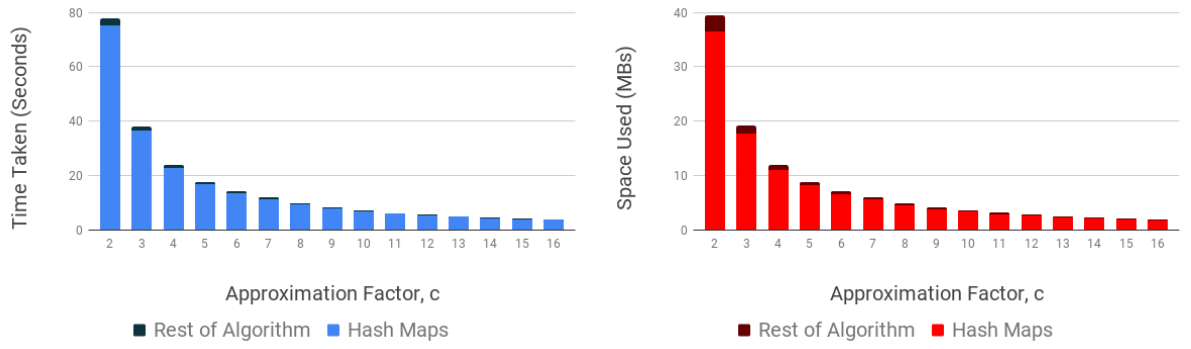
`Figure 3.3.5` shows the performance results when a vertex sample size of $\frac{12n}{10c}$ was taken compared to the initial $\sqrt{n}$. As predicted, the algorithm succeed in all cases where the new sample size was used however it's performance is significantly worse in all cases where $\frac{12n}{10c} \geq n$. The small size `facebook_deletion` means that Zipf's Law is unlikely to fit well and it is theorised that the degree distribution of social networks have a fatter tail than a standard Power-law distribution. It is clear that further research is required into a performance optimal sample size for different scenarios.

3.3.5 Edge Sampling Algorithm

Running an L_0 sampler on the set of edges means the domain of the k-wise independent hash function is the edge set and thus we require a way of indexing the edges such that each has a unique value which can then be hashed. This indexing needs to be bijective so that the recovered index can be converted back into an edge. My initial implementation assigned the edge (u, v) to $i = (\texttt{min}(u, v) - 1) \cdot n + \texttt{max}(u, v) - 1$ which maps edges to values in $[0, n^2)$. The use of `max`

and min means that (u, v) and (v, u) map to the same value which is desirable since the edges are undirected. An edge can be recovered from this mapping as $\texttt{min}(u, v) = (i \bmod n) + 1$ and $\texttt{max}(u, v) = \texttt{floor}(1/n) + 1$. This mapping means the hash maps used in L_0 sampling now have $\mathcal{O}(n^2)$ space and time complexity.

As with the vertex sampling approach finding a sufficient neighbourhood in the sampled vertices should be done using the algorithms discussed in **Chapter 2**.

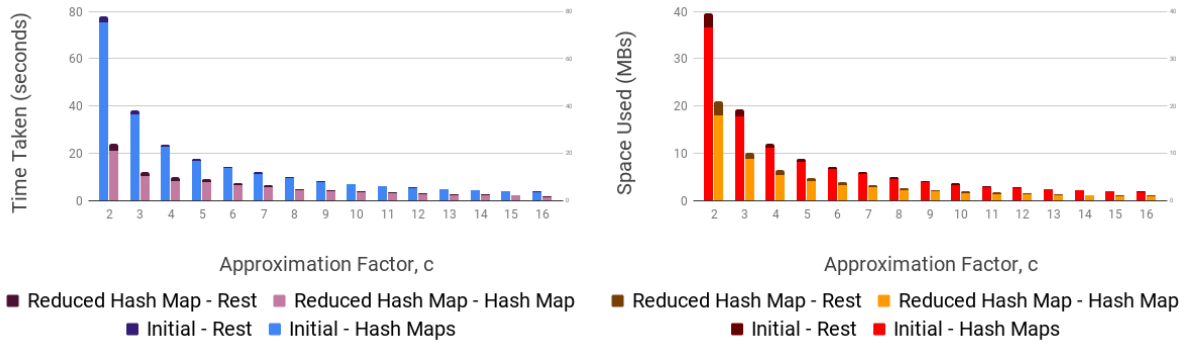


(a) Time Taken

(b) Space Used

Figure 3.3.6: Space Usage and time taken for initial implementation of edge sampling, for graph `facebook_small_deletion`, when varying approximation factor.

`Figure 3.3.6` shows the results from the initial implementation of **Algorithm 9**. As with the implementation of the vertex sampling algorithm, the majority of the space and time requirements are spent in generating the hash arrays for the L_0 samplers. The requirement for generating these is so high that I could not run the algorithm on `facebook_deletion` as the space requirements were over 1 GB and it took several minutes to generate a single hash array.



(a) Time Taken

(b) Space Used

Figure 3.3.7: Space Usage and time taken when a reduced hash map is implemented against the initial implementation of edge sampling, for graph `facebook_small_deletion`, when varying approximation factor.

To reduce the size of the hash maps I changed how the edges were indexed. The graphs used in this project are undirected so have at most $\frac{1}{2}n(n-1)$ unique edges. These can be visualised as the upper triangle of an adjacency matrix. The following formula map between an edge (u, v) to key $k \in [\frac{1}{2}n(n-1)]$. In the implementation I ensured that $v > u$ so that the edges (u, v) and

(v, u) mapped to the same k and that the equations would succeed.

$$\begin{aligned} k &= \tfrac{1}{2}n(n-1) - \tfrac{1}{2}(n-v)(n-v-1) + u + v - 1 \\ v &= n - 2 - \text{floor}\left(\tfrac{1}{2}\sqrt{-8k + 4n(n-1) - 7} - \tfrac{1}{2}\right) \\ u &= k + v + 1 - \tfrac{1}{2}n(n-1) + \tfrac{1}{2}(n-v)(n-v-1) \end{aligned}$$

Implementing this method of indexing edges optimises the hash arrays but they still impact performance too much for the implementation to be usable on anything except the smallest of graphs.`Figure 3.3.7` shows the performance improvements from this implementation.

This implementation succeeded in every case due to large number of samplers successfully sampling every edge, this is inefficient as most of these edges went unused while checking for a sufficient neighbourhood. For larger graphs not all edges would be sampled as the relative difference between the number of edges and number of samplers decreases. However, this implementation is not usable on reasonably sized graphs so I cannot effectively test alterations to the number of samplers run. The edge sampling method should be designed to work in cases where the vertex sampling method fails. Primarily this is when it is unlikely that a sufficient vertex is sampled, this occurs in graphs where only a handful of vertices are of high degree and the rest are of very low degree. In these cases there are very few edges which do not involve a vertex of high degree and by estimating how many high degree vertices there are you can calculate the expected number of edges which need to be sampled before a sufficient neighbourhood is found.

3.4 Evaluation

The performance of the implementations of **Algorithm 8** and **Algorithm 9** are dominated by the hash functions used by the L_0 samplers. This meant that the implementations could not be run on graphs of sufficient size to evaluate certain aspects of the implementations, namely the vertex sample size in **Algorithm 8** and number of samples in **Algorithm 9**. As well, it would be inappropriate to compare the performance to that of the naïve algorithm described in **Algorithm 11** as the proposed algorithms account for very little of the time and space of the whole implementation.

Algorithm 11: 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)
return FAIL `No sufficient neighbourhoods`

Chapter 4

Conclusions

4.1 Achievements

The algorithm proposed in [1] for insertion-only graphs proves very practical for solving `neighbourhood detection` on real world graphs. The changes to the implementations made reduced the time and space used by the algorithm significantly, while not compromising on the success rate. The final evaluation shows that the proposed algorithm is much more space-efficient than a naïve approach. Even for close approximations of large streams the 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, without requiring virtual memory. The run-times of my final implementation are very usable with solutions being found for a graph of over 30 million edges in a couple of minutes for close approximations $c \in [2, 4]$.

Due to difficulties in generating functions from a k-wise independent hashing family the proposed algorithm for insertion-deletion graphs could not be evaluated on graphs which were sufficiently large as to be interesting. However, I still managed to discuss several improvements to the implementation. The simplification of `1-sparse recovery` to only require two counters is a great improvement as, when the hash function is discounted, the `1-sparse recoveries` account for the majority of the space used by the L_0 samplers and thus both proposed algorithms. The probabilistic approach taken to reduce the number of L_0 samplers run for each vertex significantly improved performance by reducing redundancy, without compromising success. The indexing method which mapped edges to unique values in $[0, \frac{1}{2}n(n-1))$ was a really good improvement as it is the optimal indexing method for undirected graphs. Although, the introduction of a mathematical hashing function rather than a look up table would negate this improvement.

4.2 Future Work

To conclude this project I will mention a few areas that I think any further evaluation of the algorithms proposed in [1] should consider.

Complete Real World Evaluation

In this project I soley used graphs from social networks as they were readily available and an area where `neighbourhood detection` is an interesting problem. It would be interesting to test the implementations produced in this project on data sets from other fields. Section `I.ii Motivating Applications` provides a few examples. Further, it would be interesting to investigate how the returned neighbourhood is used to draw inferences about the graph as this would provide greater direction to how the implementations should be optimised. A possible problem could be detecting DDOS attacks from a network log.

k-Wise Independent Hashing Family

An alternative family of k-wise independent hash function is $h(x) = \left(\left(\sum_{i=0}^{k-1} a_i x^i\right) \bmod P\right) \bmod m$ where $a_0 > 0$ and $a_i < P \; \forall \; i$. A function from this family is quick to generate as only the k a_i values need to be random generated and can be stored as $k+2$ integers (the values of a_i, k and m). Both of these are dramatic improvements from the implementation of hash arrays used in the implementation of an L_0 sampler. The difficulty with implementation functions from this family is that the values of x^i quickly overflow causing inaccurate calculations. I am sure an implementation of this family of hash functions could be worked out, but I ran out of time to do so in this project.

Vertex Sample Size

I discussed how the size of the vertex sample used in **Algorithm 8** should depend on the degree distribution of the graph, and investigated one line of thought for how this could be done for graphs of social network connections. Investigating this further was outside the scope of this project but would be an interesting feature to research further.

The edge sampling approach proposed in **Algorithm 9** should be considered whilst evaluating the size of the vertex sample, as there will be cases where this approach is more efficient. The most obvious example of this would be a star graph where the vertex sampling approach would need to sample every vertex in order to guarantee find a sufficient neighbourhood, whereas the edge sampling approach only needs to sample $\frac{d}{c}$ edges. In more general scenarios it is likely that the edge sampling approach is better for close approximations and the vertex sampling approach better for loser approximations.

Reservoir Sample Size

I tested changing the reservoir sample size used in **Algorithm 2** but no strategies where found which consistently improved performance. These tests changed the sample size of every sampler used, but it may be that reducing the sample size for later samplers produces good results as those samplers are sampling for a smaller pool. This would be the main feature to look at when further trying to optimised the implementation from **Chapter 2**.

Bibliography

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

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

[3] Graham Cormode, Donatella Firmani *A Unifying Frameworkd for ℓ_0-Sampling Algorithms.* September 2014

[4] Tushti Rastogi *A power law approach to estimating fake social network accounts.* May 2016