Dissertation Type: research

DEPARTMENT OF COMPUTER SCIENCE

The set-cover problem in the semi-streaming model

Charana Nandasena

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Friday 10th May, 2019

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of MEng in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

charana nandasena

Charana Nandasena, Friday 10th May, 2019

Contents

Executive Summary

My research hypothesis is that - the algorithms of semi-streaming and sublinear-space streaming computational models produce sufficiently accurate approximations to modern, heavily investigated greedy-based techniques, additionally rivalling the performance and scalability of existing parallelizable and I/O-efficient greedy algorithms. Bridging the gap between theory and practice as well as justifying further research into non-greedy based techniques for the minimum set-cover and more generally, graph problems.

In summary I

- Discovered and understood the breath of literature concerning the minimum set-cover. In specific -
  - The class of offline algorithms, to establish a performance and accuracy baseline.
  - The class of semi-streaming and sublinear-space streaming algorithms to
    - Grasp fundamental algorithmic and implementation techniques that perform well in these computational models, adopting or modifying them perform better in practice.
    - Efficiently implement and assess these algorithms w.r.t. approximation and scalability under realistic small, large and massive problem instances in relation to the decided baseline.
- Find
  - that even I/O-efficient implementations of offline greedy-based techniques see super-linear slowdown with increasing problem size. This effect is more pronounced when sufficient RAM is unavailable or algorithms communicate directly with disk. This is in deep contrast to semi-streaming algorithms that consistently show a linear scalability.
  - that most accurate and scalable sublinear-space techniques involve randomization, which while theoretically sounds make stretched assumptions that are difficult to satisfy in-practice.
- Propose
  - an original semi-streaming algorithm that computes a solution of comparable quality to the baseline while being more computationally efficient than modern techniques, additionally being easily parallelizable to threaded or distributed computing environments.
  - a template design for deterministic, sublinear-space streaming set-cover algorithms, that compute an almost ideal solution in dramatically less memory. Importantly it doesn't fall victim to the practical in-feasibility of randomization techniques.

Supporting Technologies

- The minimum set-covering algorithms were written in **C++** and its standard library, additionally making use superior data-structures and algorithms of **Boost** header-only libraries [1].

Acknowledgements

Many thanks to my supervisor, Dr. Christian Konrad for his eager supervision and always being two steps ahead

Chapter 1

Contextual Background

1.1 Big Data

By a consequence of economies of scale, the quantities of devices generating information have become large. From industrial supercomputers and clusters to consumer mobiles, cameras, and IoT devices. The the volume, variety and velocity of data is unprecedented, being a haven of untapped potential.

The dawn of the data age can have far-reaching implications across all sectors of industry. Leaders in industry, science and government alike regularly meet difficulties in taming large data-sets. Why? Processing these data-sets are encumbered by the runtime and memory complexity of existing efficient solutions. A runtime complexity of $O(n^2)$, explodes as n becomes extremely large thereby becoming intractable. An implementation of memory complexity linear in the size of the problem instance (i.e. $O(|P|)$) quickly runs out of memory becoming I/O bound, at worst being immediately killed by the kernel alternatively observing a significant slowdown.

1.2 Semi-Streaming Model

The data-stream model [9], was introduced as the ideal setting for big-data algorithms. Allowing sequential access to the problem input and a restrictive polylogarithmic space bound, $O(log^{O(1)}n)$. Sequential access allows for a predictive access pattern that exploits the temporal and spatial locality of the memory hierarchy subsystem of modern systems. This together with a space bound sublinear in problem size assist averting I/O bound implementations. I/O bound is a programmatic state where the execution time of the process is principally determined by how long it takes reads and writes to/from memory or disk to complete. Algorithmic designs in this model tend to be simpler suggesting little overhead and a low runtime complexity also being more easily parallelised to threaded or distributed computing environments. From a functional perspective streaming offers more. For instance, the processing of online, real time data piped from a data stream source such as Kafka or even a steadily growing log file becomes plausible if we allow processing over the input as a stream of items.

Sadly, very few graph problems [14] have been considered in the streaming model. Many graphs problems are provably intractable if the available space is sublinear in the number of graph vertices [34], These space lower bounds are derived from results in communication complexity [39]. whereas many problems become feasible once there is memory roughly proportional to the number of nodes in the graph.

The semi-streaming model [28], an alternate streaming model eases these restrictions, allowing multiple passes over the data stream, ideally this is a low constant at most being logarithmic in the number of vertices n, $O(\log_2 n)$ and a larger but still sublinear space bound $O(n.\log_2^{O(1)} n)$.

1.3 Minimum Set Cover (MSC)

Given an input pair $(\mathcal{U}, \mathcal{S})$ where $\mathcal{U}$ is a set of n elements called the universe and $\mathcal{S}$ is a collection of m sets commonly referred to as the set system whose union equals the universe. The set-cover optimization problem or minimum set-cover (MSC) is finding the smallest subset of $\mathcal{S}$ that covers (i.e. whose union equals) the entire universe. As previously mentioned the MSC problem is NP-complete and can only be approximated in polynomial time. We now detail those approximation algorithms reported in existing literature.

A problem instance can be described by the following properties -

- *n* is the size of the universe $\mathcal{U}$.
- *m* is the size of the set system $\mathcal{S}$.
- *M* is the total number of items over all sets i.e. $M = \Sigma_{i=0}^{m}|S_i|$.
- *avg* $|S_i|$ is the average set size.
- Δ is the maximum set size.

The decisional set cover problem is a classical question in complexity theory, proven in 1972 to be NP-complete as part of Karp's 21 NP-complete problems [38] and its optimization variant, the minimum set-cover (MSC) is NP-complete. In complexity theory, problems of the NP-complete class cannot be solved efficiently or more formally in polynomial time by a deterministic Turing machine. As such, emerging literature detail approximation algorithms that heuristically (i.e. efficiently, sacrificing quality of solution) approximate the solution. In 1974 [36], Johnson proposed a greedy heuristic with $O(\ln \Delta + 1)$-approximation, later proven to be asymptotically tight i.e. within a constant factor better than the best possible approximation possible. A naive implementation of this algorithm performs scale poorly in practice to large instances. Taking at least 10 hours for large instances of $\sim$ 5M elements, about 1.4GB [22]. Since then modifications of the original heuristic and efficient implements have been proposed [22, 40, 44] in the literature. As we explore in *Project Evaluation*, these algorithms still become I/O bound for massive sizes i.e. $\sim 10^{12}$.

As the streaming setting became unpopular for the design of big data graph algorithm, innovations in semi-streaming model were made. In 2016, lower bounds on single [26] and multi [20] pass approximations of the semi-streaming MSC were established with accompanying heuristics of asymptotically tight approximation.

1.4 Project Overview

The primary goals of this project are too:

- Discover and grasp salient literature on offline, semi-streaming and sublinear MSC algorithms.
- Adopt and compare techniques in efficient implementation to realize these algorithms. Primarily those of semi-streaming and sublinear literature for which efficient implementation hasn't been investigated.
- Investigate novel semi-streaming and sub-linear MSC heuristics that perform favourably in-practice possibly by adopting algorithmic techniques presented in existing literature.
- Evaluate the experimental performance and quality of solution of these algorithms in a variety of realistic scenarios.

1.5 Project Outline

In the remainder of Chapter 1 I highlight the need for research and experimental evaluation of algorithms in different computational models and the need for big-data algorithms for the MSC problem. In Chapter 2 I review literature that make significant contribution to our purpose. Those that propose efficient offline alternatives to the classical greedy algorithm, interesting algorithmic techniques and novel semi-streaming and sublinear heuristics for the MSC. In Chapter 3 I detail our approach to efficient implementation, minor algorithmic modifications to existing heuristics and our two significant contributions. Firstly, a semi-streaming heuristic appropriately named the Largest Set Heuristic (LSH) that computes a solution of marginally worse quality to the best performing semi-streaming heuristics in significantly less time, additionally being parallelizable to threaded or distributed computing environments giving it scalability to massive instances. Secondly, a plug-able design for sublinear set-covers (SLSC) that computes a solution of comparable quality to best performing offline heuristics utilizing dramatically less memory. In Chapter 4 I evaluate the runtime performance and solution quality of these algorithms in a variety of realistic scenarios like when we alter the input instance to be a small, large or random, or make constraints on the amount of available RAM. In Chapter 5, I summaries our findings, objectively concluding whether we've move any closer to our overarching goal of finding scalable, big-data algorithm for the MSC problem.

1.6 Motivation

1.6.1 Computational models

From a theoretical perspective, investigation of semi-streaming and sublinear algorithms gives a measure of quality to solutions feasible in these computational models. Pertaining to our project, the in-practice performance metrics that more often than not strays away from theoretical guarantees bridge the gap between theory and practice and test the their viability as examples of big-data algorithms for massive problem instances.

Hallmark characteristics of big-data algorithms are runtime and memory usage for which the newly proposed, faster semi-streaming and low memory sublinear heuristic are good candidates to consider.

1.6.2 Big-data MSC

Any optimization problem that can be formulated as a set cover that prioritize scalability or real time processing are candidates for applications of the semi-streaming MSC. For example, the translation of a graph into an instance of the set cover problem defined by mapping vertices in a graph into a set which contains the neighbourhood of the vertex and itself allows for extracting an approximation for the Minimum Dominating Set (MDS), a salient property of a graph. As a quintessential example, In a social network where a node represents a member and a set their friends, the problem of the obtaining the smallest group of people which connections to everyone else on the network can be formulated as a instance of the MSC problem.

Graphs have become the de-facto standard for representing many types of highly-structured data e.g., web-pages and hyperlinks; neurons and synapses; papers and citations; IP addresses and network flows; people and their friendships. However, analyzing these graphs via classical algorithms can be challenging given the sheer size of the graphs. For example, both the web graph and models of the human brain would use around 10^{10} nodes and IPv6 supports 2^{128} possible addresses emphasizing the need for scalable, big-data alternatives.

1.6.3 Applications of MSC

The MSC arises in wide range of settings, including operations research, machine learning, planning, data quality and data mining.

Web or Social graphs, exist as large instances and hence rely on practically applicable big data algorithms such as of those in the semi-streaming model. For example, Stergiou and Tsioutsiouliklis [44] describe the setting of a web crawler that generates seed URLs that scales the crawled corpus at a rate approximately 4x faster to prior heuristics. This was achieved in realizing a scalable seed generation algorithm as a layered set cover, a loose reformulation of the max *k*-set-cover. Of a similar nature, Saha and Getoor [43] proposed an blog-watch application that automated the finding and maintaining over time of the best k blogs covering a set of topics.

The shift scheduling problem is of commercial importance to any service organization to appropriately staff the demand of its goods or service. Integer linear programming approaches based on a set covering formulation [24] were initially suggested. Gaining popularly with the coming of the branch-and-price [41] algorithm that made formulating these complex ILP instances simpler and solving them on large scale feasible.

However, the greatest far-reaching impact of set cover involves its use in future planning where informed choices are made in buying resources with fluctuating costs and demands. When fluctuations are unpredictable, solutions are modelled as streaming algorithms under real time figures and statistics. An popular instance of future planning is automated network design of high speed networks. A solution was first suggested by Bellcore [23]. In their work the problem was fundamentally formulated as a time parameterization of the set cover problem where elements represent bandwidth demand and sets represent systems that can cover a collection of demands. They obtain a planning heuristics of the same accuracy as those in the standard static model [42], this result wasn't online (in real time) requiring a prediction of demand-cost fluctuations.

Optimization problems arising in a similar context include survivable networks [29] in automated network design and facility location [35] including natural extensions of these problems under projections of increasing demands and decreasing costs which remain open problems.

Chapter 2

Technical Background

Given an input pair $(\mathcal{U}, \mathcal{S})$ where $\mathcal{U}$ is a set of n elements called the universe and $\mathcal{S}$ is a collection of m sets commonly referred to as the set system whose union equals the universe. The set-cover optimization problem or minimum set-cover (MSC) is finding the smallest subset of $\mathcal{S}$ that covers (i.e. whose union equals) the entire universe. As previously mentioned the MSC problem is NP-complete and can only be approximated in polynomial time. We now detail those approximation algorithms reported in existing literature.

A problem instance can be described by the following properties -

- *n* is the size of the universe $\mathcal{U}$.
- *m* is the size of the set system $\mathcal{S}$.
- *M* is the total number of items over all sets i.e. $M = \Sigma_{i=0}^{m} |S_i|$.
- *avg* $|S_i|$ is the average set size.
- Δ is the maximum set size.

2.1 Naive Heuristic [21]

Early studies of the MSC made theoretical comparisons between the classical greedy algorithm and the naive heuristic [21]. This algorithm and its approximation-guarantee is an important finding as we find its analogue in the semi-streaming model.

Algorithm 1 Naive Heuristic

```
1: procedure NAIVE($\mathcal{U}, \mathcal{S}$)
2:     Sort $S_i \in \mathcal{S}$ by non-increasing cardinality of $|S_i|$
3:     for all $S_i \in \mathcal{S}$ do
4:         if $|S_i \setminus C| > 0$ then
5:             $Sol \leftarrow id(S_i)$
6:             $C \leftarrow C \cup S_i$
7:         end if
8:     end for
9: end procedure
```

The naive heuristic operates by initially sorting the collection of sets $\mathcal{S}$ in non-increasing order of set cardinality (i.e. number of elements), subsequently iterating through them including a set iff. it contains one or more uncovered elements. In affect covering every element with its largest set.

Lemma 2.1.1. *Adapting the proof of [22] for a non-increasing sort. The algorithms approximation is upper bounded by $n/4$.*

Proof. Consider $\mathcal{U} = \{1, \ldots, 2k\}$ and $S_i = \{i, i+1, \ldots, k+i\} \in \mathcal{S}$ for $1 \leq i \leq k$ where $|S_i| = k$, therefore being in non-decreasing order of size.

The algorithm will always add all k sets to its solution as S_i contains currently uncovered element $k+i$. The exact solution however is 2 ($S_1 \cup S_k = \{1, \dots, 2k\}$), therefore the worst case approximation-factor is $k/2 = (n/2)/2 = n/4$. □

2.1.1 Greedy [36]

In 1974 [36], Johnson was first to propose and prove a factor $(\ln n + 1)$-approximation classical greedy algorithm. This was later shown to be asymptotically tight in 1998 as no polynomial-time $(1 - o(1)) \cdot \ln n$-approximation algorithm can exist unless NP has slightly super-polynomial time algorithms [27].

Algorithm 2 Greedy Algorithm

```
1: procedure GREEDY($\mathcal{U}, \mathcal{S}$)
2:     $Sol = \emptyset$
3:     $C = \emptyset$
4:     while $C \neq \mathcal{U}$ do
5:         $S_k = argmax_{S_i \in \mathcal{S}}(S_i \cap C)$
6:         $C = C \cup S_k$
7:         $Sol = Sol \cup id(S_k)$
8:     end while
9: end procedure
```

A brief summary of its approximation proof will aid the understanding of future, more performant variants of the greedy algorithm.

Lemma 2.1.2. *Greedy has a* $(\ln n + 1)$*-approximation [36].*

Proof. Let OPT be the optimal solution. Let C_t be the covered elements at time t. At time t, there must be some set of size at least $|\mathcal{U} \setminus C_t|/OPT$ uncovered elements, otherwise the optimal solution OPT does not exist. Therefore $|\mathcal{U} \setminus C_{t+1}| \leq |\mathcal{U} \setminus C_t|(1 - 1/OPT)$, as the number of uncovered elements must of decreased by at least $|\mathcal{U} \setminus C_t|/OPT$ elements. Extrapolating this invariant from the base-case when $|\mathcal{U} \setminus C_0| = n$, the number of uncovered elements at time t is at least $n(1 - 1/OPT)^t < n \cdot e^{-t/OPT}$. When $t = OPT \cdot \ln n$ sets have been chosen, all elements of the universe are guaranteed to covered. Since this value might not be an integer we upper-bound the number of sets chosen $\lceil OPT \cdot \ln n \rceil \leq OPT \cdot \ln n + 1 < OPT(\ln n + 1)$ to conveniently prove an approximation-factor of $\ln n + 1$. □

2.1.2 Efficient Implementation

Particular attention must be given to appropriate use of data-structures to facilitate a speedy implementation. In the following we describe the particular components required to realize this, additionally giving context to how future literature improved upon this and to what degree.

Determining as the algorithm progresses which elements of the universe have so far been covered is easily tracked using a n-bit bit-map, where 1 or 0 represents whether the element has been covered or not.

After finding the set of next largest uncovered cardinality S_k, we must determine and update the remaining sets that contain the newly covered elements of S_k. To quickly determine these sets we maintain an inverted index that for each element lists the identifiers of the sets that contain it. Time taken to generate this set is linear in the problem instance i.e. O(M).

The greedy algorithm must also maintain a ”priority-queue” data-structure that encapsulates the set system and supports 2 operations, extract-maximum and update-set. Extract-maximum finds the next optimal set and update-set updates the uncovered cardinality of a given set in removing elements that have been covered so far.

A key aspect in the implementation of the greedy algorithm is the choice of data-structure to achieve this priority-queue. Cormode et al. [22] analyses two data-structures.

MultiPass, a greedy variant uses an array-based "priority-queue". An extract-max operation consequently requires the user to traverse the array of sets performing comparison operations to find the next largest set in O(m) time. An update-set operation however can accesses a set of choice in constant O(1) time. Notice that the operations may be interleaved, updating the uncovered cardinality before a comparison of uncovered cardinality is made as the set system is scanned, removing the need for an inverted-index. The aggregate runtime of this implementation is upper-bounded by $O(|Sol| \cdot \Sigma|S_i|)$ or more simply $O(|Sol| \cdot M)$ as every element of the set system must be checked for coverage (known as a check-covered operation) every iteration a total of $|Sol|$ times where $|Sol|$ is the size of the solution and therefore the number of iterations before termination.

An alternative max-heap "priority-queue" requires only a constant overhead O(1) extract-max operation to obtain the root element. The update-set operation however takes a longer $O(\log m)$ time. This term dominates the runtime and is upper-bounded by $O(M \cdot \log m)$ as all sets can be cleaned a total of M times if every set-update operation only removed a single newly uncovered element.

Through experimentation the multiple pass algorithm showed poor scalability as it is required to read through the entire problem instance to include a single set. The max-heap, inverted-index approach was far more conservative in the number of times set-update operations and became known in future works as the conventional efficient implementation of the greedy algorithm.

2.2 Disk Friendly Greedy [22]

It seems unpredictable which sets are to be cleaned each iteration and it is highly probably that these sets aren't sequentially placed, especially as the set system grows. This frequent and random-access to memory or disk make such an algorithm regardless of implementation, un-scalable.

2.2.1 Algorithm.

In 2010, Cormode et al. proposed the *Disk Friendly Greedy* (DFG), an efficient, scalable algorithmic variant of the classical greedy algorithm of a provably small margin inaccurate.

The algorithm begins with a pre-processing step that partition the sets into sub-collections, assigning a set S_i into sub-collection SS^k iff. $\mathsf{p}^k \leq |S_i| < \mathsf{p}^{k+1}$. The algorithm then proceeds in two loops. The first loop iterates over the buckets of which there are $log_p\Delta$ (where Δ is the size of the largest set of the set system) in order of decreasing k. The second loop examines each set of the bucket in sequential order, cleaning it by removing already covered elements and including it into the solution iff. after cleaning it still belongs in that bucket i.e. its set size is still within the limits of its bucket's range. Otherwise the cleaned set is re-inserted at a lower bucket. In the base case k=1 bucket, if a set of the bucket contains any uncovered elements, it is immediately added to the solution.

instead of choosing the next optimal set with largest number of uncovered elements, it chooses to include a set that is only within a small fraction p worse. Allowing the heuristic to be implemented in a disk friendly manner to perform well in practice with only a marginally worse approximation ratio of a small constant factor p.

Approximation Factor. Following from aforementioned proof of $(\ln n + 1)$-approximation factor for the classical greedy algorithm. Cormode et al. proved that DFG produces a solution of only marginally worse $(p \ln n + 1)$-approximation.

The bucketing techniques makes it such that a set S_i that remains in its bucket with range $p^k \leq |S_i| < p^{k+1}$ after being cleaned is atmost a factor p worse than the optimal set. The next chosen set therefore contains $\mathsf{p} \cdot |\mathcal{U} \setminus C_t| / |OPT|$ uncovered elements. Carrying this through the calculation of greedy's approximation proof the solution size is strictly less than $1 + p \ln n \cdot |OPT| \leq (1 + p \ln n) \cdot |OPT|$.

Runtime. An equally positive characteristic of the bucketing technique is its implication on the theoretical runtime.

Algorithm 3 Disk Friendly Greedy

```
procedure PREPROCESS($\mathcal{U}, \mathcal{S}$)
    $K \leftarrow \lceil log_p \Delta \rceil$
    for all $k = \{K, K-1, \dots, 1\}$ do
        $SS^k \leftarrow \emptyset$
    end for
    for all $S_i \in \mathcal{S}$ do
        $SS^k \leftarrow SS^k \cup \{S\}$ where $p^k \leq |S_i| < p^{k+1}$
    end for
    return $SS$
end procedure

procedure DFG($\mathcal{U}, \mathcal{S}$)
    $SS \leftarrow$ PREPROCESS($\mathcal{V}, \mathcal{S}$)
    for all $k = \{K, K-1, \dots, 1\}$ do
        for all $S_i \in SS^k$ do
            if $|S_i \setminus C| \geq p^k$ then
                $Sol \leftarrow Sol \cup \{i\}$
                $C \leftarrow C \cup (S_i \setminus C)$
            else
                $S_i \leftarrow |S_i \setminus C|$
                $SS^{k'} \leftarrow SS^{k'} \cup \{S_i\}$ where $p^{k'} \leq |S_i| < p^{k'+1}$
            end if
        end for
    end for
    return $Sol$
end procedure
```

At worst a set is cleaned and migrated atmost $log_p\Delta$ times, shrinking by a factor p each time. An upper-bound on the total number of check-covered operations performed on a set can be captured in the summation of this infinitely decreasing geometric series

$$|S_i| + \frac{|S_i|}{p} + \frac{|S_i|}{p^2} \cdots = (1 + \frac{1}{p-1})|S_i|$$

. The union bound over all sets thus gives us a worst-case runtime of $(1 + \frac{1}{p-1})\Sigma_i |S_i|$, which is barely a factor $\frac{1}{p-1}$ larger than a sequential scan over the set system. More commonly in practice, sets are expected to drop multiple levels.

2.2.2 Memory/Disk Considerations.

Importantly the scalability of the Disk-Friendly greedy comes from a maximization of sequential operations by the spatial locality of sets in a bucket. However a significant criticism of its scalability is the random-access writes to lower buckets when a set is migrated. This operation, if frequent, cannot scale to *massive* problem instances for which the program tends to be I/O bound.

Practical results nonetheless show a dramatic improvement in speed as a consequence of reduced IO operations, specifically in the case when the program is more likely to become I/O bound (i.e the problem instance fits doesn't fit in memory). Experimental evaluation of Cormode et al. saw a 2x performance improvement for RAM-based experiments and a much larger 37x improvement for Disk-based experiments for their *kosarak* dataset with a solution of only 0.5% more sets than greedy. We refer to Table 3 and Table 4 of [22] for all details of the experiments conducted.

2.3 WebScale [44]

In ”Set Covers at WebScale” [44] Stergiou & Tsioutsiouliklis introduces two algorithmic techniques to augment serial offline MSC heuristics.

FGreedy. At each iteration of the classical greedy algorithm all newly covered element of the next optimal set are eagerly removed from all remaining sets. We can notice the runtime of MultiPass or conventional greedy is dominated by the overhead of set-update operations. This is true more generally, regardless of data-structure. This performance penalty was identified and remedied by deferring updates until required. In particular by utilizing a max-heap ”priority-queue” if the cardinality of the root-element is updated and after re-insertion remains the root, it is the next best set, otherwise this process is repeated until so. This is done hopefully so as to make a significant trade-off of a large number set update operations for an additional smaller number of extract-max operations. Surprisingly, the number of extract-max operations now dominates the number of update-set operations as the set of an extract-max operation is either re-inserted or chosen into the solution. The effect of this manipulation on the runtime was further investigated by Lim et al. [40] further discussed in 2.4.

RFGreedy. Analogous to the bucketing technique of DFG, Stergiou & Tsioutsiouliklis propose including a set into the final solution if after updating its uncovered cardinality, its size remains to be only a small constant p smaller. Importantly, this shows this technique is generalizable beyond a collection of buckets to other sorted or ordered ”priority-queues” such as a binary-heap.

2.4 Lazy Greedy [40]

Lim et al. [40] probes further into theoretical and experimental consequences of the algorithmic modification of deferring set-update operations appropriately named the lazy-greedy approach. Additionally proposing an efficient data-structure to implement the priority queue and algorithm that brings all these enhancements together.

2.4.1 Array-of-Lists

As expressed before, the classical greedy or eager-greedy is encumbered by the dominating overhead of a large number of set-update operations, this is less true for lazy-greedy however still accounting for a large fraction of the overall runtime. Disregarding reducing this proportion by further algorithmic modification, the $O(\log m)$ overhead of heap-based set-update is a limiting implementation factor.

Lim et al. identified and remedied this with a array-of-lists priority-queue where `pqueue[uc]` maps to a list of sets that contain uc uncovered elements. The largest non-empty index of `pqueue` therefore indicates the optimal set. After obtaining the largest element, a sequential scan through the lower indices of `pqueue` finds us the next optimal set. As a set can be at largest size n, the aggregate upper-bound on the overhead of extract-max operations is $O(n+m)$. A set-update operation involves deleting a set from its current list and re-linking it to a list at a lower index in $O(1)$ time. In summary making a significant trade-off in the cumulative burdening overhead of previously $O(\log M)$ now $O(1)$-cost set-update operations for an additional small overhead of $O(n)$ in iterating over the `pqueue` array. With this modification, the dominating term falls from $O(M \cdot \log M)$ to $O(M)$.

2.4.2 Worst-Case Runtime

Lim et al. show through example that such a deferred set-updating can in some difficult problem instances of size M require the lazy-greedy method to spend $\Theta(M^{4/3})$ time performing about $\Theta(M^{2/3})$ extract-max operations. This is asymptotically greater than the runtime of eager-greedy $O(M)$.

2.4.3 Experimentation

In experiments Lim et al. compared the number of elementary operations namely extract-max, update-set and check-covered, under different sized ”very large” problem instances. We copy the summary of these experiments in Table 4 [40] verbatim.

Operations	Dataset	Eager-Greedy	Lazy-Greedy	DFG (p=1.1)
Solution Size	Retail	5,106	5,113	5,119
	Accidents	181	180	185
	Webdocs	406,372	406,400	406,428
Extract-Maximum	Retail	5,106	162,961	-
	Accidents	181	1,080,524	-
	Webdocs	406,372	3,291,586	-
Check-Covered	Retail	92,540	1,239,924	1,223,062
	Accidents	6,630	18,170,446	17,766,461
	Webdocs	133,008,957	335,085,502	326,336,293
Queue-Reinsertion	Retail	778,704	74,817	74,009
	Accidents	2,365,229	743,400	709,066
	Webdocs	46,092,276	1,559,610	1,562,286
Runtime (s)	Retail	0.39	0.18	0.18
	Accidents	1.53	0.85	0.84
	Webdocs	73.10	5.34	5.11

Table 2.1: [40] Table 4

We refer to the largest dataset, webdocs. A few observations can be made. The number of queue re-insertions (or set-update) operations see a 30 times reduction for only a 8 times and 2.5 times increase in extract-maximum and check-covered operations respectively. However these operations are fast, check-covered entailing only a mask and array access on a n-bit data-structure and similarly (bar the mask operation) the case for obtaining the root of a heap-based priority queue. This assumption is justified by the dramatic difference in measured execution time.

2.4.4 Comparison with DFG

The efficiency of the DFG algorithm is in the scalability of its bucketing technique. As the number of sets grow larger than the maximum size of a set, the size of buckets become large, maximizing the number of sequential operations over data, scaling (more) gracefully with growing input when it cannot be stored in-memory.

This same sentiment is reflected in the list-of-arrays structure of lazy-greedy, although on a much granular level. This being the key difference in the two algorithms, lazy-greedy maintains n lists (or analogously buckets) whereas DFG only maintains $\log_p n$ of these, which for an appropriately large estimation factor p, can better maximizes the number of sequential operations (as a consequence of larger buckets/lists) and better minimize the number of queue re-insertions (as there are less buckets/lists). This is reflected in the as low if not lower number of elementary operations of an execution of DFG comparatively to eager-greedy and even lazy-greedy 2.1.

Surprisingly, course grained bucketing places an upper-bound on the total number of queue re-insertions by placing a hard $O(\log_p n)$ upper-bound on the number of times a set can be cleaned (by accepting a marginally worse set) thereby as-well upper-bounding the worst-case runtime, a previously mentioned hindrance of lazy greedy. If we refer to Table 3 of [40] we notice for datasets of $M \gg n$, lazy-greedy's runtime tends towards its worst-case approximation with eager-greedy performing 4 times faster and DFG, 40 times faster.

Intuitively, DFG is a scalable generalization of the lazy greedy algorithm for a variable bucket size, put another way, lazy greedy is the optimized equivalent of DFG as $p \to 1$ (p tends to 1). This generalization however still provides a high quality solution of experimental estimation factor much lower than p, if we refer to webdocs 2.1, the solution is actually only 1% worse.

2.5 Semi-Streaming Model

Now that we've determined a arguably scalable, asymptotically tight, offline MSC algorithm, DFG. Future work will aim at making comparisons between this and streaming algorithms of semi-streaming and sublinear $o(m \cdot n)$ (strictly less than the size of the problem instance) space bounds.

To recap, the semi-streaming model enforces a computational model i.e. a set of rules and restrictions that govern the execution of the algorithm. In particular the set system can only be read in sequential order (i.e. as a stream), only $O(n \cdot log^{O(1)}n)$-space and a $O(log^{O(1)}n)$-pass-complexity (number of passes over the stream) is granted. These restriction bring several advantages. Firstly, streaming the set system maximizes the number of sequential memory-accesses, exploiting the spatial locality and cache pre-fetching mechanisms of modern machines. Secondly, a low sublinear space-bound dissuades algorithms becoming I/O bound, a programmatic state due to frequent, unpredictable random-accesses to memory. Finally, these heuristics commonly have succinct description and a simple, efficient implementation. However these come consequence of a trade-off in the approximation guarantee.

In summary, these advantages give strong suspect that semi-streaming and sublinear-space algorithms in-practice might be suitable candidates for big-data MSC on instances of ”massive” size with ideally a negligibly worse solution.

2.6 Single-Pass Semi-Streaming Algorithms

In [26], Rosen and Emek designed an single-pass MSC heuristic *SSSC (Semi-Streaming Set Cover)* of approximation factor $O(\sqrt{n})$. This was then verified to be asymptotically tight (even if we allow for randomization) by an accompanying lower-bound proof. An asymptotically optimal solution, performs at worst a constant factor times worse than the optimal solution. Consequentially this means, no alternate heuristic (possibly randomized) can obtain a solution any better than a constant factor to that obtained from SSSC. It is important to note that this is reasonably worse than what is provable in the offline model $\log n + 1$.

2.6.1 Algorithm

Algorithm 4 is a relaxation of the partial, weighted MSC detailed in the full-text.

Algorithm 4 SSSC

```
1:  procedure ROSEN($\mathcal{V}, \mathcal{S}$)
2:      for all $t = \{1, 2, \ldots, m\}$ do
3:          Read set $s_t \in \mathcal{S}$ from the stream
4:          Compute the subset effective subset $T \subseteq s_t$
5:          for all $v \in \mathcal{T}$ do
6:              sid($v$) $\leftarrow t$
7:              eff($v$) $\leftarrow \log_2 |T|$
8:          end for
9:      end for
10:     return sid
11: end procedure
```

The algorithm makes a single sequential scan through the set system, updating a state captured in two data-structures sid and eff. Returning a cover-certificate representing the solution to a MSC, a function that maps every element of the universe to the set that covers it.

State. Data-structures sid and eff are functions of signatures $\mathcal{U} \to [1, m]$ and $\mathcal{U} \to [1, n]$ respectively. That is to say for every element $v \in \mathcal{U}$, eid maintains a mapping to the identifier of the set that most effectively covers that element. eff for the same element maintains a positive integer representing the effectivity that the set covers the element.

Update Rule. Consider set s_t at time t and some subset $T \subseteq s_t$. The effectively of subset T is defined by $\text{eff}_t(T) = log_2|T|$. Subset T is chosen as an effective subset to cover all $v \in T$ at time t iff. $\forall v \in T$, $\text{eff}_t(T) > \text{eff}_t(v)$.
We compute the most effective subset T (if one exists) for set s_t at time t as follows.

1. Sort the elements of set s_t in non-decreasing order of effectivity eff(v).
2. We iterate through the sorted set from the most effective element to find an element s.t. the number of elements of lower or equal effectivity (that will comprise the potential subset T) is twice as large as the element's effectivity.

Return. The algorithm on termination returns a cover-certificate $CC : V- > [1, m]$. A function, mapping every element of the universe to an identifier of the most effective set that covers it.

Space. The algorithmic data-structures eff and sid stores a single variable of maximum size n and m respectively for every element of the n-element universe. Since $\log x$-bits are requires to store a value x, the total space of both data-structures is upper-bounded by $n \cdot \log n + n \cdot \log m \leq O(n \cdot \log^{O(1)} n)$.

Runtime. It takes $O(|s_t| \cdot \log |s_t|)$ time to perform a non-decreasing sort over the elements of set s_t. Finding the most effective subset at time t of s_t (if one exists) over these sorted vertices takes $O(|s_t|)$. A cover-certificate can be extracted in $O(1)$ time by simply return the size $O(n \cdot \log m)$ structure sid. Processing over all sets of the set system, the aggregate runtime can be described by $O(\Sigma(|s_t| \cdot \log |s_t|) + \Sigma|s_t| + 1)$ or more compactly as $O(\Sigma|s_t| \cdot \log |s_t|)$ by emphasizing the dominating term.

We refer to subsection **??** for details of the efficient implementation of this algorithm.

2.7 Multi-Pass Semi-Streaming Algorithms

In 2016 Chakrabarti & Wirth [20], answer an open question raised by Saha & Getoor [43] who . In particular, what is the tight trade-off between the pass-complexity p in the semi-streaming model w.r.t. it's approximation guarantee. Decisively proving that no approximation better than $0.99 \cdot n^{1/(p+1)}/(p+1)^2$ is achievable when we grant p-passes over the set stream, even if we allow randomization. Additionally designing a deterministic p-pass semi-streaming heuristic to the set cover problem with asymptotically tight $(p + 1) \cdot n^{1/(p+1)}$-approximation factor. Importantly, this guarantee implies a $2 \log n$-approximation if we grant an about logarithmic number of passes $\log_2 n - 1$.

2.7.1 Algorithm

The algorithm iteratively for sequence of geometrically decreasing thresholds τ_j governed by the number of passes p, scans the set stream $\mathcal{S}$, selecting a set iff. it covers at least as many uncovered elements as the threshold τ_j for the current iteration j.

Folding
The *Naive Progressive Greedy* obtains a p-pass $p \cdot n^{1/p}$-approximation. A modification involving folding the last 2-passes into a single pass, achieves the advertised $(p + 1) \cdot n^{1/(p+1)}$-approximation. The (p+1)th-pass is run as usual, additionally in parallel a simple backup cover-certificate is built $CC : V- > [1, m]$ mapping every vertex of the universe to a set that covers it. In a final post-processing step after the penultimate pass, any uncovered elements are covered by choosing the corresponding mapping of the backup certificate cover.

Runtime.
For every pass, every set of the set stream must be cleaned. Over p-passes, this accumulates a total of p.M check-covered operations. Additionally the Sol and $Cover$ data-structures are only updated a total of $p \cdot n^{1/p}$ (following from the approximation guarantee) and n times respectively. The runtime is therefore $O(p \cdot M + p \cdot n^{1/p} + n)$ or $O(p \cdot M)$ if we emphasize the dominating term.

Algorithm 5 Naive Progressive Greedy
,
```
procedure GreedyPass($\mathcal{S}, \tau, Sol, Cover$)
    for all $t = \{0, 1, \ldots, m\}$ do
        Read set $s_t \in \mathcal{S}$ from the stream
        if $|s_t \setminus C| \geq \tau$ then
            $Sol \leftarrow Sol \cup \{t\}$
            for all $x \in s_t \setminus C$ do
                $Cover[x] \leftarrow t$
            end for
        end if
    end for
end procedure

procedure NaiveProgressiveGreedy($\mathcal{V}, \mathcal{S}$)
    $Cover[1, n] \leftarrow 0$
    $Sol \leftarrow \emptyset$
    for all $j = \{0, 1, \ldots, p\}$ do GreedyPass($\mathcal{S}, n^{1-j/p}, Sol, Cover$)
    end for
    return $Sol, Cover$
end procedure
```

2.8 Streaming in Sublinear-Space

An emerging area of interest following from theoretical work in the semi-streaming MSC is the proof of approximation guarantees better than that possible in the semi-streaming model when we allow for larger than semi-streaming $O(n \cdot log^{O(1)} n)$ but still sublinear $o(m \cdot n)$ space. More precisely, what is the tight approximation-space trade-off for single and multiple pass MSC algorithms in the streaming model.

What is the in-practice solution quality and performance and how do these compare to those of competitive offline and semi-streaming heuristics. Does the additional space produce algorithms of significantly better approximation or performance and are these more viable as scalable, big-data alternatives.

2.9 Deterministic Single-Pass Sublinear-Space Streaming Algorithms

2.9.1 Lowerbound

Following the work of Rosen & Emek [26] on a strict $O(\sqrt{n})$-approximation lower-bound for single-pass MSC algorithms in the semi-streaming mode, Assadi et al. [33] gave an tight lower-bound on the trade-off in space w.r.t. approximation. In specific they prove that for a $\alpha = o(\sqrt{n})$-approximation where $m = n^{O(1)}$, any single-pass streaming algorithm (possibly randomized with high success probability) that α-approximates the set-cover requires at least $\Omega(\frac{m \cdot n}{\alpha})$-space.

Notice that this space-bound is only true for $\alpha = o(\sqrt{n})$, an approximation strictly better than that possible for single-pass *semi-streaming* algorithms, as the space-requirement in the semi-streaming model $O(n \cdot \log^{O(1)} n)$ is respectably lower than that imposed by the sublinear space-requirement $O(\frac{m \cdot n}{\sqrt{n}}) = O(m \cdot \sqrt{n})$ and of an equivalent approximation guarantee. Importantly, this establishes that there is no non-trivial trade-off in space for approximations better than that possible by semi-streaming heuristics, an open-question raised by Indyk et al. [33].

2.9.2 Algorithm

To this they propose a α-approximation MSC heuristic of asymptotically tight (w.r.t. the tight trade-off) space-bound $\Theta(\frac{m \cdot n}{\alpha})$ when $\alpha = o(\sqrt{n})$. The deterministic algorithm is as follows, Merge every α sets of the stream as a single set, storing these sets in memory, filtering duplicate elements as necessary. An exact MSC can be computed on this reduced instance, awarding at worst an α-approximation.

The $O(\frac{m \cdot n}{\alpha})$ space-bound can be justified if we notice that the projection of α sets of size atmost n is still atmost n, reducing the size of the problem instance by atmost a factor of α in the worst case problem instance $O(m \cdot n)$ i.e. when all sets are of size n. The α-approximation is a consequence of every set of OPT (the optimal solution), requiring only a single non-merged set from a merged set, dragging the other $\alpha - 1$ redundant sets into the final solution.

We will explore the implementation and experimental performance of this algorithm in Chapter 3.

2.10 Randomized Single-Pass Sublinear Space Streaming Algorithms

While we've seen examples of deterministic algorithms, techniques in randomized MSC algorithms have become popular since their advent in 2014 [25]. Their purpose being to still give an approximation as close to that possible in the offline model at decreased space-bounds. This interest in randomization was first sparked by Demaine et al. [25] they proved that any deterministic constant factor approximation algorithm for the set streaming set-Cover with constant number of passes requires $\Omega(mn)$ space, inspiring the use of randomization to achieve low approximations.

Algorithm 6 Projected Set Cover

,

```
1: procedure ProjectedSetCover($\mathcal{U}_{red}, \mathcal{S}$)
2:     $\mathcal{S}_{red} \leftarrow \{S_i \cap \mathcal{U}_{red} | S_i \in \mathcal{S}\}$
3:     $sol_{red} \leftarrow$ OfflineSetCover($\mathrm{U}_{red}, \mathcal{S}_{red}$)
4:     return $sol_{red}$
5: end procedure
```

We start by defining by a fundamental component of these randomized sublinear-space streaming algorithms, the Projected Set Cover 6. It works by reading the set system into RAM, projecting them onto the reduced element space $\mathcal{U}_{red}$ and passing this reduced set-cover instance $(\mathcal{U}_{red}, \mathcal{S}_{red})$ into any appropriate offline MSC algorithm.

2.10.1 Set Sampling

The following 2.10.1 is the set sampling lemma of Demaine et al. [25] which provides a guarantee w.h.p (with high probability) between the number of randomly sampled sets of the set stream and the average number of sets that contain the remaining uncovered elements.

Algorithm 7 Set Sampling

,

```
1: procedure SetSampling($\mathcal{U}, \mathcal{S}$)
2:     Let $S_{rnd}$ be a collection of $L$ sets of $\mathcal{S}$ picked uniformly at random
3:     $\mathcal{U}_{rem} \leftarrow \mathcal{U} \setminus \bigcup_{S_i \in \mathcal{S}} S_i$
4:     if each element of $\mathcal{U}_{rem}$ appears in less than $\frac{mc \log n}{L}$ sets of $\mathcal{S}$ then
5:         $sol_{rnd} \leftarrow$ ProjectedSetCover($\mathrm{U}_{rem}, \mathcal{S}$)
6:         return $S_{rnd} \cup s_{rnd}$
7:     else
8:         return $INVALID$
9:     end if
10: end procedure
```

Lemma 2.10.1. *For the minimum set cover instance $(\mathcal{U}, \mathcal{S})$. Let S_{rnd} be a subset of L sets from $\mathcal{S}$ chosen uniformly at random. Then with probability $1 - \frac{1}{n^{c-1}}$, S_{rnd} covers all elements of $\mathcal{U}$ that appear in atleast $\frac{mc \log n}{L}$ sets. Therefore w.h.p all uncovered elements of $\mathcal{U}$ appear in atmost $\frac{mc \log n}{L}$ sets of $\mathcal{S}$.*

This set sampling can be leveraged as a pre-processing to a following offline MSC algorithm as described 7 to give a $O(\frac{L}{|OPT|} + p)$-approximation, $O(\frac{mc \log n}{L} \cdot n) = \tilde{O}(\frac{m.n}{L})$ sublinear space MSC algorithm where ρ is the approximation-factor of the offline MSC algorithm. The approximation follows from the fact that we initially sample L sets followed by atmost an additional $p.|OPT|$ sets from the offline algorithm. The space-bound follows from the fact that on average $\frac{mc \log n}{L}$ sets contain all uncovered elements which is atmost n. Moreover if we sample $L = p.k$ where and k is an $O(1)$-approximation of the size of the optimal solution $|OPT|$, $S_{rnd} \cup sol_{rnd}$ is atmost $2kp$ still guaranteeing a low offline-like $O(\rho)$-approximation.

2.10.2 Element Sampling

Algorithm 8 Element Sampling

,

```
 1: procedure ELEMENT SAMPLING($\mathcal{U}, \mathcal{S}, n$)
 2:     Let $\mathcal{U}_{sample}$ be a subset of $\mathcal{U}$ of size $c\sqrt{pnk \log m}$ sampled uniformly at random
 3:     $Sol_{sample} \leftarrow$ PROJECTEDSETCOVER($\mathcal{U}_{sample}, \mathcal{S}$)
 4:     $\mathcal{U}_{rem} = \mathcal{U} / \bigcup_{S_i \in Sol_{sample}} S_i$
 5:     if $|\mathcal{U}_{rem}| \leq \sqrt{pnk \log m}$ then
 6:         $Sol_{rem} \leftarrow$ PROJECTEDSETCOVER($\mathcal{U}_{rem}, \mathcal{S}$)
 7:         return ($Sol_{sample} \cup Sol_{rem}$)
 8:     else
 9:         return invalid
10:     end if
11: end procedure
```

Lemma 2.10.2. *Given an set-cover instance $(\mathcal{U}, \mathcal{S})$ and k, the size of the optimal set cover. Let $\mathcal{U}_{sample}$ be a sample of $\mathcal{U}$ of size $O(pk \log m/\varepsilon)$ chosen uniformly at random where $\varepsilon < 1$. A p-approximate cover C_{sample} of $(\mathcal{U}_{sample}, \mathcal{S}_{sample})$ where $\mathcal{S}_{sample} = \{S_i \cap \mathcal{U}_{sample} | S_i \in \mathcal{S}\}$ corresponds to a ε-cover (a partial set-covering where a fraction $1 - \varepsilon$ of the universe is covered) of $\mathcal{U}$ with high-probability. In particular if $\mathcal{U}_{sample}$ is of size $cpk \log m/\varepsilon$, then C_{sample} is a ε-cover of $(\mathcal{U}, \mathcal{S})$ with probability $1 - \frac{1}{m^{(c-2)}}$.*

We can design a 2-pass element sampling algorithm as detailed 8 by performing a single element-sampling, a check whether the sampling succeeded, followed by a p-approximation of the remaining element space. This algorithm has $2p$-approximation and $O(m \cdot pk \log m/\varepsilon + m \cdot \varepsilon n)$-space whose space-bound can be optimized to $O(m \cdot \sqrt{pkn \log m})$ for $\varepsilon = \sqrt{\frac{pk \log m}{n}}$.

2.11 Randomized Multi-Pass Sublinear-Space Streaming Algorithms

2.11.1 Lowerbound

Following the work of Chakrabarti & Wirth [20] on a strict $0.99n^{1/(p+1)}/(p+1)^2$-approximation lowerbound for all p-pass MSC algorithms in the semi-streaming model, Assadi [12] later gave insight into the space-approximation trade-off for the streaming set cover when we allow for a relatively smaller number of passes, say $\log n^{O(1)}$.

In particular they prove that any $\alpha = o(\log n / \log \log n)$-approximation (a strictly better approximation than that possible in $(\log n^{O(1)})$-passes by a semi-streaming algorithm, $\log n^{O(1)}$-pass streaming algorithm) requires atleast $\Omega(mn^{1/\alpha})$-space.

2.11.2 Related Work

It was in [25] that Demaine et al. that the set and element sampling randomization techniques were introduced to construct algorithms of offline-like approximation factors in strictly less space. In this work they show an α-approximation, $O(\alpha)$-pass, $O(mn^{\Theta(1/\log \alpha)})$ sublinear-space streaming MSC algorithm. Har-Peled et al. [33] later improve over this algorithm by developing an algorithm of similar approximation, fewer passes and $O(mn^{\Theta(1/\alpha)})$-space. They further conjectured that the pass-space tradeoff might be tight supported by a $\Omega mn^{1/2p}$-space lowerbound for p-passes. In this work by Assadi [12], he shows by slightly modification the algorithm only requires $O(mn^{1/\alpha})$-space. In particular, they show an algorithm with $(\alpha + \varepsilon)$-approximation, $(2\alpha + 1$-pass, $\tilde{opt}(mn^{1/\alpha}/\varepsilon^2 + n/\varepsilon)$-space in the streaming setting.

2.11.3 Algorithm

Algorithm 9 Multiple Pass Assadi

’

```
1:  procedure MP-ASSADI($\mathcal{U}, \mathcal{S}, \alpha, k, \varepsilon$)
2:      $Sol \leftarrow \emptyset$
3:      for all $t = \{0, 1, \ldots, m\}$ do
4:          Read edge $s_t \in \mathcal{S}$ from the stream
5:          if $|s_t \cap \mathcal{U}| > n/(\varepsilon \cdot k)$ then
6:              $Sol \leftarrow Sol \cup \{t\}$
7:              $U \leftarrow U \setminus S_i$
8:          end if
9:      end for
10:     for j = 1 to $\alpha$ iterations do
11:         Let $\mathcal{U}_{sample}$ be a subset of $\mathcal{U}$ chosen by picking each element independently and w.p. $\frac{16pk\log m}{n^{1-1/\alpha}}$
12:         $Sol_{sample} \leftarrow$ PROJECTEDSETCOVER($\mathcal{U}_{sample}, \mathcal{S}$)
13:         $Sol \leftarrow Sol \cup Sol_{sample}$
14:         $\mathcal{U} = \mathcal{U} / \bigcup_{S_i \in Sol_{sample}} S_i$
15:     end for
16:     return $Sol$
17: end procedure
```

The algorithm makes use of k, an $(1+\varepsilon)$-approximation of $|OPT|$. As before the results of running MP-Assadi under an 2-approximation of $|OPT|$ where $\varepsilon = 1$ can be obtained by running the algorithm under $\log n$ guesses of $k \in [1, n]$ in parallel where thread i uses $k = 2^i$ returning the solution of the lowest value of k for which the entire universe is covered implying α successful element samplings.

The algorithm makes use of two intruiging techniques

Single shot pruning.
Firstly, a pruning step is employed that chooses sets of the stream that contains more than $n/(\varepsilon \cdot k)$ uncovered elements. This initial pass samples atmost $\varepsilon \cdot k$ sets and more importantly limiting the number of elements any set contains in $\mathcal{U}$ after pruning to $n/(\varepsilon \cdot k)$.

Iterative Element Sampling.
Secondly, the algorithm takes an iterative element samples approach to iteratively reduce the size of the uncovered universe by a constant factor, until all elements are covered. Specifically, if we uniformly at random sample from the universe with probability $\frac{16pk\log m}{n^{1-1/\alpha}}$ by the element sampling lemma, a p-approximation cover of $\mathcal{U}_{sample}$ corresponds to a $(n^{1/\alpha})$-cover of $\mathcal{U}$ w.p. $1 - \frac{1}{m^2}$ (See Lemma 3.11 [12]). Over α iterations if each element sampling succeeds the universe should be covered $\mathcal{U} \leq \frac{n}{n^{(1/\alpha)(\alpha)}} \leq 1$.

Chapter 3

Project Execution

This chapter details most of the main contributions of our paper. Our semi-streaming and sublinear-space streaming set-cover algorithms. Two algorithmic modifications of the Rosen [26] algorithm for speed and accuracy respectively. An assessment of all these algorithms on medium sized datasets, isolating the most competent of the group to be further assessed on large to massive graph datasets in the subsequent chapter.

3.1 Largest Set Heuristic

We first propose a simple deterministic semi-streaming heuristic, the *Largest Set Heursitic* (LSH) that offers several advantages over the single-pass semi-streaming algorithms -

- Computationally, more efficient if we make comparisons of an algorithms time complexity.
- It offers a simple and scalable implementation, being easily parallelised to threaded or distributed computing environments.
- Even when restricted to a single thread , implementations perform in significantly less time to other competitive offline and even semi-streaming algorithms.

3.1.1 Algorithm

The algorithm similar to Rosen, returns a cover certificate, a complete function $C : \mathcal{U} \to [1, m]$ that maps every element $v \in \mathcal{U}$ to an set identifier that identifies the largest set of $\mathcal{S}$ that contains v.

Algorithm 10 Largest Set Heuristic

```
1:  procedure LSH($\mathcal{U}, \mathcal{S}$)
2:      for all $t = \{0, 1, \ldots, m\}$ do
3:          Read set $s_t \in \mathcal{S}$ from the stream
4:          for all $v \in s_t$ do
5:              if $|s_t| > \text{eff}(v)$ then
6:                  $\text{eid}(v) \leftarrow t$
7:                  $\text{eff}(v) \leftarrow |s_t|$
8:              end if
9:          end for
10:     end for
11: end procedure
```

This algorithm is surprisingly the analogue of the offline naive heuristic mentioned in section 2.1 thereby generating a solution of identical quality. That is to say, following from the worst-case approximation ratio of the naive algorithm we show that the LSH performs at most a factor $n/4$ worse than the optimal solution.

Lemma 3.1.1. *The naive heuristic indeed captures for every element, the largest set that covers it. The same notion captured by LSH.*

Proof. Given Sol, the solution of naive. There is some set $S_{i'} \in Sol$ that is the largest set to contain some arbitrary $v \in \mathcal{U}$, otherwise by contradiction if a larger set $S_{i''}$ were present that contains v it would be present earlier in the non-decreasingly sorted set system and therefore chosen as element v would be still be uncovered. $\square$

3.2 Optimizing Rosen

The aforementioned Rosen algorithm 4 is a relaxation of the partial, weighted MSC. As such the algorithm can be further optimized when solving a simpler problem, the un-weighted MSC.

For an element v, once Rosen makes a choice of some effective subset $T_t = R(s_t) \subset s_t$ where $v \in T_t$, the next effective subset $T_k = R(T_k) \subset s_k$ where $k > t$ and $v \in T_k$ must be *twice* as effective i.e. $log_2|T_k| > log_2|T_t|$. This requirement on the update rule is an artifact of simplifying the proof of approximation guarantee. In many cases this is sub-optimal.

Consider the stream of sets $S_1 = \{1,2,3,4\}, S_2 = \{5,6,1,2\}, S_3 = \{7,8,1,2\}, S_4 = \{9,10,1,2\}, S_5 = \{5,6,7\}, S_6 = \{8,9,10\}$. By Rosen, $Sol = \{S_1, S_2, S_3, S_4\}$ where $R(S_1) = \{1,2,3,4\}, R(S_2) = \{5,6\}, R(S_3) = \{7,8\}, R(S_4) = \{9,10\}$. The optimal solution is infact $Sol_{OPT} = \{S_1, S_5, S_6\}$ where $R(S_1) = \{1,2,3,4\}, R(S_5) = \{5,6,7\}, R(S_6) = \{8,9,10\}$.

The sets S_5 and S_6 can cover the elements of sets S_2, S_3 and S_4 more effectively i.e. in less sets. Seeing as $|R(S_5)| > |R(S_2)|$ and $R(S_2) \subset R(S_5)$, $R(S_5)$ is the more optimal choice. Rosen however didn't make these choices seeing as $\lceil log_2|R(S_5)|\rceil \not> \lceil log_2|R(S_2)|\rceil$. As the effectivity is a non-decreasing quantity, by allowing Rosen to choose even marginally more optimal sets, we make an (if only minor) improvement in solution.

3.2.1 Adapting Proof of Approximation

We show a $f(n) = (\frac{r^*+2}{r^*-1} + 1) \cdot \sqrt{n}$-approximation factor where $\lim_{r^* \to \infty} f(n) = 2 \cdot \sqrt{n}$ for the modified Rosen (an un-weighted set-cover algorithm) A.1 by adapting the proof of $8 \cdot \sqrt{n}$ approximation for the Rosen (an un-relaxed weighted, partial set-cover algorithm) detailed by Rosen & Emek [26]. This indeed shows our algorithmic relaxation is sound with an approximation no worse than the original Rosen.

3.3 Speeding up Rosen

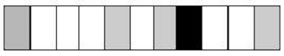
With this enhancement in approximation, we explore finding a similar improvement for its runtime complexity.

3.3.1 Randomizing Rosen

We experiment with a randomization of Rosen, appropriately named *Randomized Rosen*.

The algorithm chooses to compute an estimate of the most effective subset over a randomly sampled subset $r_t \subseteq s_t$ for all $s_t \in \mathcal{S}$ where $|r_t| = \min\{|s_t|, \tau\}$ and τ is some threshold. In summary aiming to make a trade-off between the threshold τ and the leading term of the runtime complexity associated with computing the effective effective subset, more specifically in sorting the elements of the set.

Update Rule.



We illustrate the logic behind the randomization technique with the above figure. Let blocks represent elements of some set s_t sorted in non-decreasing order of effectivity at time t. Let the grey elements be those randomly

Algorithm 11 Randomized-Rosen

```
procedure RANDOMIZED-ROSEN($\mathcal{V}, \mathcal{S}, \tau$)
    for all $t = \{0, 1, \dots, m\}$ do
        Read set $s_t \in \mathcal{S}$ from the stream
        for all $v \in s_t$ do
            if $|s_t| > \text{ceff}(v)$ then
                $\text{ceid}(v) \leftarrow t$
                $\text{ceff}(v) \leftarrow |s_t|$
            end if
        end for
        Randomly sample subset $r_t \subseteq s_t$ where $|r_t| \leq \tau$
        Compute the subset effective subset $T \subseteq r_t$
        for all $v \in \mathcal{T}$ do
            $\text{reid}(v) \leftarrow t$
            $\text{reff}(v) \leftarrow |T|$
        end for
    end for
    return $CC(v) = v \rightarrow reid(v) : reid(v) \ ? \ ceid(v)$
end procedure
```

sampled and the black element be that of largest effectivity s.t. the number of elements before it plus one is strictly larger than its effectivity i.e. they represent the most effective subset at time t.

For a uniform random sampling which gathers elements all across this array evenly, the grey elements closest to the black element (the optimal choice) are appropriate estimations of the most optimal choice, giving a comparatively good estimation of the most effective subset.

Specifically, the update rule is as follow

- Obtain a randomly sampling $r_t \subseteq s_t$ s.t. $|r_t| = \min\{|s_t|, \tau\}$ for some defined threshold τ.
- Calculate the fraction of elements sampled $p = |r_t|/|s_t|$.
- Sort the element of r_t by their effectivity at time t, iterating from the most effective element to find the element of largest effectivity s.t. $\frac{1}{p} *$ (the number of elements of lower effectivity) $+ 1$ is greater than its effectivity.
- Using this element whose effectivity is a good estimate of the effectivity of the optimal choice we iterate through original set finding all elements with equal or lower effectivity which will become part of the most effective subset at time t.

Certificate Cover However random sampling may drop some elements of the universe, failing to compute a complete 1-cover. To remedy this, in parallel we maintain a backup LSH cover-certificate , later used to fill any holes in the randomized certificate-cover reid.

Runtime. Following from the time complexity analysis of Rosen algorithm it is easy to see that the Randomized-Rosen runs in $t(\tau) = O(\Sigma\min\{|S_i| \cdot \log |S_i|, \tau \cdot \log \tau\}) \leq O(m \cdot \tau \cdot \log \tau)$ which performs better than the worst case runtime of the original Rosen $O(\Sigma|S_i| \cdot \log |S_i|) \leq O(m \cdot \Delta \cdot \log \Delta)$ when $\tau < \Delta$.

3.4 Dataset

We now lead into a series of experiments to quantify and qualify the competency of these algorithms. Our datasets are courtesy of the *Frequent Itemset Mining Dataset Repository* [3] that host several random and combinatorial itemset mining problem instances that are suitable as valid un-weighted set-cover instances.

System. All experiments on FIMI datasets were conducted on a 2.6 GHz Intel Core i7 running macOS High Sierra under light load. The system is equipped with 16GB of RAM. The program is compiled with g++ 8.3.0 and the

-O3 flag.

In table 3.1 we give a description of these datasets. We clarify what the $lowerbound$ metric is in the following subsection.

Size (Kb)	n	m	M	avg $\|S_i\|$	Δ	$lowerbound$
chess.dat	75	3196	118252	37	37	2
retail.dat	16470	88162	908576	10	76	314
pumsb.dat	2113	49046	3629404	74	74	29
kosarak.dat	41270	990002	8019015	8	2498	29
webdocs.dat	5,267,656	1,692,082	299,887,139	177	71,472	159

Table 3.1: FIMI Datasets

As the largest of these files took reasonably long to read-in, it was evident that conventional I/O mechanism weren't scalable enough for the projected 1TB file size. We instead read-only memory-mapped the file. A memory-mapping assigns a direct byte-to-byte correlation between the file itself and several pages of virtual memory, this technique was two advantages. Firstly, a read operation isn't treated as a potentially unsafe system call which must initially be authorized by the kernel, instead going directly to physical memory. Secondly, the allocated segment of virtual memory usually corresponds to the page cache [5], a segment of main memory dedicated to maintaining pages from disk, circumventing the need to copy these pages into the programs user-space for security reasons. We obtained this read-only memory-mapped file with the help of the Boost *iostream* header-only library and noticed a 16% speedup for the *webdocs.dat* dataset.

In the effort of verifying the correctness of our implement, an additional pass is taken to confirm that the entire universe is indeed covered by solution. For this endeavour we wrote a test suit for relevant algorithms using the *catch2* [2] C++ library.

3.4.1 Lowerbound

We would like to make a comparison of the solution size of MSC approximators such as the classical greedy w.r.t. the size of the exact solution. This would give us a measure of difficulty of the problem instance. For example, if the size of an approximation tends towards a factor close to its approximation-ratio larger than the exact solution $|OPT|$, this implies that these datasets are indeed "difficult" instances.

However, we cannot calculate the size of the exact solution directly as this is extending the decisional set-cover problem over the integers $[1, n]$. This is NP-complete [38] and thereby in-feasible for modestly large datasets. We therefore make an estimation of this value, appropriately named the dataset *lowerbound*, computed as the number of sets whose total element count equals or exceeds the size of the universe i.e. -

```
1:  procedure LOWERBOUND($\mathcal{U}, \mathcal{S}$)
2:      Sort $s_i \in \mathcal{S}$ by non-increasing cardinality of $|s_i|$
3:      $total = 0$
4:      for all $t = \{0, 1, \dots, m\}$ do
5:          Read set $s_t \in \mathcal{S}$
6:          $total = total + |s_t|$
7:          if $total \geq n$ then return $t$
8:          end if
9:      end for
10: end procedure
```

This calculation is almost always a significant underestimation as it assumes each of these set are disjoint, highly unlikely if we compute the ratio $\frac{M}{m}$ i.e. the number of sets an element on average appears in.

From our results, while on toy sized instances greedy performs within 10% of its optimal solution [31] the difference between our lowerbounds and later recorded experimental $(\log(n) + 1)$-approximations suggest these larger instances are much less trivial.

3.4.2 Experimental improvement of modified Rosen

In-practice the modified Rosen 3.14 shows an, if only minor improvement in solution size. For the remainder of this paper we alias the Modified-Rosen when referring to Rosen.

Algorithm	chess.dat	retail.dat	pumsb.dat	kosarak.dat	webdocs.dat
Rosen	13	6060	917	18760	416671
Modified Rosen	12	5854	901	18579	412656

Table 3.2: Improvement in Rosen solution quality

3.5 Experimental Results of Offline Algorithms

We perform some RAM-based experiments on the FIMI datasets, filtering from the runtime, the time required to read-in the set-system. Reproducing a few of these offline algorithms as faster C++ implementations we noticed similar trends as found in the literature. Here we introduce a new term known as *deviation* that measures as a percentage what magnitude worse is the solution w.r.t. a given baseline. We say a algorithm of low deviation is more *accurate*.

DFG (p = 1.05) and lazy-greedy performed almost identically, with comparable run-times and solution sizes, with DFG finding slightly better solutions for larger problems. The naive algorithm unsurprisingly wasn't scalable, consequence of the large up-front cost in sorting the sets. Interestingly, its solution progressively tended towards the optimal greedy solution for the larger problems, we questioned how this trend extrapolated for ”massive” datasets, becoming an initial discovery in justifying the need for LSH 3.1, a scalable alternative that discards the need for set sorting.

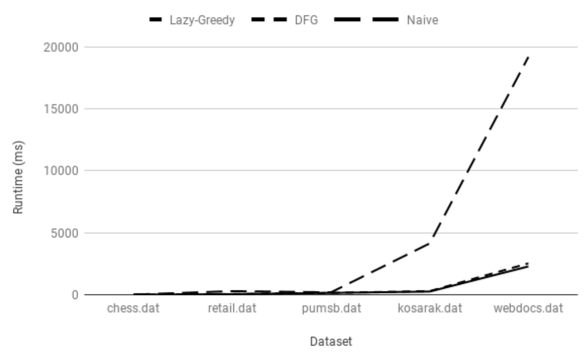
These results justify the aforementioned properties detailed in subsection 2.4.4, decisively concluding that DFG represents the best possible in the offline (or random-access) model and is a fair baseline for comparison, with an approximation competitive to the classical greedy at a performance and scalability leading its contenders.

Size (Kb)	Algorithm	Time (ms)	$\|Sol\|$
chess.dat	Lazy-Greedy	5	7
	DFG	4	8
	Naive	10	25
retail.dat	Lazy-Greedy	48	5,113
	DFG	75	5,125
	Naive	292	7,149
pumsb.dat	Lazy-Greedy	146	757
	DFG	189	735
	Naive	194	1,306
kosarak.dat	Lazy-Greedy	257	17,750
	DFG	336	17,738
	Naive	4,143	20,669
webdocs.dat	Lazy-Greedy	2,284	406,400
	DFG	3,021	406,357
	Naive	19,202	433,425

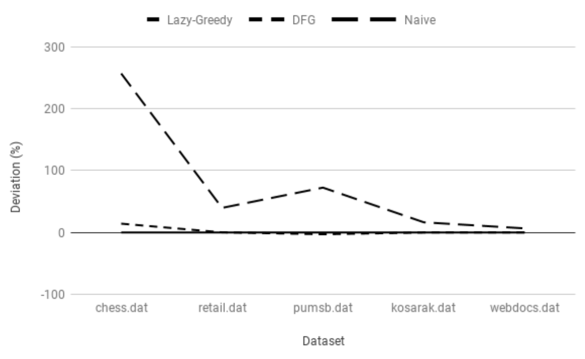
Table 3.3: RAM-based performance of offline algorithms on FIMI Datasets

3.6 Experimental Results of Semi-Streaming algorithms

Next we evaluate a group of semi-streaming algorithms whose empirical performance so far hasn't been investigated. In particular the single-pass Rosen [26], our Randomized-Rosen 3.3, the multi-pass Progressive Greedy



(a) Graphical plot of 3.3 for runtime in ms



(b) Graphical plot of 3.3 for deviation w.r.t. the solution of the conventional greedy

(PGreedy) [20] and our Largest Set Heuristic (LSH) 3.1. Note that all Rosen instances has been optimized with modification 3.2 for increased accuracy.

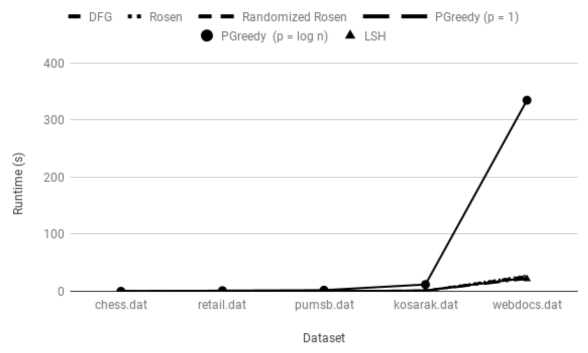
3.6.1 Implementation

The `sid` and `eff` data-structures were realized as an array of n elements however for real dataset it is unlikely to find the universe normalized s.t. it can be conveniently described by the range $[1, n]$. For this reason a script was written to after computing the universe, sort it and substitute elements in the universe and set system s.t. they uniformly increment within the range $[1, n]$. This normalized set-cover instance is written to file to be used by the MSC algorithms.

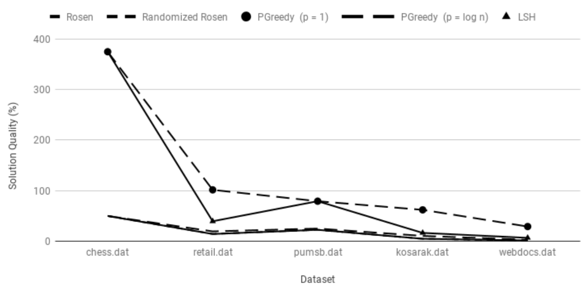
We utilize the sorting procedure of the C++ STL library, however this can be replaced by several other faster, memory efficient or parallelizable alternatives of the boost *Sort* library, if marginal performance is of importance. For the Randomized-Rosen algorithm to *uniformly* sample a subset of the set, we use the random sampling procedure of the C++ STL under the Mersenne Twister pseudo-random generator that satisfies several randomness tests [6] and was fed a random seed of sufficient entropy from linux's $\backslash dev \backslash random$.

3.6.2 Experiments

As streaming algorithms are characterized by the number of passes over the set system and each pass reads a set directly from disk, it is important to consider this additional time streaming from disk when making precise performance comparisons.



(a) Graphical plot of 3.5 for runtime in ms



(b) Graphical plot of 3.5 for deviation w.r.t. the solution of DFG

We observe that set streaming from disk is the dominating source of overhead regardless of streaming implementation, all algorithms therefore see an almost identical trend in scalability. As a quintessential example of the impact of set streaming, the Progressive-Greedy (PGreedy) for a polynomial number of passes terminated in

dataset	Algorithm
DFG	$\log n + 1$
Rosen	$8\sqrt{n}$
Randomized-Rosen	-
PGreedy (p = 1)	$2 \cdot \sqrt{n}$
PGreedy (p = $\log_2 n - 1$)	$2 \cdot \log n$

Table 3.4: Algorithm approximation guarantees

dataset	Algorithm	Time (s)	$\|Sol\|$
chess.dat	DFG (p = 1.05)	0.011	8
	Rosen	0.009	12
	Randomized-Rosen	0.008	12
	PGreedy (p = 1)	0.011	38
	PGreedy (p = $\log_2 n - 1$)	0.037	12
	LSH	0.008	38
retail.dat	DFG (p = 1.05)	0.179	5,125
	Rosen	0.134	5,854
	Randomized-Rosen	0.14	5,863
	PGreedy (p = 1)	0.163	10,334
	PGreedy (p = $\log_2 n - 1$)	1.192	-
	LSH	0.125	7,149
pumsb.dat	DFG (p = 1.05)	0.417	735
	Rosen	0.322	901
	Randomized-Rosen	0.304	901
	PGreedy (p = 1)	0.289	1,317
	PGreedy (p = $\log_2 n - 1$)	2.071	919
	LSH	0.272	1,317
kosarak.dat	DFG (p = 1.05)	1.361	17,738
	Rosen	1.187	18,579
	Randomized-Rosen	1.185	18,584
	PGreedy (p = 1)	1.388	28,747
	PGreedy (p = $\log_2 n - 1$)	11.953	19,606
	LSH	1.096	20,658
webdocs.dat	DFG (p = 1.05)	23.494	406,357
	Rosen	27.744	412,656
	Randomized-Rosen	26.68	412,656
	PGreedy (p = 1)	23.847	524,370
	PGreedy (p = $\log_2 n - 1$)	334.99	417,707
	LSH	21.677	433,407

Table 3.5: RAM-based Performance of semi-streaming algorithms on FIMI Datasets

almost polynomial times more time than when run for a single pass.

Turning to solution accuracy it is essential to keep in mind the approximation ratios of these semi-streaming algorithms when making a comparison against its solution and the chosen baseline, the solution of DFG. The Progressive Greedy algorithm didn't fair well when restricted to a single-pass (29% deviation for webdocs) although fairing much better when allowed a polynomial number of passes (2.7% deviation for webdocs) for a substantial increase in runtime, hindering it scalability. Remarkably Rosen with a similar $O(\sqrt{n})$ approximation factor consistently produced accurate solutions (1% deviation for webdocs) in spite of making a similar approximation guarantee. The Randomized-Rosen faired almost similarly with a marginal if not negligibly better runtime with its largest improvement of a measly 1 second for the webdocs dataset where as compare to smaller datasets Δ strays greatly from the average set size. Coming not too far behind, the Largest Set Heuristic as expected faired well, with only a 6% deviation from baseline with the fastest runtime of all mentioned MSC algorithms.

3.7 Experimental Results of Deterministic Sublinear-Space Streaming Algortithms

We now finally evaluate the performance of sublinear-space streaming algorithms, to recap these allow for more than semi-streaming space $\Omega(n \cdot \log n^{O(1)})$ but strictly less than linear space $o(mn)$. This allows for single [26] and multiple [20] pass approximations in the streaming model better than the proven lower bounds ($O(\sqrt{n})$ and

$(p+1)n^{1/p+1}$ respectively) placed on semi-streaming algorithms, but no better than those of offline algorithms $\ln n + 1$.

Evaluation of the experimental performance of this class of algorithms is important to answer whether we can in-practice achieve a significantly better solution or even runtime to that of semi-streaming algorithms forf ideally small additional increase in space.

3.7.1 Implementation

As previously mentioned, In 2016 Assadi et al. [13] resolved the space-approximation trade-off of single-pass streaming algorithms in showing any deterministic algorithm with α-approximation requires at least $\Omega(\frac{m \cdot n}{\alpha})$-space when $\alpha = o(\sqrt{n})$ if we allow for exponential runtime. For convenience we refer to this algorithm as *SP-Assadi*.

SP-Assadi was only meant to re-enforce their accompanying proof of tight space-approximation trade-off and not be computationally efficient, in particular computing an exact MSC solution is only feasible in exponential time by an NP-complete ILP. We can nonetheless substitute this exact solution with an approximation, courtesy of the DFG heuristic. If we're being precise, the approximation of greedy is actually $H(\Delta)$, the Δ'th-harmonic number i.e. the summation of the reciprocals of the first Δ natural numbers. The α-approximation is thereby replaced by an only marginally worse $p \cdot \log \max\{(\Delta \cdot \alpha), n\}$ approximation. We obtain a still logarithmic offline-like approximation, better than single-pass semi-streaming $O(\sqrt{n})$ approximation in sublinear space.

3.7.2 Experiments

Below is the results of running the SP-Assadi for α values $\ln n$, $\frac{\sqrt{n}}{\ln n}$ and $\sqrt{n}$ with linearly decreasing space requirement $O(\frac{m \cdot n}{\ln n})$, $O(m \cdot \sqrt{n} \cdot \ln n)$ and $O(m \cdot \sqrt{n})$ respectively.

dataset	alpha	approx	space	$\|Sol\|$	Time (ms)
chess.dat	$\ln n$	$p \cdot \ln \max\{(\Delta \cdot \ln n), n\} + 1$	$O(\frac{m \cdot n}{\ln n})$	20	6
	$\frac{\sqrt{n}}{\ln n}$	$p \cdot \ln \max\{(\Delta \cdot \frac{\sqrt{n}}{\ln n}), n\} + 1$	$O(m \cdot \sqrt{n} \cdot \ln n)$	12	13
	$\sqrt{n}$	$p \cdot \ln \max\{(\Delta \cdot \sqrt{n}, n\} + 1$	$O(m \cdot \sqrt{n})$	32	5
retail.dat	$\ln n$	$p \cdot \ln \max\{(\Delta \cdot \ln n), n\} + 1$	$O(\frac{m \cdot n}{\ln n})$	27720	179
	$\frac{\sqrt{n}}{\ln n}$	$p \cdot \ln \max\{(\Delta \cdot \frac{\sqrt{n}}{\ln n}), n\} + 1$	$O(m \cdot \sqrt{n} \cdot \ln n)$	34749	226
	$\sqrt{n}$	$p \cdot \ln \max\{(\Delta \cdot \sqrt{n}, n\} + 1$	$O(m \cdot \sqrt{n})$	85632	317
pumsb.dat	$\ln n$	$p \cdot \ln \max\{(\Delta \cdot \ln n), n\} + 1$	$O(\frac{m \cdot n}{\ln n})$	3395	250
	$\frac{\sqrt{n}}{\ln n}$	$p \cdot \ln \max\{(\Delta \cdot \frac{\sqrt{n}}{\ln n}), n\} + 1$	$O(m \cdot \sqrt{n} \cdot \ln n)$	2988	337
	$\sqrt{n}$	$p \cdot \ln \max\{(\Delta \cdot \sqrt{n}, n\} + 1$	$O(m \cdot \sqrt{n})$	12015	219
kosarak.dat	$\ln n$	$p \cdot \ln \max\{(\Delta \cdot \ln n), n\} + 1$	$O(\frac{m \cdot n}{\ln n})$	130870	1096
	$\frac{\sqrt{n}}{\ln n}$	$p \cdot \ln \max\{(\Delta \cdot \frac{\sqrt{n}}{\ln n}), n\} + 1$	$O(m \cdot \sqrt{n} \cdot \ln n)$	214738	1901
	$\sqrt{n}$	$p \cdot \ln \max\{(\Delta \cdot \sqrt{n}, n\} + 1$	$O(m \cdot \sqrt{n})$	810173	2093
webdocs.dat	$\ln n$	$p \cdot \ln \max\{(\Delta \cdot \ln n), n\} + 1$	$O(\frac{m \cdot n}{\ln n})$	1647570	49778
	$\frac{\sqrt{n}}{\ln n}$	$p \cdot \ln \max\{(\Delta \cdot \frac{\sqrt{n}}{\ln n}), n\} + 1$	$O(m \cdot \sqrt{n} \cdot \ln n)$	1692084	39502
	$\sqrt{n}$	$p \cdot \ln \max\{(\Delta \cdot \sqrt{n}, n\} + 1$	$O(m \cdot \sqrt{n})$	1693710	32589

Table 3.6: Performance of SP-Assadi on FIMI Datasets

Contrary to its better approximation, in-practice the algorithm performed significantly worse than Rosen [26]. Even more surprisingly, for larger data-sets, the solution quality becomes as bad a naively choosing all m sets.

3.8 Experimental Results of Randomized Sublinear-Space Streaming Algortithms

3.8.1 Set Sampling

We evaluate the algorithm for increasing set samplings $L = c\log n$, $L = 2\frac{\sqrt{n}}{\log n}$, $L = c\sqrt{n}$, $L = c\sqrt{n}\log n$ and $L = cn$ which have linearly decreasing space bounds O(mn), O($m\sqrt{n}\log^2 n$), O($m\sqrt{n}\log n$), O($m\sqrt{n}$) and O($m\log n$).

On almost all datasets a sampling of $L = 2\frac{\sqrt{n}}{\log n}$ sets, gave the best reduction in space for an almost identical approximation. On the largest webdocs dataset, a significant almost 4 times reduction in space was observed whereas on the smaller datasets this effect was less pronounced. Intuitively this can be explained by the larger average and maximum set size of the webdocs dataset, a set sampling of these larger sets is more likely to cover a large proportion of the universe giving this noticeable drop in size of the remaining set-system.

While $L = 2\frac{\sqrt{n}}{\log n}$ is an appropriate rule of thumb to obtain an offline-like approximation, to infact guarantee this we'd need to sample $O(kp)$ sets, however this introduces a dependency on k, a $O(1)$-approximation of $|OPT|$. We'll discuss the impact of this in the following randomization technique, element sampling.

dataset	Lazy-Greedy	$L = 2\log n$	$L = 2\frac{\sqrt{n}}{\log n}$	$L = 2\sqrt{n}$	$L = 2\sqrt{n}\log n$	$L = 2n$
pumsb.dat	757	758	757	772	1170	2650
retail.dat	5,113	5,130	5,131	5,212	6,600	19,548
kosarak.dat	17,747	17,753	17,751	17,943	20,736	56,879
webdocs.dat	406,400	406,407	406,451	408,021	443,075	-

Table 3.7: Approximation of the set-sampling technique on FIMI Datasets

dataset	Lazy-Greedy	$L = 2\log n$	$L = 2\frac{\sqrt{n}}{\log n}$	$L = 2\sqrt{n}$	$L = 2\sqrt{n}\log n$	$L = 2n$
pumsb.dat	221Mb	27Mb	22Mb	11Mb	1Mb	0.4Mb
retail.dat	55Mb	45Mb	47Mb	32Mb	7Mb	0.8Mb
kosarak.dat	489Mb	363Mb	331Mb	260Mb	54Mb	8Mb
webdocs.dat	18,303Mb	10,380Mb	5,105Mb	2,001Mb	766Mb	-

Table 3.8: Memory usage (Mb, Megabits) of the set-sampling technique on FIMI Datasets

3.8.2 Element Sampling

Implementation

The element sampling requires a $O(1)$-approximation of k and while the $O(L+p)$-approximation, $O(\frac{mn}{L})$-space set sampling technique provides a smooth approximation-space trade-off for larger samplings L, a 2-pass element sampling has a hard lower bound on its space complexity which it achieves when it precisely balances the size of the sampled and remaining element spaces. The number of elements sampled $\frac{pk\log m}{\varepsilon}$ has a direct translation on the number of remaining elements $\varepsilon \cdot n$ and while a carefully selected value of $\varepsilon = \sqrt{\frac{pk\log m}{n}}$ achieves this lower bound up-to a constant factor (i.e. $O(m \cdot \sqrt{pkn\log m})$), a worse approximation of k can have adversely impact this balance. If we say k' is a $O(1)$-approximation of k i.e. $k' = ak$. The element sampling procedure randomly samples $c(\sqrt{\frac{pk\log m}{\varepsilon}} - \sqrt{\frac{pk\log m}{\varepsilon/a}})$ more elements leaving a smaller proportion ε/a of the universe uncovered.

This approximation of k also has a direct implication on the success probability of element sampling. If we estimate $k < |OPT|$ two thing happen. Firstly, our $\sqrt{pnk\log m}$ threshold on the maximum size of the remaining element space is lower and secondly, we initially sample fewer elements leading to a larger remaining element space. Combined, this dramatically lower the probability of a successful element sampling. For the exact value of k or larger, its success probability as proven is high therefore we can obtain results of running the 2-pass element sampling for an 2-approximation of $|OPT|$ by running the algorithm under $\log n$ guesses of $k \in [1, n]$ in parallel where thread i uses $k = 2^i$ returning the solution of the lowest value of k for which the element sampling was successful. While this is feasible for smaller problems, it is a major limitation on the scalability of the technique for larger problem instances and since approximating the size of the $|OPT|$ is also NP-complete and can be approximated no better than $\log n + 1$ [36], we are limited to this approach.

Experiments

We implement the 2-pass, $2p$-approximation element sampling using the lazy-greedy with optimal $\log n + 1$ approximation, setting $c = 3$ and $p = \log n + 1$. In our first experiment we set $\varepsilon = \sqrt{\frac{pk \log m}{n}}$, the theoretical expression which optimizes the algorithms space-bound, by equally balancing the number of elements between the sampled and remaining element spaces. If we take a look at Tables 3.9 and 3.10, this setting quite significantly drops the space requirement with an if only minor increase in solution size, however the 2-pass set sampling achieved an even better space-bound with a better approximation for $L = 2\sqrt{n}$ sampled sets.

Algorithm	chess.dat	pumsb.dat	retail.dat	kosarak.dat	webdocs.dat
Lazy-Greedy	7	757	5113	17,747	406,407
2-pass Element Sampling	-	791	5400	18,305	410,503

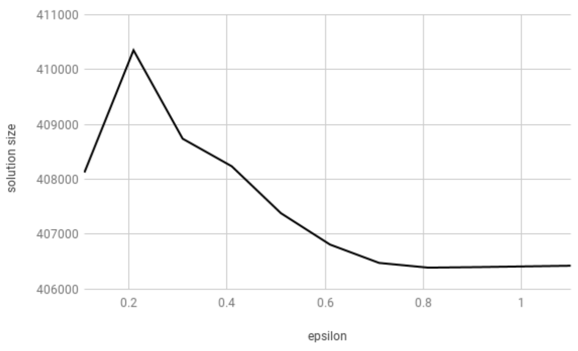
Table 3.9: Solution size of 2-pass Element Sampling for $\varepsilon = \sqrt{\frac{pk \log m}{n}}$

Algorithm	chess.dat	pumsb.dat	retail.dat	kosarak.dat	webdocs.dat
Lazy-Greedy	7Mb	221Mb	55Mb	489Mb	18,303Mb
2-pass Element Sampling	-	95Mb	31Mb	259Mb	3,502Mb

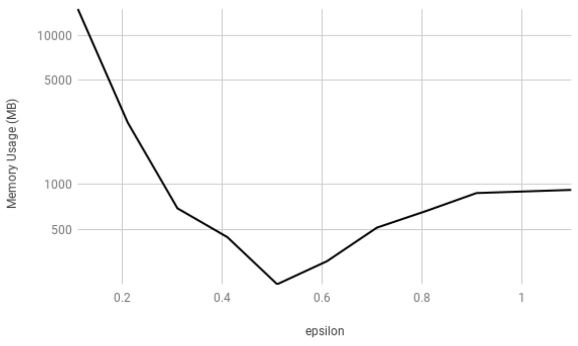
Table 3.10: Memory usage of 2-pass Element Sampling for $\varepsilon = \sqrt{\frac{pk \log m}{n}}$ in Megabits

Probing further we interpolate epsilon in the range $[0.01, 1.01]$ in 0.1 increments for the webdocs dataset noting its approximation 3.3a and memory usage 3.3b. If we imagine overlaying the curves, we notice somewhat opposing bell curves. At the extreme ends when the balance is skewed to heavily favour either the sampled or remaining element spaces, we obtain better approximations whereas closer to the middle a better balance and optimization of the algorithms space requirement is achieved.

More importantly the space curve show that a much better space-bound (of 215MB) can be achieved that what was seen for $\varepsilon = \sqrt{\frac{pk \log m}{n}}$. This highlights the impact of an over-estimation of the exact value of $|OPT|$.



(a) Trend in solution with epsilon



(b) Trend in memory usage (Mb) with epsilon

Figure 3.3: Performance metrics on LAWA datasets

3.8.3 Iterative Element Sampling

In Table 3.11 and Table 3.12 we detail the approximation and memory usage (in Megabits) of the MP-Assadi for 2 and a logarithmic number of passes for which the algorithm promises $O(m.\sqrt{n})$ and $O(m)$ space bounds and $3p$ and $((\log n + 1) \cdot p)$ approximations respectively.

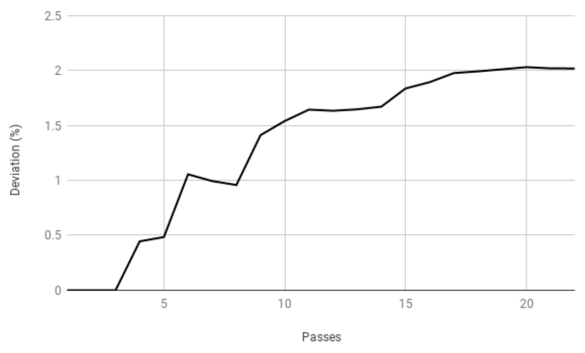
The algorithm see's no improvement in space for $\alpha = 2$ this was due to an over-sampling which could of been rectified by adjusting the sampling probability however in the idea of fidelity we evaluated the algorithm as is. For a logarithmic number of passes and more generally for α-passes the space-bound 3.4b exponentially decreased for an exponentially decreasing sampling probability with very minor, if negligible increase in approximation 3.4a

dataset	chess.dat	pumsb.dat	retail.dat	kosarak.dat	webdocs.dat
LGreedy	7	757	5,113	17,747	406,339
$\alpha = 2$	7	784	5,113	17,750	406,400
$\alpha = \log n$	7	805	5,624	18,765	414,577

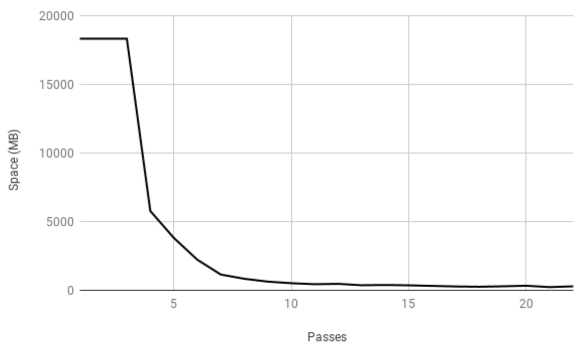
Table 3.11: Solution of MP-Assadi on FIMI Datasets

Algorithm	chess.dat	pumsb.dat	retail.dat	kosarak.dat	webdocs.dat
Lazy-Greedy	7Mb	221Mb	55Mb	489Mb	18,303Mb
$\alpha = 2$	7Mb	221Mb	55Mb	489Mb	18,303Mb
$\alpha = \log n$	7Mb	56Mb	11Mb	60Mb	293Mb

Table 3.12: Memory Usage (Mb, Megabits) of MP-Assadi on FIMI Datasets



(a) Pass-Approximation Tradeoff of MP-Assadi



(b) Pass-Space Tradeoff of MP-Assadi

3.9 A Fast Sublinear-Space Heuristic for MSC

Summarizing the viability of the detailed sublinear-space streaming techniques -

- The deterministic, single-pass SP-Assadi [13] produced solution of dismal quality.
- The 2-pass set sampling and element sampling techniques gave offline-like approximations at a compelling reduction in space. However we could not guarantee an offline-like p-approximation for set sampling or a successful element sampling with high probability without an $O(1)$-approximation of $|OPT|$. Additionally, randomization technique has the chance to unexpectedly fail, requiring multiple runs in the worst case. These are significant disadvantages of the techniques.
- The randomized, multiple pass, MP-Assadi [12] which iteratively leverages the element-sampling technique carries the same flaws and as we've noticed multiple passes over a set system is un-scalable in practice.

Since none of these heuristics are viable as big-data algorithms, i propose a template design for the construction of performant 2-pass deterministic sublinear-space heuristics that builds on the advantages of strong semi-streaming and offline solvers. We name this template *SLSC* (Sub-Linear Set-Cover) for convenience.

3.9.1 SLSC Design

The high level idea is as follows -

- **Step 1** involves performing a pre-processing over the SC instance $(\mathcal{U}, \mathcal{S})$ using a existing semi-streaming algorithm, returning a comparatively larger solution to that of offline heuristics.
- **Step 2** then makes a second-pass over the stream $\mathcal{S}$, storing only these sets in-memory as a reduced set-system named $\mathcal{S}'$.
- **Step 3** The produced MSC solution is obtained in using an efficient offline solver over the reduced SC instance $(\mathcal{U}, \mathcal{S}')$.

The sublinear $O(|\mathcal{S}'|.n) \leq O(m.n)$ space-bound is contingent on the size of $\mathcal{S}'$ and therefore the accuracy of the semi-streaming pre-processor. It is easy to see $|\mathcal{S}_\prime| \leq m$, at worst being m therefore operating in linear space $O(m.n)$ but frequently in-practice this is never the case, usually being significantly smaller than m.

The intuition of further refining the solution of the semi-streaming heuristic with an offline solver follows from the fact that multiple sets of $\mathcal{S}_\prime$ might cover the same element, this is true for all elements of the universe. These redundant sets allow for further optimization, if we can remove these in an efficient manner.

3.9.2 Experiments

We measure and compare the performance of concrete implementations of the SLSC design using a variety of semi-streaming and offline solvers against their standalone implementations.

dataset	SS	Offline	$\|Sol\|$	Time (ms)	RAM (Mb)
chess.dat	LSH	DFG (p=1.05)	16	0.016	0.08Mb
	LSH		37	0.008	-
	Rosen	DFG (p=1.05)	11	0.016	0.02Mb
	Rosen		12	0.008	-
		DFG (p=1.05)	8	0.01	-
retail.dat	LSH	DFG (p=1.05)	5371	0.262	10Mb
	LSH		7148	0.126	-
	Rosen	DFG (p=1.05)	5286	0.266	6Mb
	Rosen		6059	0.134	-
		DFG (p=1.05)	5125	0.163	-
pumsb.dat	LSH	DFG (p=1.05)	851	0.545	5Mb
	LSH		1317	0.271	-
	Rosen	DFG (p=1.05)	779	0.568	4Mb
	Rosen		917	0.297	-
		DFG (p=1.05)	735	0.382	-
kosarak.dat	LSH	DFG (p=1.05)	18542	2.189	53Mb
	LSH		20657	1.092	-
	Rosen	DFG (p=1.05)	18125	2.259	24Mb
	Rosen		18759	1.157	-
		DFG (p=1.05)	17738	1.306	-
webdocs.dat	LSH	DFG (p=1.05)	407066	43.993	8850Mb
	LSH		433406	21.529	-
	Rosen	DFG (p=1.05)	406778	47.849	8234Mb
	Rosen		416670	25.785	-
		DFG (p=1.05)	406357	23.194	-

Table 3.13: RAM-based experiments of SLSC algorithms on FIMI datasets

According to our results, in composing solvers with the SLSC design, the solution size of semi-streaming (LSH, Rosen) solvers drop significantly towards the best-case approximation of the offline DFG solver. For the case of the largest dataset webdocs.dat, the solution of LSH which was 6.24% worse than that of DFG becomes only 0.17% worse. Similarly, Rosen improves from 2.53% to 0.10%. Additionally using less space than the set sampling technique for $L = 2\frac{\sqrt{n}}{\log n}$ on all datasets except that of webdocs.

The larger improvement in solution quality for LSH is a consequence of two phenomena. Firstly, the solution is larger for its larger approximation factor and contain a larger number of redundant sets that may be pruned off. Secondly and more importantly, sets of this solution are larger finding more redundant set covering where a large number of elements of the universe are likely to be covered by more sets. This inspired me to make an interesting modification of Rosen, namely **Robust Rosen**.

3.9.3 Robust Rosen

The modification guarantees an identical approximation factor only guiding Rosen to make more robust set selections which facilitate a better pruning.

Consider a simplified instance where $\mathcal{U} = [1, 6]$, and $\mathcal{S} = \{S_1 = \{1, 2, 3, 4\}, S_2 = \{5, 6\}, S_3 = \{5, 6, 1\}\}$. As the Rosen algorithm processes the set -

- S_1, the effectivity of 1,2,3 and 4 becomes 4.
- S_2, the effectivity of element 5 and 6 becomes 2.
- S_3, nothing changes as element 1 is covered more effectively by S_1 and element 5 and 6 by set S_2 respectively.

S_3 contains a subset of equal effectivity $|5, 6| = 2$ to that of S_2, additionally containing the already covered element 1. This case where equivalently suitable choices are available, an informed decision can be made to choose sets that would consolidate and existing coverage. Specifically, when we are given the choice between two equally large (therefore equally effective) effective subsets choosing the subset belongs to the larger set that contains more already covered elements promotes this additional consolidation of our final covering.

Experiments

The webdocs.dat dataset for which previously the composition of Rosen and DFG (p=1.05) gave a 0.10% deviation is now only 0.02% worse.

Algorithm	chess.dat	retail.dat	pumsb.dat	kosarak.dat	webdocs.dat
Rosen-DFG	11	5286	779	18125	406778
RobustRosen-DFG	9	5202	779	17969	406444

Table 3.14: Improvement in Rosen-DFG solution quality

Chapter 4

Critical Evaluation

So far we've been been using frequent-itemset-mining (FIM) instances as valid un-weighted set-cover instances, while this is an acceptable method of assessing the chosen MSC algorithms, it isn't a realistic use-case or practical application of a MSC algorithm, mostly since a MSC on a FIM instance doesn't correspond to any meaningful metric. We therefore turn to solving an important combinatorial problem, the minimum dominating set on graph datasets several times greater to previously accessed FIM datasets, our largest being 851GB and a valid instance of a big-data dataset.

4.1 Dominating Set Problem

In graph theory, a minimum dominating set (MDS) for a graph $G = (V, E)$ is a subset D of V such that every vertex not in D is adjacent to at least one member of D and no smaller dominating set exists. For example consider a social graph where vertices correspond to individuals and edges to friendships. The MDS coincides to the group of individuals that knows everybody else on the network, in practice this is a salient metric, characteristic of the dataset. The minimum dominating set (MDS) too is one of Karps 21 NP-complete problems thereby also intractable in polynomial time.

For our purposes there exists a polynomial time reduction from the Dominating Set problem to the Set Cover problem that preserves its approximation ratio [10] i.e. a polynomial-time α-approximation algorithm for the minimum set cover can provide an α-approximation for a reduced MDS problem.

Reduction. Given a graph $G = (V, E)$ with $V = [1, n]$, construct a set cover instance (U, S) as follows: the universe U is V, and the family of subsets is $S = S_1, S_2, \ldots, S_n$ such that set S_v consists of the vertex v and its neighbourhood i.e. all vertices adjacent to v in G. If D is the MDS for G, then $C = \{S_v : v \in D\}$ is a feasible solution of the MSC, where $|C| = |D|$ conserving the approximation ratio between problem instances.

We leverage this reduction by writing a script to reformulate a graph dataset representing a MDS problem instance as a MSC, later written to disk to be fed as input to an existing MSC approximation algorithm. The particulars of how this script were written are quintessential examples of problems encountered when taming big data datasets. Ideally we'd like to read every edge, appending its destination vertex to the set identified by the source vertex, however for datasets several times larger than available RAM, this technique quickly runs out becoming severely I/O bound or in our case where the system didn't support a swap partition, the process is immediately killed. We could choose to stream over the dataset maintaining the description of only a selected subset of all sets. Given we compute Δ with an initial pass over the stream, we can guarantee that the constructed sets do not incur an out-of-memory exception, however this requires multiple passes over the entire dataset and as we've seen when evaluating semi-streaming algorithms this easily becomes unsalable.

We instead choose to iteratively memory-map 5GB chunks of the total dataset, computing partial sets for which edges appear in the chunk. Each partial set is identified by an positive integer identifier and written to disk in sorted order w.r.t. its identifier. A final merge step carefully reads from the multiple output files reconstructing the complete neighbourhood from the partial sets.

The following experiments unless otherwise stated were performed on the University of Bristol's high performance computing machine - Blue Crystal Phase 4.

System. A node supports a Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz running CentOS Linux release 7.3.1611 (Core) under no load. The system is equipped with 100GB of RAM. The program is compiled with g++ 7.3.0 and the -O3 flag.

4.2 SNAP

The Stanford Network Analysis Project (SNAP) hosts a collection of 50 large network datasets[7]. There largest in particular, the *com-Friendster* includes 65,608,366 nodes and 1,806,067,135 edges. From here we obtained a set of reasonably large network graphs 4.1 several being friendship social networks (ego-Facebook, soc-Pokec, com-LiveJournal and com-Orkut) and others being web-graphs (web-NotreDame and web-Google).

dataset	Size (GB)	n	m	M (edges)	avg $\|S_i\|$	max $\|S_i\|$	$\|lowerbound\|$
ego-Facebook	1.0M	4,037	4,030	173,052	42	996	8
web-NotreDame	9.5M	316,716	137,337	1,454,855	10	3,176	1594
web-Google	34M	707,656	736,759	5,021,394	6	454	28108
soc-Pokec	201M	1,514,551	1,429,929	30,153,278	21	8,661	5591
patents.dat	128MB	3,258,983	2,089,345	16,518,948	7	770	89,796
com-LiveJournal	473M	3,974,288	3,974,286	68,038,342	17	14,713	6,317
com-Orkut	1.7G	3,070,923	3,070,920	229,939,943	74	33,025	373

Table 4.1: SNAP Datasets

4.2.1 Results

From previous experiments we identified several weak algorithms -

- The progressive greedy which for a single pass was fairly in-accurate and un-scalable when extrapolated to logarithmically many passes with a still lower accuracy than Rosen. Based on this in-scalability of logarithmic pass streaming algorithms the previously proposed $O(\log n)$-pass, $O(\log n)$-approximation semi-streaming algorithm of Saha & Getoor [43] too can be considered a weak candidate.
- The Randomized-Rosen didn't provide enough of an runtime-approximation trade-off to justify itself against the LSH, which with the least overhead gave an only 6% deviation.
- We've discussed the deficiencies of the sub-linear streaming algorithms previously 3.9.

For these reasons the mentioned algorithms were ignored from further experiments.

Table 4.2 and Figure 4.1 show the performance metrics of adept offline (DFG), semi-streaming (Rosen and LSH) and sublinear-space streaming (Rosen-DFG and LSH-DFG) set cover algorithms on these datasets. We additionally record the memory usage by finding the median of the largest peaks in heap memory consumption using Valgrind's *massif* heap profiler [8].

Of the semi-streaming algorithms Rosen and LSH show a fluctuating deviation between 2%-32% and 11%-56% from the baseline, respectively. While these are significant, they are still significantly better than what their approximation guarantee dictates, most importantly taking space only a fraction of the size of the problem instance. Most algorithms scaled linearly with Rosen exhibiting a slightly super-linear trajectory because of the $O(\Sigma_i |S_i| \cdot \log |S_i|)$ super-linear time required in sorting sets, exacerbated and showing its worst-performance on the larger *com-LiveJournal* and *com-Orkut* datasets with much larger average and maximum set sizes. In these cases and more generally LSH consistently terminated earliest with greatest scalability of the group.

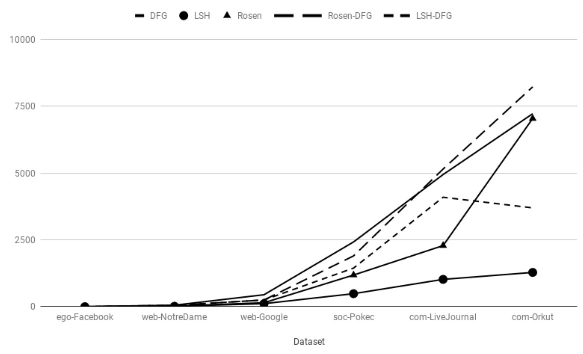
The sublinear-space algorithms Rosen-DFG and LSH-DFG fluctuated between 0.06%-12% and 0.3%-20% from baseline respectively, a big improvement over semi-streaming algorithms in about half the space of an offline algorithm.

Now while our sublinear-space streaming algorithms take almost twice as long as the stream is read through twice 4.2, our graphical plot 4.1a disregards the overhead of this second pass. We do this to illustrate a trend that becomes important when considering which semi-streaming pre-processing might be more beneficial for performance. In more detail, because the solution size of LSH is somewhat bigger and individual sets are by design bigger, the intermediate reduced set-system of LSH-DFG takes more space however showing better scalability. When set sizes are larger, reducing the potential size of the solution we see a dip in runtime between *com-LiveJournal* and *com-Orkut* datasets in spite of com-Orkut being 4 times larger. The initial LSH hence behaves as a fast pre-processor of the set system dramatically reducing drastically the overhead of the much slower DFG.

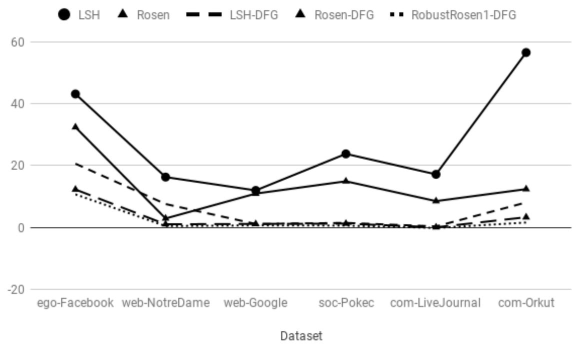
A practical takeaway from the mentioned observations is that for graphs with a large average in-degree or neighbourhoods which translates to large set sizes or in set-cover terms $M >> m$, a LSH-DFG approach is fastest. Similarly Rosen and Rosen-DFG should be avoided in these cases. When this isn't the case LSH-DFG, Rosen or LSH perform best w.r.t. accuracy at increasing degrees of trade-off in accuracy for speed. When memory is of more importance, semi-streaming algorithms are by a larger margin superior.

dataset	Algorithm	Time (ms)	$\|Sol\|$	RAM
ego-Facebook	DFG	15	50	1.3MB
	LSH	13	88	0.06MB
	Rosen	16	74	0.06MB
	Randomized-Rosen	16	74	0.06MB
	LSH-DFG	27	63	0.11MB
	Rosen-DFG	26	57	0.06MB
	RobustRosen-DFG	28	56	0.07MB
web-NotreDame	DFG	272	38,152	12.4MB
	LSH	231	45,582	4.9MG
	Rosen	239	39,300	4.9MB
	LSH-DFG	481	41,283	6.5MB
	Rosen-DFG	484	38,556	4.4MB
	RobustRosen-DFG	488	38,328	4.5MB
web-Google	DFG	1,157	149,731	45MB
	LSH	824	170,122	11MB
	Rosen	858	168,154	11MB
	LSH-DFG	1,672	151,469	15MB
	Rosen-DFG	1,671	151,469	13MB
	RobustRosen-DFG	1,763	150,762	13MB
soc-Pokec	DFG	5,505	243,063	246MB
	LSH	3,569	319,025	23MB
	Rosen	4,266	285,759	23MB
	LSH-DFG	7,609	246,738	121MB
	Rosen-DFG	8,064	246,148	96MB
	RobustRosen-DFG	8,295	244,562	95MB
com-LiveJournal	DFG	12,401	795,098	563MB
	LSH	8,480	960,146	62MB
	Rosen	9,744	869,247	62MB
	LSH-DFG	19,009	797,948	330MB
	Rosen-DFG	20,066	795,654	276MB
	RobustRosen-DFG	20,800	793,840	271MB
com-Orkut	DFG	26,881	114,297	1.8GB
	LSH	20,945	263,315	47MB
	Rosen	26,701	130,481	47MB
	LSH-DFG	43,022	124,363	559MB
	Rosen-DFG	47,548	118,230	185MB
	RobustRosen-DFG	44,619	116,161	193MB

Table 4.2: RAM-based performance metrics on SNAP datasets



(a) Graphical plot of 4.2 for runtime in ms



(b) Graphical plot of 4.2 for deviation (vertical axis in as percentage) w.r.t. the solution of DFG

Figure 4.1: Performance metrics on SNAP datasets

4.3 LAWA

The aforementioned datasets are not valid instances of big data. Big data instances are so large that they don't fit in main memory, usually ranging from several million gigabytes to a few petabytes. To test the scalability of MSC heuristics, performance measures for these much larger datasets are required. In supporting this agenda, the study uses a diverse collection of instances derived from various sources of popular graphs made publicly available by the Laboratory for Web Algorithmics (LAWA) [19, 18] described in Table 4.3.

dataset	Size (GB)	n	m	M(edges)	avg $\|S_i\|$	max $\|S_i\|$	$\|lowerbound\|$
twitter-2010.dat	13G	35,689,148	40,103,281	1,468,365,180	36	2,997,469	19
sk-2005.dat	17G	50,635,753	43,732,623	1,949,412,600	44	12,870	20,263
friendster.dat	16G	64,961,031	37,551,360	1,806,067,135	48	3,615	37,598
uk-2007-05.dat	33G	102,304,612	92,946,883	3,738,733,646	40	15,402	23,453
uk-union.dat	49G	126,454,248	121,503,287	5,507,679,819	45	22,429	40,552

Table 4.3: LAWA Datasets

Table 4.5 and Figure 4.2 present the performance statistics for these datasets.

4.3.1 Results

Not many new observations can be made w.r.t. the performance of the different algorithms. The accuracy of semi-streaming algorithms however tend to see less variance with Rosen and LSH seeing a 6%-11% and 9%-15% deviation from baseline respectively and Rosen-DFG and LSH-DFG with a less than 1% deviation on all instances. If anything, these within 4% worse approximations of LHS compared to Rosen only endorse its use for much larger datasets for which we can hope to expect an almost negligible difference in accuracy.

4.4 Disk Experiments

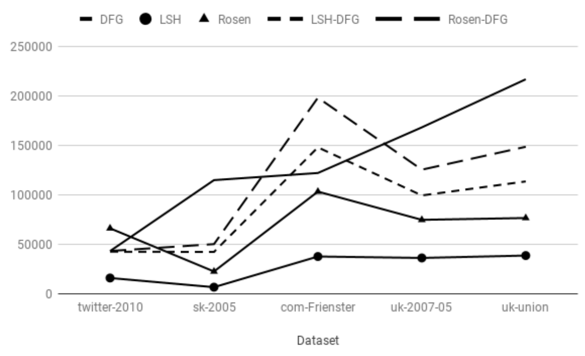
Previously conducted experiments are illustrative of performance when adequate RAM is available to contain the read-in set-cover problem instance and additionally required algorithmic data-structures of set-cover heuristics. In the following we explore two cases. Firstly, the trend in performance when only the problem instance is unable to fit into memory. Secondly, when the both the problem instance *and* algorithmic data-structures cannot fit into memory. To realize this scenario we artificially restricted the available RAM and disk buffer to 150MBs and 300MBs respectively, results are tabulated in 4.6. 4.7 and graphically plotted in 4.3a, 4.3b.

It is a common occurrence that on massive instances, even powerful machines are unable to contain the set system in RAM. Scalability tests in this environment is a must. Additionally the super RAM-restricted setting

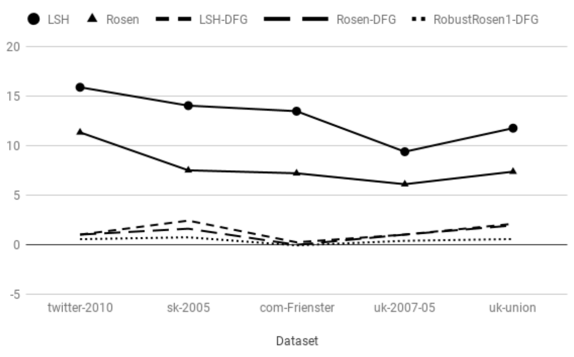
Table 4.4: FIMI Dataset characteristics

dataset	Algorithm	Time (s)	$\|Sol\|$	RAM
twitter-2010.dat	DFG	169	3,845,395	11.7GB
	LSH	141	4,571,771	557MB
	Rosen	192	4,337,114	557MB
	LSH-DFG	294	3,884,954	7.8GB
	Rosen-DFG	294	3,884,954	6.4GB
	RobustRosen1-DFG	340	3,866,863	6.5GB
	RobustRosen2-DFG	351	3,866,406	-
sk-2005.dat	DFG	281	8,016,792	15.5GB
	LSH	173	9,325,772	791MB
	Rosen	189	8,667,590	791MB
	LSH-DFG	376	8,216,250	4GB
	Rosen-DFG	384	8,148,252	3.4GB
	RobustRosen1-DFG	384	8,077,135	3.4GB
	RobustRosen2-DFG	404	8,093,661	-
friendster.dat	DFG	291	10,879,088	14.4GB
	LSH	207	12,573,198	1GB
	Rosen	273	11,724,263	1GB
	LSH-DFG	487	10,905,649	10.6GB
	Rosen-DFG	537	10,878,076	9.3GB
	RobustRosen1-DFG	560	10,871,472	9.2GB
	RobustRosen2-DFG	535	10,871,351	-
uk-2007-05.dat	DFG	548	16,970,489	30GB
	LSH	416	18,728,309	1.6GB
	Rosen	454	18,073,818	1.6GB
	LSH-DFG	859	17,145,594	7.2GB
	Rosen-DFG	885	17,145,594	6.3GB
	RobustRosen1-DFG	858	17,037,606	6.2GB
	RobustRosen2-DFG	868	17,069,026	-
uk-union.dat	DFG	697	18,396,629	44GB
	LSH	519	20,848,668	1.9GB
	Rosen	557	19,860,545	1.9GB
	LSH-DFG	1,074	18,792,123	8.7GB
	Rosen-DFG	1,109	18,761,661	7.4GB
	RobustRosen1-DFG	1,101	18,501,836	7.3GB
	RobustRosen2-DFG	1099	18,547,326	-

Table 4.5: RAM-based performance metrics on LAWA datasets



(a) Graphical plot of 4.5 for runtime in ms



(b) Graphical plot of 4.5 for deviation (vertical axis in as percentage) w.r.t. the solution of DFG

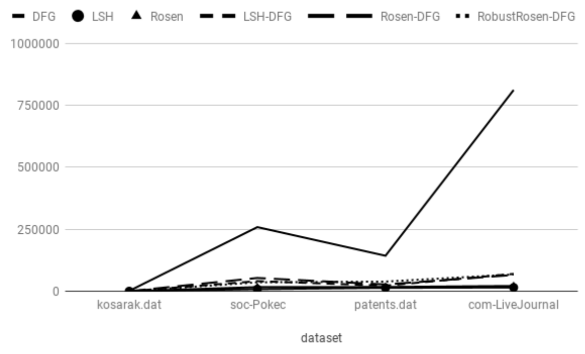
Figure 4.2: Performance metrics on LAWA datasets

dataset	DFG	LSH	Rosen	LSH-DFG	Rosen-DFG	RobustRosen-DFG
kosarak.dat	1,399	201	169	80	202	153
soc-Pokec	258,888	8,537	16,756	40,760	53,574	36,505
patents.dat	143,300	13,698	17,246	22,556	27,391	39,051
com-LiveJournal	812,504	15,009	21,337	70,082	65,696	67,797

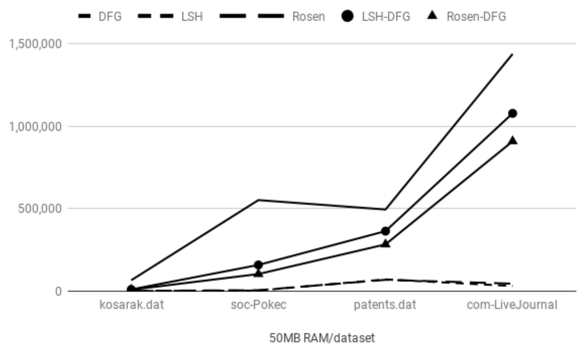
Table 4.6: Performance (ms) when limited to 150MB of RAM

dataset	DFG	LSH	Rosen	RobustRosen	LSH-DFG	Rosen-DFG
kosarak.dat	66,593	1,506	2,349	8,715	7,165	8,286
soc-Pokec	551,836	4,209	6,116	117,507	62,449	85,669
patents.dat	494,882	71,613	69,166	357,196	276,928	1,056,758
com-LiveJournal	1,437,165	31,736	45,740	1,023,119	853,910	-

Table 4.7: Performance (ms) when limited to 300MB of RAM



(a) Graphical plot of Table 4.6, vertical axis in ms



(b) Graphical plot of Table 4.7, vertical axis in ms

where sufficient RAM isn't available to contain the data-structures of the MSC algorithm models embedded devices and computing nodes of low power clusters[11], providing an understanding of the performance of these algorithms on such devices.

System. Experiments were conducted on a Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz running Ubuntu 18.04.1 LTS under light load. The system is equipped with SSD of which 22GB is allocated for swap-space and a magnetic disk on which the datasets reside and are read from. The program is compiled with g++ 7.3.0 and the -O3 flag.

In either experiment it is the semi-streaming algorithms that are least impacted in constrained memory environments as can be seen in the shallow steepness of their performance curves. This is a consequence of the restrictive nature of the semi-streaming model. Firstly, streaming the set system maximizes the number of sequential memory-accesses, exploiting the spatial locality and cache pre-fetching mechanisms of modern machines. Secondly, a low sublinear space-bound dissuades algorithms becoming I/O bound, a programmatic state due to frequent, unpredictable random-accesses to memory. We see this second property if we take a look at previous experiments on SNAP 4.2 and LAWA datasets 4.5 which shows that the size of the semi-streaming data-structures is in-practice several magnitudes smaller than the size of the set system.

In these RAM-restricted experiments, memory is paged in and out of swap-partition, a dedicated region of disk (usually a fast SSD drive) that behaves as an additional lower level of the memory hierarchy. In more detail, when processes of the system exhaust available RAM, pages of disk are swapped between RAM and this swap-partition. When this happens, the program becomes severely IO-bound, as the program continuous to make expensive read-/write disk operations even though the program doesn't necessarily need access to any file on disk. For this reason, as semi-streaming algorithms can maintain a much larger proportion of their data-structures in RAM they incur a smaller proportion of expensive page-faults operations, primarily operating in RAM, they show better scalability.

In contrast, the offline DFG begins to slow-down considerably and whereas in previous RAM-based experiments, com-LiveJournal showed a less than a 2 times larger runtime when compared to Rosen or LSH, under a RAM-restricted environment, DFG begins to take 38 times and 54 times longer than Rosen and LSH respectively. The takeaway being the magnitude of the benefit reaped by algorithms of the semi-streaming algorithm.

The sublinear-space streaming LSH-DFG and Rosen-DFG algorithms walk a tight line between being or not being scalable. When sufficient RAM is available to contain the reduced set system of the initial pre-processing, the algorithm runs entirely in-RAM, evading the performance penalty of paging memory in and out of the swap-partition otherwise it sees a similar yet less pronounced fate of all offline algorithms.

4.5 Parallelising LSH

As mentioned a primary advantage of LSH is that is can be parallelised to threaded or distributed computing environments.

Algorithm 12 Parallel LSH

```
1:  procedure PLSH($\mathcal{U}, \mathcal{S}, threads$)
2:      for all $thread = \{0, \dots, threads\}$ do
3:          for all $t = \{(\frac{m}{threads} \cdot thread, \dots, \frac{m}{threads} \cdot (thread + 1)\}$ do
4:              Read set $s_t \in \mathcal{S}$ from the stream
5:              for all $v \in s_t$ do
6:                  if $|s_t| > \text{eff}_{thread}(v)$ then
7:                      $\text{sid}_{thread}(v) \leftarrow t$
8:                      $\text{eff}_{thread}(v) \leftarrow |s_t|$
9:                  end if
10:             end for
11:         end for
12:     end for
13:     for all $thread = \{0, \dots, threads\}$ do
14:         for all $v = \{(\frac{n}{threads} \cdot thread, \dots, \frac{n}{threads} \cdot (thread + 1)\}$ do
15:             $thread' = \text{argmax}_{thread}(\text{eff}_{thread}(v))$
16:             $\text{sid}_0(v) \leftarrow \text{sid}_{thread'}(v)$
17:         end for
18:     end for
19: end procedure
```

Such a modification by its nature is unsuprisingly simple 12. Each thread is made to process over its portion of the stream dictated by its thread id, only updating its separate copy of the `sid` and `eff` data-structures. A additional aggregation step is included to merge the individual copies, selecting the first largest set to cover every vertex. This step too can be parallelised by allowing each thread to process over a restricted portion of the element space $\mathcal{U}$.

4.5.1 Performance

In 2012 Blelloch et al. [16] presented their work in parallelised I/O efficient set-covering algorithms, effectively tying existing working in parallel [15] and I/O efficient [22] set-cover algorithms together. We assess our optimized multi-core LSH under a set of evaluation criteria they describe which adequately assess such parallelised algorithms -

- **Solutions Quality.** The parallel algorithm should deliver solutions with no significant loss in quality when compared to the sequential counterpart
- **Parallel Overhead.** The parallel algorithm running on a single core should not take much longer than its sequential counterpart, showing empirically that it is work efficient
- **Parallel Speedup.** The parallel algorithm should achieve good speedup2, indicating that the algorithm can successfully take advantage of parallelism

It is easy to see that a parallelised implementation couldn't have any impact on its approximation factor or even the final solution as even for equally large sets that cover an element, LSH chooses the largest. This is true even for its parallelised equivalent, the final solution therefore isn't stochastic i.e. it doesn't see any difference between invocations.

We can see that the parallel overhead of the additional aggregation step is a negligible portion of the overall runtime 4.9 and when P-LHS was limited to a single thread 4.8 it's run didn't sway significantly away from it's serialized implementation 4.5. Its scalability is decent with a 2.5x speed up for the 18GB *twitter-2010.dat* dataset and a larger 7.5x speed up for the largest *uk-union.dat* dataset. This can be simply explained by noticing that each thread scans over its smaller portion of the input from disk, ideally scaling the processing time linearly in the thread count. More subtly and in more detail this is a consequence of the disk-read pre-fetching mechanism done separately for every thread in parallel, granting the I/O scheduler more disk operations to figure out the fastest order to satisfy the multiple concurrent IO read operations placing larger demand and better utilizing the bandwidth of the disk. We've optimized the set-streaming step, which remembering back was the primary overhead of all streaming-based algorithms, this is especially true for RAM-based programs where this is only time disk is accessed.

To further understand its scalability we study the trajectory of its runtime w.r.t the number of threads used as it is manipulated between 1 and 28 for the *com-Orkut* dataset. Figure 4.10 shows indeed that its performance scales well till around 15 threads at which point additional threads make almost no difference. This limiting non-linear scalability is a consequence of saturating the disk's bandwidth at which point the physical hardware of the disk cannot transfer data any faster, we can however replace the hard drive with a solid state drive which allows for a larger maximum bandwidth achieving better scalability, again only to a certain (although much larger) number of threads.

Algorithm	twitter-2010.dat	sk-2005.dat	friendster.dat	uk-2007-05.dat	uk-union.dat
P-LHS$_1$	154	175	208	420	533
P-LHS$_{28}$	62	19	36	41	72

Table 4.8: Runtime of RAM-based run of Parallel LSH for 1 and 28 threads threads. All time measurements are in **seconds**.

Algorithm	Processing	Aggregating
P-LHS$_1$	99.7%	0.2%
P-LHS$_{28}$	99.0% (8.15x)	0.9% (1.69x)

Table 4.9: Proportion of runtime spend in its individual components and their speed ups for the **com-Orkut** dataset.

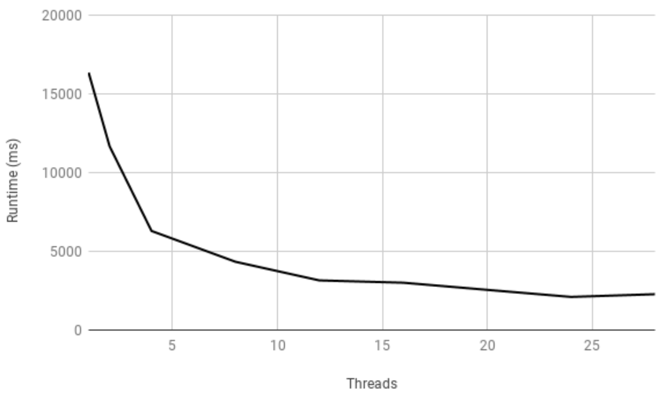


Table 4.10: Trend in runtime for Parallel LSH for the **com-Orkut** dataset as the thread count varies between 1 and 28.

4.5.2 Massive Instances

Our last dataset is a massive 851GB graph instance instance courtesy of the BMSB web-crawler [17] and publically available on the Laboratory for Web Algorithmics (LAWA) [4]. This instance was too large to be run by our offline or sublinear-space streaming algorithms, the semi-streaming algorithms however consumed sufficiently low memory to be run by our 400GB RAM computing environment. Without a baseline, we can't critically evaluate the quality of their approximations.

While Rosen takes about three and a half hours longer than LSH and considerably longer if LSH is parallelized across 28 threads, its solution is adequately better than that of LSH's to justify its usage in an official capacity.

dataset	Size	n	m	M (edges)	avg $\|S_i\|$	max $\|S_i\|$
eu-2015	851GB	1,068,549,225	960,864,336	91,645,133,736	95	35,340

Algorithm	Time (hours)	$\|Sol\|$
LSH	6.9	152,053,216
Rosen	10.4	136,018,449
P-LSH$_{28}$	-	152,053,216

Chapter 5

Conclusion

5.1 Project Success

We determine whether our findings and contributions satisfy and are in-line with our original project aims -

Research, understand and evaluate modern literature on the class of semi-streaming and sublinear-space techniques w.r.t their viability as big-data minimum set-cover algorithms. I feel this has been achieved. We began with a series of several small, quick experiments where we felt out the competence of most, if not all fundamental techniques to streaming-based MSC algorithms. We pruned weak candidate of which there were many, critically evaluating the remainder on their scalability to massive graph instances in RAM-based and Disk-based experiments. Ultimately concluding that only a semi-streaming approach can adequately satisfy all our requirements.

Discover algorithms, possibly of these classes that satisfy this criteria while consistently delivering a good approximation on realistic datasets. This has been achieved. Our semi-streaming algorithm, LSH is the quintessential example of an ideal computationally efficient algorithm, scaling linearly (not super-linearly like Rosen) with the size of the problem. It also being parallelizable is a game changer, we can imagine such an algorithm taming much larger datasets when run across an entire HPC cluster. Our sublinear-space streaming set-cover template, SLSC like all its competitors takes only a single additional pass but unlike them make no stretches in its assumptions, being attractive from a practical standpoint.

5.2 Findings

Here we summaries our findings throughout the course of the project -

Discrepancies between theory and practice.
1) In accessing several algorithms it was a surprising realization that many techniques though detailed in theory and mathematically sound, failed to follow through on its guarantees in-practice. Even on large problems the approximation factor of an algorithm didn't have an overwhelming bearing on its solution. The Progressive Greedy (PGreedy) and LSH were quintessential examples of this, with PGreedy (resp. LSH) behaving worse (resp. better) than its $2 \cdot \sqrt{n}$ (resp. $O(n/2)$) approximation factor.
2) The set sampling and element sampling techniques have a dependency on k, a $O(1)$-approximation of $|OPT|$, theoretically in-feasible by any polynomial time approximation scheme and only circumvented by running the algorithm a logarithmic number of times. While the algorithm still runs in polynomial time, the implications of such an assumption are a stretch in a practical setting.

Accuracy of non-greedy and randomization techniques.

1) Our research hypothesis on the effectivity of non-greedy techniques of the streaming model was validated from observations on the experimental performance of semi-streaming algorithms on large to massive datasets where we noticed that the non-greedy Rosen and LSH showed deviations at worst 10% and 15% from the optimal solution respectively.
2) set sampling and element sampling did well to give offline-like approximations for probability-based techniques.

Streaming is sometimes expensive.
While semi-streaming is efficient in space and scalable when restricted to one or two passes. The majority of the overhead of any streaming algorithm is in the set streaming, it wouldn't be a stretch to say that such an algorithm can be thought of operating entire in disk i.e. restricted to zero bytes of RAM. for this reason streaming algorithms of many more passes, intuitively perform even worse than an offline algorithm restricted to a few hundred megabytes of RAM, limited by the unrecoverable overhead of reading the dataset several times from disk.

Practical Takeaways
Practical takeaways from this work is that when sufficient RAM is available to hold the entire problem instance, an I/O efficient offline heuristic like DFG [22] or Manis [16] thought not as I/O efficient as a semi-streaming algorithm terminated in reasonable time when executing within memory, not triggering any very expensive page-faults. When memory is an issue but an high accuracy is desired, SLSC allows for an almost ideal approximation in roughly half the space, ironically taking twice as long but preventing an I/O bound programmatic state which comes with it a much larger than two times speed penalty. On low memory nodes or ”massive” datasets, there is no choice but to choose a marginally in-accurate semi-streaming algorithm if the instance is to feasibly terminate.

5.3 Future Work

5.3.1 Commercial Value

There are a number of settings where the fundamental problem could be formulated as a streaming set-cover instance. In the contextual background we listed a number of such applications 1.6.3 from less important tooling such as a web-crawler [43] or blog-watch application [44] to fundamental optimization problems (shift scheduling, future planning etc.) at the center of the functioning of enterprise algorithms and applications to which proficient algorithms can being a reduction in operating costs or value in a businesses product over its competitors. An initially considered, eventually unexplored direction would have been the adaptation of recently publish and newly discovered streaming techniques to build such an commercial application or algorithm.

5.3.2 Different Graph Problems

A vast number of graph problems have been proven tractable in the data-stream [14] and semi-streaming [28] models. We could see the streaming techniques explored in this paper being adapted to potentially solve other graph problems or more likely, construct a more scalable, efficient streaming algorithm. For example an unpublished work of my supervisor leverages the iterative element sampling technique of Assadi [12] to solve the MDS problem in the edge-streaming model.

5.3.3 Different Computational Models

A dynamic graph is defined by a sequence of insertions or deletions of edges or vertices. The goal of a dynamic graph algorithm is to *quickly* correct its solution after such a dynamic change, rather recomputing it from scratch [30]. While the data-stream, semi-streaming and sublinear-space streaming models considers graphs defined by a sequence of edge or set (neighborhood) insertions, this omits deletions, because of this it is sometimes referred to as the insert-only streaming model. Our work in such streaming techniques for the MSC can be adapted to hold for dynamic streams and more generally for other graph problems as has already been done [37]. In a previous publication [32], the authors made a different interpretation of the dynamic set-cover in the offline model where the set of elements to be covered at each timestep changed and the algorithm was constrained to a limited number of changes. We could explore such an interpretation in the streaming model, again adapting our techniques.

Bibliography

[1] Boost c++ libraries. [Online; accessed 29-March-2019].

[2] Catch2. [Online; accessed 4-May-2019].

[3] Frequent itemset mining dataset repository. [Online; accessed 28-March-2019].

[4] Laboratory for web algorithmics. [Online; accessed 8-May-2019].

[5] The linux documentation project - linux page cache. [Online; accessed 4-May-2019].

[6] Robert g. brown's general tools page. [Online; accessed 5-May-2019].

[7] Stanford large network dataset collection. [Online; accessed 10-April-2019].

[8] Valgrind. [Online; accessed 5-April-2019].

[9] Charu C Aggarwal. *Data streams: models and algorithms*, volume 31. Springer Science & Business Media, 2007.

[10] Jochen Alber, Michael R Fellows, and Rolf Niedermeier. Polynomial-time data reduction for dominating set. *Journal of the ACM (JACM)*, 51(3):363–384, 2004.

[11] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 1–14. ACM, 2009.

[12] Sepehr Assadi. Tight space-approximation tradeoff for the multi-pass streaming set cover problem. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 321–335. ACM, 2017.

[13] Sepehr Assadi, Sanjeev Khanna, and Yang Li. Tight bounds for single-pass streaming complexity of the set cover problem. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 698–711. ACM, 2016.

[14] Ziv Bar-Yossef, Ravi Kumar, and D Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 623–632. Society for Industrial and Applied Mathematics, 2002.

[15] Guy E Blelloch, Richard Peng, and Kanat Tangwongsan. Linear-work greedy parallel approximate set cover and variants. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 23–32. ACM, 2011.

[16] Guy E Blelloch, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Parallel and i/o efficient set covering algorithms. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 82–90. ACM, 2012.

[17] Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. BUbiNG: Massive crawling for the masses. In *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web*, pages 227–228. International World Wide Web Conferences Steering Committee, 2014.

[18] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[19] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[20] Amit Chakrabarti and Anthony Wirth. Incidence geometries and the pass complexity of semi-streaming set cover. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 1365–1373. SIAM, 2016.

[21] Flavio Chierichetti, Ravi Kumar, and Andrew Tomkins. Max-cover in map-reduce. In *Proceedings of the 19th international conference on World wide web*, pages 231–240. ACM, 2010.

[22] Graham Cormode, Howard Karloff, and Anthony Wirth. Set cover algorithms for very large datasets. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, pages 479–488. ACM, 2010.

[23] Steven Cosares, David N Deutsch, Iraj Saniee, and Ondria J Wasem. Sonet toolkit: A decision support system for designing robust and cost-effective fiber-optic networks. *Interfaces*, 25(1):20–40, 1995.

[24] George B Dantzig. Letter to the editora comment on edie's traffic delays at toll booths. *Journal of the Operations Research Society of America*, 2(3):339–341, 1954.

[25] Erik D Demaine, Piotr Indyk, Sepideh Mahabadi, and Ali Vakilian. On streaming and communication complexity of the set cover problem. In *International Symposium on Distributed Computing*, pages 484–498. Springer, 2014.

[26] Yuval Emek and Adi Rosén. Semi-streaming set cover. *ACM Transactions on Algorithms (TALG)*, 13(1):6, 2016.

[27] Uriel Feige. A threshold of ln n for approximating set cover. *Journal of the ACM (JACM)*, 45(4):634–652, 1998.

[28] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. *Theoretical Computer Science*, 348(2-3):207–216, 2005.

[29] Michel X Goemans, Andrew V Goldberg, Serge A Plotkin, David B Shmoys, Eva Tardos, and David P Williamson. Improved approximation algorithms for network design problems. In *SODA*, volume 94, pages 223–232, 1994.

[30] Jonathan L Gross and Jay Yellen. *Handbook of graph theory*. CRC press, 2004.

[31] Tal Grossman and Avishai Wool. Computational experience with approximation algorithms for the set covering problem. *European Journal of Operational Research*, 101(1):81–92, 1997.

[32] Anupam Gupta, Ravishankar Krishnaswamy, Amit Kumar, and Debmalya Panigrahi. Online and dynamic algorithms for set cover. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 537–550. ACM, 2017.

[33] Sariel Har-Peled, Piotr Indyk, Sepideh Mahabadi, and Ali Vakilian. Towards tight bounds for the streaming set cover problem. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 371–383. ACM, 2016.

[34] Monika Rauch Henzinger, Prabhakar Raghavan, and Sridhar Rajagopalan. Computing on data streams. *External memory algorithms*, 50:107–118, 1998.

[35] Kamal Jain and Vijay V Vazirani. Primal-dual approximation algorithms for metric facility location and k-median problems. In *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*, pages 2–13. IEEE, 1999.

[36] David S Johnson. Approximation algorithms for combinatorial problems. *Journal of computer and system sciences*, 9(3):256–278, 1974.

[37] Michael Kapralov, Yin Tat Lee, CN Musco, CP Musco, and Aaron Sidford. Single pass spectral sparsification in dynamic streams. *SIAM Journal on Computing*, 46(1):456–477, 2017.

[38] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[39] Eyal Kushilevitz. Communication complexity. In *Advances in Computers*, volume 44, pages 331–360. Elsevier, 1997.

[40] Ching Lih Lim, Alistair Moffat, and Anthony Wirth. Lazy and eager approaches for the set cover problem. In *Proceedings of the Thirty-Seventh Australasian Computer Science Conference-Volume 147*, pages 19–27. Australian Computer Society, Inc., 2014.

[41] Anuj Mehrotra, Kenneth E Murphy, and Michael A Trick. Optimal shift scheduling: A branch-and-price approach. *Naval Research Logistics (NRL)*, 47(3):185–200, 2000.

[42] Milena Mihail. Set cover with requirements and costs evolving over time. In *Randomization, approximation, and combinatorial optimization. algorithms and techniques*, pages 63–72. Springer, 1999.

[43] Barna Saha and Lise Getoor. On maximum coverage in the streaming model & application to multi-topic blog-watch. In *Proceedings of the 2009 siam international conference on data mining*, pages 697–708. SIAM, 2009.

[44] Stergios Stergiou and Kostas Tsioutsiouliklis. Set cover at web scale. In *Proceedings of the 21th acm sigkdd international conference on knowledge discovery and data mining*, pages 1125–1133. ACM, 2015.

Appendix A

Appendix

A.1 Proof of approximation for the modified Rosen

We show a $f(n) = (\frac{r^*+2}{r^*-1} + 1) \cdot \sqrt{n}$-approximation factor where $\lim_{r^* \to \infty} f(n) = 2 \cdot \sqrt{n}$ for the modified Rosen (an un-weighted set-cover algorithm) by adapting the proof of $8 \cdot \sqrt{n}$ approximation for the Rosen (an un-relaxed weighted, partial set-cover algorithm) detailed by Rosen & Emek. [26].

Observation A.1.1. *If $T \subseteq e_t$ where $v \in T$ is an effective subset at time t, then $T' = T \cup \{u\}$ is an effective subset where $u \in e_t$ iff. eff$(u) \leq$ eff(v).*

Proof. It is obvious that $|T'| \geq |T|$. Therefore given eff$(u) <$ eff(v) and $|T| >$ eff(v), this implies that $|T'| >$ eff(u), this makes T' an effective subset of e_t. $\square$

Observation A.1.2. *If $T \subseteq e_t$ is an effective subset at time t, then eff$_{t+1}(v) \geq$ eff$_t(v) = |T|$ i.e. the effectivity of an element is non-decreasing.*

Proof. Importantly these two observations establish that the effectivity of an element, eff(v) is a non-decreasing quantity. $\square$

Lemma A.1.3. *For some integer r, the algorithm guarantees that $|\{v \in e_t | \text{eff}_{t+1}(v) \leq r\}| < r + 1$ i.e. that the effectivity of an effective subset at iteration t is upper-bounded by r + 1 for all elements whose effectivity at t + 1 is still less than r.*

Proof. Prove by contradiction. If $R = \{v \in e_t | \text{eff}_{t+1}(v) \leq r\}$ and $|R| \geq r + 1$. Therefore R is an effective subset at time t for $v \in T$ implying eff$_{t+1}(v) = r + 1$. A contradiction as we stated eff$_{t+1}(v) \leq r$ $\square$

Lemma A.1.4. *The algorithm guarantees that $|I(\leq r)| < r + 1 \cdot |OPT|$ where $I(\leq r) = \{v \in V | \text{eff}_\infty(v) \leq r\}$.*

Proof. By non-decreasing effectivity, its holds that $|\{v \in e | \text{eff}_\infty(v) \leq r\}| < r + 1$ i.e. at termination there is no set of the stream with effective subset T where $|T| > r$. Thus

$$|I(\leq r)| < \sum_{e \in OPT} |\{v \in e| \text{ eff}_\infty(v) \leq r\}| < \sum_{e \in OPT} (r + 1) < (r + 1) \cdot |OPT|$$

. The first equality holds as OPT is a set-cover of $\mathcal{U}$. $\square$

Lemma A.1.5. *The set collection $S(r) = \{s \in \mathcal{S} | \exists v \in I(r) \text{ s.t. } eid(v) = id(v)\}$ satisfies $c(S(r)) \leq b(V)/r$.*

Proof. For $e \in S(r)$ then $R(e) \subseteq e$ where $|R| = r$ implying that $|R(e)| > (r - 1)$. If $e_t, e'_t \in S(r)$ then subsets $R(e_t)$ and $R(e'_t)$ are disjoint.

$$\sum_{e \in S(r)} 1 \leq \sum_{e \in S(r)} |R(e)|/(r-1) \leq |V|/(r-1)$$

. $\square$

Corollary A.1.5.1. *The extension on the cost upper-bound of $S(\geq r)$ comes in the worst case when $I(r) = I(\geq r)$ i.e. for $e \in S(\geq r)$ with $R(e) \subseteq e$ where $|R| = r$ implying that $c(S(\geq r) \leq b(V)/(r-1)$.* $\square$

Lemma A.1.6. *For some $0 < \varepsilon \leq 1$, let r^* be the largest integer s.t. $|S(\leq r^*)| \leq \varepsilon.|V|$ i.e. the value of r^* s.t. the total cardinality of vertices covered with effectivity r^* is less the total cardinality all the elements of the universe. Then it is true that $|S \geq r^*| < \frac{r^*+2}{r^*-1} \cdot |OPT| \cdot \frac{1}{\varepsilon}$.*

Proof. Let r^* be an integer $r^* + 1 \geq \varepsilon \cdot \frac{|V|}{|OPT|} \geq r^* + 2$. From Lemma 2, we can guarantee that

$$|I(\leq r^*)| \geq 2^{r^*+1} \cdot |OPT| \leq \varepsilon \cdot b(V)$$

.

Following from **Corollary 1**

$$|S(\geq r^*)| \leq |V|/(r-1) \leq \frac{r+2}{r-1} \cdot |OPT| \cdot \frac{1}{\varepsilon}$$

. $\square$

The algorithm outputs a un-weighted complete set-cover certificate for $(\mathcal{U}, \mathcal{S})$ whose image has cost $O(\sqrt{n} \cdot |OPT|)$.

Proof. Let $|I(\leq r^{**})| = |\mathcal{U}|$ where $\mathcal{X} : \mathcal{U} \to [1, m]$ is a valid complete cover-certificate for the un-weighted MSC. Let $\mathcal{X}'$ be the (partial) function $\mathcal{X}' : \mathcal{U} - I(\leq \text{r}^*) \to [1, m])$ where $|I(\leq r^*)| \leq \sqrt{n}$. Let $\mathcal{X}''$ be the (complete) function $\mathcal{X}'' : \mathcal{U} \to [1, m]$ extended by mapping $v \in I(\leq \text{r}^*)$ to $e \in S(\leq r^{**})$ where $v \in R(e)$.

From **Lemma 4.** it follows that $c(X') < \frac{r^*+2}{r^*-1} \cdot \sqrt{n} \cdot c(OPT)$ and since $1 \leq |OPT|$ and $|I(\leq r^*)| \leq \sqrt{n}$, it follows that

$$|Im(\mathcal{X}')| \leq \frac{r^*+2}{r^*-1} \cdot \sqrt{n} \cdot |OPT| + |I(\leq r^*)| \cdot |OPT| \leq (\frac{r^*+2}{r^*-1} + 1) \cdot \sqrt{n} \cdot |OPT|$$

upper bounding the solution quality as $O(\sqrt{n} \cdot |OPT|)$. $\square$