Sketching data structures for massive graph problems

Juan P. A. Lopes[1], Fabiano S. Oliveira[2], Paulo E. D. Pinto[2], and Valmir C. Barbosa[1]

[1] Federal University of Rio de Janeiro, Brazil
{jlopes,valmir}@cos.ufrj.br
[2] State University of Rio de Janeiro, Brazil
{fabiano.oliveira,pauloedp}@ime.uerj.br

Abstract. In this work, we explore the application of sketching data structures to solve problems in graphs that do not fit entirely in memory. These structures allow compact representations of data, admitting some probability of failure. We aim at the implicit representation and dynamic connectivity problems. Our contributions include two new probabilistic implicit representations, one that uses Bloom filters and allows representing sparse graphs with $O(|E|)$ bits, and another that uses MinHash sketches and represents trees with $O(|V|)$ bits. We also describe a variant of an ℓ_0-sampling sketch that allows proving a tighter upper bound on the failure probability of sampling.

Keywords: Sketching data structures · Graphs · Stream algorithms.

1 Introduction

Sketching data structures allow the representation of data in a compact fashion, often in sublinear space with respect to the original data. The interest in these data structures has increased in recent years, as a direct consequence of the emergence of applications that deal with large volumes of streaming data. In these applications, it is often necessary to answer queries quickly, which is infeasible by simply querying over stored data due to high latency. Be that as it may, such volumes do not generally fit into memory in the first place. Sketching data structures offer a good compromise for many applications, allowing less memory and CPU usage at the cost of decreased accuracy.

In this work, we survey some sketching data structures and their applications to massive graph problems. In Section 2, we describe the application of Bloom filters and MinHash sketches to the implicit graph representation problem [16], one of them representing trees with better space complexity than the optimal deterministic representation. In Section 3, we detail two variants of a sketch to solve the ℓ_0-sampling problem, which can be used to determine dynamic connectivity in n-vertex graph streams using $O(n \log^3 n)$ bits [1, 13].

2 Probabilistic implicit graph representations

An *implicit graph representation* is a vertex labeling scheme that allows testing the adjacency between any two vertices efficiently by just comparing their labels [15, 9, 16]. More formally, given a graph class $\mathcal{C}$ with $2^{\Theta(f(n))}$ graphs with n vertices, a representation is said to be *implicit* if

1. it is *space-optimal*, that is, it requires $O(f(n))$ bits to represent graphs in $\mathcal{C}$;
2. it *distributes information evenly* among vertices, that is, each vertex is represented by a *label* using $O(f(n)/n)$ bits;
3. the *adjacency test is local*, that is, when testing the adjacency of any two vertices, only their labels are used in the process.

According to this definition, the *adjacency matrix* is an implicit representation of the class containing all graphs, because there are $2^{\Theta(n^2)}$ graphs on n vertices and the adjacency matrix can represent them using $\Theta(n^2)$ bits. On the other hand, for m the number of edges, the *adjacency list* is not an implicit representation, because it requires $\Theta(m \log n)$ bits to represent the same graph class, which may require $\Theta(n^2 \log n)$ bits in the worst case (e.g., for complete graphs). In contrast, an adjacency list is space-optimal to represent trees, as $O(m \log n) = O(n \log n)$ for trees and there are $2^{\Theta(n \log n)}$ trees on n vertices, but still it is not an implicit representation because it does not distribute information evenly: each tree vertex may use $\Theta(n \log n)$ bits to represent its adjacency in an adjacency list (e.g., the center vertices of stars).

In [11], the concept of *probabilistic implicit graph representations* was explored, extending the concept of implicit representations by relaxing one of the properties: the adjacency test is *probabilistic*, meaning that it has a constant probability of resulting in false negatives or false positives. A 0% chance of false positives and negatives implies an ordinary implicit representation. The main benefit of probabilistic representations is the ability to trade accuracy for memory, that is, to achieve more space-efficient representations by allowing some incorrect results in adjacency tests. We present two novel probabilistic implicit representations, each based on a distinct sketching data structure.

2.1 Representation based on Bloom filters

The Bloom filter is a data structure that represents a set $S' \subseteq S$ and allows testing elements for set membership with some probability of false positives, but no false negatives [2]. A Bloom filter consists of an array M of m bits and k pairwise independent hash functions, $h_i : S \to [1, \ldots, m]$ for $1 \leq i \leq k$. The insertion of an element x is performed by computing k hash values, $h_1(x), \ldots, h_k(x)$, and setting these indices in the array to 1, that is, $M[h_i(x)] \leftarrow 1$ for all $1 \leq i \leq k$. The membership query for some element x is done by verifying whether all bits in positions given by the hash values are 1, that is, by verifying whether $M[h_i(x)] = 1$ for all $1 \leq i \leq k$. If at least one bit is 0, then x is certainly not in the set. If all bits are 1, it is assumed that the element is in the

set, although this may not be the case (a false positive). The probability of a false positive when n elements are already stored (event FP) can be determined from the probability of collisions in all k hash values, that is,

$$\Pr[\mathrm{FP}] = \Pr\left[\bigwedge_{1\le i\le k} M[h_i(x)] = 1\right] = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - \mathrm{e}^{-kn/m}\right)^k.$$

Defining $q = m/n$, that is, q as the ratio between the size of M in bits and the number of stored elements, it is possible to show that the probability of false positives is minimized when $k \approx q \ln 2$, so $\Pr[\mathrm{FP}] \approx (1 - e^{-\ln 2})^{q \ln 2} \approx 0.6185^q$. Thus, for example, setting the dimension of M to 10 bits per element and using 7 hash functions, it is possible to estimate set membership with less than 1% of false positives.

Bloom filters are commonly used in database systems, both to avoid the attempt to fetch non-existing data and to optimize communication costs in distributed joins. In summary, Bloom filters are useful in contexts where the performance gain in negative queries makes up for the cost of false positives.

Bloom filters can also be used in implicit graph representations, as follows. For each vertex, a Bloom filter is created using some constant number of bits per element (say, 10 bits), representing the set of vertices adjacent to it. The set of Bloom filters of all vertices constitutes a probabilistic implicit representation. This representation requires $\Theta(\sum_{v \in V(G)} d(v) = 2m)$ bits to represent any graph, which makes it equivalent to the adjacency matrix in the worst case (e.g., for complete graphs). However, this representation has better space complexity for sparse graphs than the deterministic one. In fact, it is better for any graph having $m = o(n^2)$. Also, it has the property of not allowing false negatives in adjacency tests. That is, it will never fail to report an existing edge, although it may report the existence of non-existing edges with a small probability.

The theoretical predictions about this representation were verified through three practical experiments. These experiments aimed to validate the rate of false positives as the graph's density ($2m/(n^2 - n)$) or the number of bits per edge (q) changed while keeping other parameters fixed (results are shown in Figure 1).

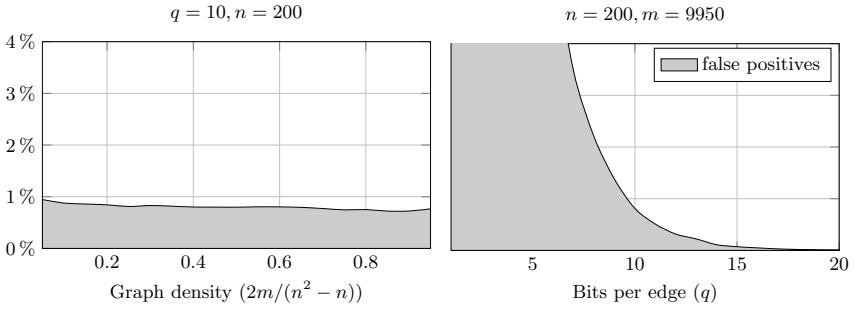

Fig. 1. Rate of false positives.

2.2 Representation based on MinHash

MinHash is a sketching data structure that represents sets $A, B \in \mathcal{S}$ and allows estimating their Jaccard coefficient, $J(A,B) = \frac{|A \cap B|}{|A \cup B|}$ [3]. The estimation is done by computing a signature (a k-tuple of hash values) for each set $S \in \mathcal{S}$, using k pairwise independent hash functions $h_1, \ldots, h_k$. Each element in the signature is given by $h_i^{\min}(S) = \min\{h_i(x) : x \in S\}$, $1 \leq i \leq k$. The probability of two sets A and B having a common signature element can be shown to be equal to their Jaccard coefficient, that is, $\Pr[h_i^{\min}(A) = h_i^{\min}(B)] = J(A,B)$, $1 \leq i \leq k$. Given two sets A, B, let X_i denote the Bernoulli random variable such that $X_i = 1$ if $h_i^{\min}(A) = h_i^{\min}(B)$, or $X_i = 0$ otherwise. The set $\{X_1, \ldots, X_k\}$ consists of an independent set of unbiased estimators for $J(A,B)$, in such a way that increasing k decreases the estimator variance. The error bounds for the estimation of $J(A,B)$ can be proved using the Chernoff inequalities. In particular, to achieve an error factor of θ with probability greater than $1-\delta$, k should be chosen such that $k \geq \frac{2+\theta}{\theta^2} \ln(2/\delta)$.

MinHash's original motivation remains its most useful application, detecting plagiarism. It is possible to evaluate the similarity of two documents by only comparing their MinHash signatures in constant time. It can also be used in conjunction with HyperLogLog [7] to estimate the cardinality of set intersection without having both sets in the same machine [12].

In the context of graphs, we introduced a probabilistic implicit representation based on MinHash in which the main idea is, for any graph $G = (V,E)$ in a class $\mathcal{C}$ and for some pair of constants $0 \leq \delta_A < \delta_B \leq 1$, to find representing sets $S_v \neq \emptyset$ for every $v \in V$ such that the following two conditions hold: (i) $J(S_u, S_v) \geq \delta_B$ if and only if $(u,v) \in E$, and (ii) $J(S_u, S_v) \leq \delta_A$ if and only if $(u,v) \notin E$. Therefore, no pairwise Jaccard coefficient of representing sets should lie within the interval (δ_A, δ_B). This way, the adjacency (u,v) could be tested by determining $J(S_u, S_v)$ and comparing it with δ_A and δ_B. We use MinHash to provide not the exact values, but estimates of the Jaccard coefficients. Therefore, the actual idea to test adjacency is to assume that $(u,v) \in E$ if $J(S_u, S_v) > \delta$ for some $\delta_A \leq \delta \leq \delta_B$. Note that only the signatures of the representing sets must be stored, requiring a constant number of elements. Furthermore, those signatures can be represented with a constant number of bits [10], and therefore a representation based on MinHash requires $O(n)$ bits to represent any class for which such representing sets exist.

In [11], we presented an algorithm to build such representing sets for trees with $\delta_A = 1/3$ and $\delta_B = 1/2$. Given a tree T, the construction is performed recursively starting at an arbitrary vertex v, with S_v being defined with ℓ arbitrary distinct elements, where $\ell = \min\{2^r : r \in \mathbb{N} \mid 2\Delta(T) \leq 2^r\}$. Transforming T into a tree rooted at v, for each level the procedure alternates between choosing S_u as a subset of S_p (*selection* phase) and choosing S_u as a superset of S_p (*extension* phase), where p is the parent of u in T. Figure 2 exemplifies this construction.

The selection phase is done as follows. For a set $S_p = \{a_1, \ldots, a_x\}$, $x/2$ subsets $U_1, \ldots, U_{x/2}$ are selected from it, each with $x/2$ elements, such that each pair of subsets has $x/4$ elements in common. This way, $J(U_i, U_j) = 1/3$ for

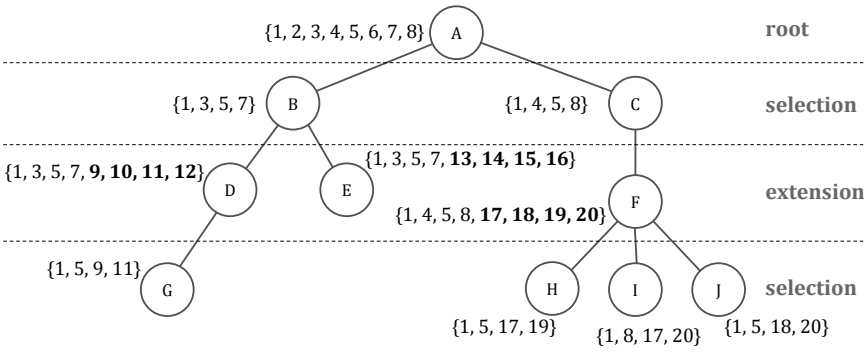

Fig. 2. Example of representing sets for a given tree.

$1 \leq i < j \leq x/2$ and $J(U_i, S_p) = 1/2$ for $1 \leq i \leq x/2$. Thus, each child of p must be assigned a distinct U_i as its representing set. The efficient implementation of this selection procedure is based on the representation by a binary string u_i, with length $x/2$, of a subset $U_i \subset S_p$, such that if the j^{th} bit of u_i has value b, then a_{2j-1+b} belongs to U_i. The generation of the strings that represent $U_1, \ldots, U_{x/2}$ can be achieved iteratively, starting from a 1×1 matrix and, at each step, fourfolding the current matrix with negated bits in the lower right quadrant. This is illustrated in Figure 3 for $S_p = \{1, \ldots, 8\}$. The extension phase is done through the inclusion of $|S_p|$ unique elements from the already defined representing sets.

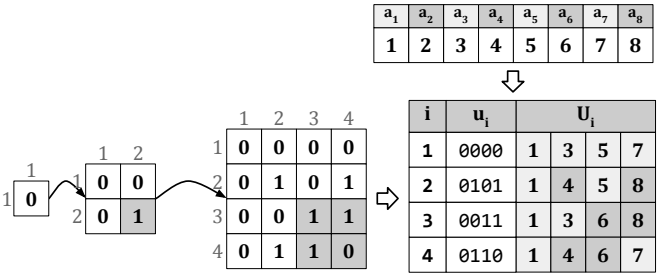

Fig. 3. Example of a subset selection for $S_p = \{1, \ldots, 8\}$.

The MinHash signatures are then computed for the representing sets and used as labels for the corresponding vertices. As this labeling scheme requires only $O(n)$ space to probabilistically represent trees, a class with $2^{\Theta(n \log n)}$ graphs on n vertices, such a probabilistic representation has better space complexity than the optimal deterministic representation.

The theoretical predictions about this representation were verified through three practical experiments. The experiments aimed to validate the rate of false positives and negatives as we change the evaluation threshold (δ), the number

of vertices in the graph (n), and the signature size (k), while keeping the other parameters fixed. The results are shown in the Figure 4.

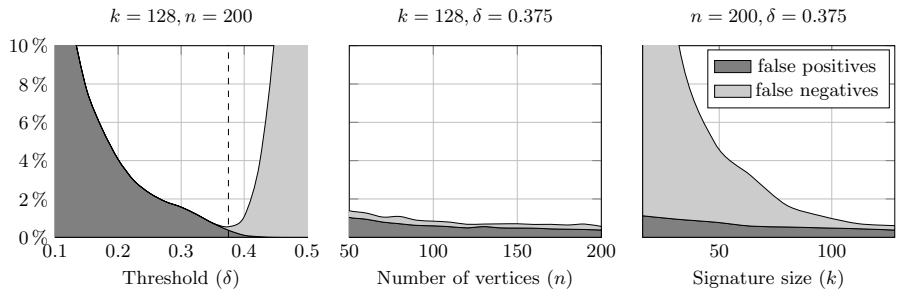

Fig. 4. Rate of false positives and negatives.

2.3 Considerations on bipartite graphs

In [16], it is shown that any hereditary graph class with $2^{\Theta(n^2)}$ members of n vertices should entirely include either the bipartite, co-bipartite, or split graphs. Also, it is possible to transform any graph $G = (V, E)$ into a bipartite graph $G' = (V', E')$ such that $V' = \{v_1, v_2 \mid v \in V\}$, and $E' = \{(u_1, v_2), (v_1, u_2) \mid (u, v) \in E\}$. Any efficient representation of G' can be used to efficiently represent G. This makes the search for a probabilistic representation of bipartite graphs specially appealing. However, we proved the non-existence of some representations. For example, it is impossible to construct a MinHash-based representation with $\delta_A = 0.4$ and $\delta_B = 0.6$ for a graph as simple as the complete bipartite $K_{3,3}$ [11]. Our proof is based on the formulation of a corresponding integer linear programming problem, which turns out to be infeasible. This suggests that further investigation concerning this probabilistic implicit graph representation is that of characterizing the class of graphs that are amenable to it.

3 Graph-streams connectivity

In many real applications, graphs are not static entities. Instead, it is often the case that edges and vertices are added and removed with high frequency. The study of fully dynamic graph algorithms is already well established [6, 13], but the recent explosion in the scale of graphs has encouraged further research into algorithms that require sublinear space to compute queries on them. In this work, we present two variants of an ℓ_0-sampling sketch, a data structure that allows the sampling of edges in graph cut-sets and can be used to determine dynamic graph connectivity using $O(n \log^3 n)$ bits.

3.1 ℓ_0-sampling sketch

The ℓ_0-sampling problem consists in sampling a nonzero coordinate from a dynamic vector $\boldsymbol{a} = (a_1, \ldots, a_n)$ with uniform probability. This vector is defined in a turnstile model, which consists of a stream of updates $S = \langle s_1, s_2, \ldots, s_t \rangle$ on $\boldsymbol{a}$ (initially $\mathbf{0}$), where $s_i = (u_i, \Delta_i) \in \{1, \ldots, n\} \times \mathbb{R}$ for $1 \leq i \leq t$, meaning an increment of Δ_i units to a_{u_i}. It is desirable that such sample be produced in a single pass through the stream with sublinear space complexity. The challenge arises from the fact that, since Δ_i can be negative and hence some updates in the stream may cancel others, directly sampling the stream may lead to incorrect results. In order to achieve sublinear space complexity in a single pass, an ℓ_0-sampling algorithm must represent $\boldsymbol{a}$ through a sketch.

In [5], a seminal sketch-based algorithm for the ℓ_0-sampling problem was introduced. The algorithm uses a universal family of hash functions to partition the vector $\boldsymbol{a}$ into $O(\log n)$ subvectors with exponentially decreasing probabilities of representing each element of $\boldsymbol{a}$. It is proved that there is a constant lower bound on the probability that at least one of those subvectors has exactly one nonzero coordinate. Through a procedure called 1-*sparse recovery*, which stores $O(\log n)$ bits for each subvector, it is possible to recover such coordinate. Considering that the probability of failure has a constant upper bound, running $O(\log(1/\delta))$ independent instances of the algorithm can ensure a success probability of at least $1 - \delta$. The total space complexity of this algorithm is $O(\log^2 n \log(1/\delta))$. Further studies show stronger results by relaxing assumptions on the hash functions used [14, 8]. Nevertheless, they keep the same worst-case space complexity. In fact, any algorithm that performs ℓ_0-sampling in a single pass should require $\Omega(\log^2 n)$ bits in the worst case [8]. This holds even if the algorithm allows a relative error of ϵ and a failure probability of δ.

1-sparse recovery procedure A vector is 1-*sparse* when it has a single nonzero coordinate. A 1-sparse recovery procedure allows deciding whether a vector $\boldsymbol{a}$ is 1-sparse, and recover the only nonzero coordinate from it. Note that while $\boldsymbol{a}$ is expected to be 1-sparse at the time of a successful recovery, it may have any number of nonzero coordinates before that. This procedure is a building block for many ℓ_0-sampling algorithms. Here we present a false-biased randomized variant that handles cases where $\boldsymbol{a}$ has negative values [4]. It begins by choosing a sufficiently large prime $p \leq n^c$, for some constant $c > 1$, and a random integer $z \in \mathbb{Z}_p$. Then, iterating through all $s_i = (u_i, \Delta_i) \in S$, three sums are computed:

$$b_0 = \sum_{i=1}^{t} \Delta_i, \qquad b_1 = \sum_{i=1}^{t} \Delta_i u_i, \qquad b_2 = \sum_{i=1}^{t} \Delta_i z^{u_i} \mod p.$$

If $\boldsymbol{a}$ is 1-sparse, it is easy to see that the nonzero coordinate i can be recovered as $i = b_1/b_0$, with $a_i = b_0$. However, verifying that $\boldsymbol{a}$ is 1-sparse requires more effort.

Theorem 1. *If $\boldsymbol{a}$ is 1-sparse, then $b_2 \equiv b_0 z^{b_1/b_0} \mod p$. Otherwise, $b_2 \not\equiv b_0 z^{b_1/b_0} \mod p$ with probability at least $1 - n/p$.*

Proof (sketch). If $\boldsymbol{a}$ is 1-sparse, with a nonzero coordinate i, it is trivial to see that $b_2 \equiv a_i z^i \mod p$. Otherwise, $b_2 \equiv b_0 z^{b_1/b_0} \mod p$ may still hold if z is a root in $\mathbb{Z}_p$ of the polynomial $p(z) = b_0 z^{b_1/b_0} - \sum \Delta_i z^{u_i}$. As $p(z)$ is a degree-n polynomial, it has at most n roots in $\mathbb{Z}_p$. Therefore, given that z is chosen at random, the probability of a false recovery is at most n/p. □

This 1-sparse recovery procedure stores z, b_0, b_1, and b_2. Assuming that every a_i is limited by a polynomial in n, the total space required is $O(\log n)$ bits.

Algorithm Here, two variants of the same ℓ_0-sampling sketch are presented. Both variants define $\boldsymbol{a}^{(1)}, \boldsymbol{a}^{(2)}, \ldots, \boldsymbol{a}^{(m)}$ subvectors of $\boldsymbol{a}$. For all $1 \leq j \leq m$, each $a_i \neq 0$ has a $1/2^j$ probability of being *present* at $\boldsymbol{a}^{(j)}$, that is, $a_i^{(j)} = a_i$ with probability $1/2^j$, otherwise $a_i^{(j)} = 0$. To decide whether $a_i^{(j)}$ is present, we draw a hash function $h_j : \{1, \ldots, n\} \to \{0, \ldots, 2^m - 1\}$ from a universal family, and observe whether $m - \lfloor \log_2 h_j(i) \rfloor = j$, which happens with probability $1/2^j$. An independent 1-sparse recovery is then computed for each $\boldsymbol{a}^{(j)}$. The variants differ only in the number of functions used. Variant (a) uses a single hash function for every $\boldsymbol{a}^{(j)}$ (Algorithm 1), while Variant (b) uses a different function for each subvector (Algorithm 2). While Variant (a) is more useful in practice, the error analysis for Variant (b) is more straightforward. We provide empirical evidence that the error in either variant converges quickly as a function of n.

Algorithm 1 Variant (a)

```
M[1..m]: 1-sparse recoveries
for each (u_i, Δ_i) ∈ S do
    k ← m − ⌊log_2 h(u_i)⌋
    M[k].b_0 += Δ_i
    M[k].b_1 += Δ_i u_i
    M[k].b_2 += Δ_i M[k].z^{u_i}  mod p
for j ∈ [1..m] do
    v ← M[j].b_0 M[j].z^{M[j].b_1/M[j].b_0}  mod p
    if M[j].b_2 = v then
        return M[j].b_1/M[j].b_0
report FAILURE
```

Algorithm 2 Variant (b)

```
M[1..m] : 1-sparse recoveries
for each (u_i, Δ_i) ∈ S do
    for j ∈ [1..m] do
        k ← m − ⌊log_2 h_j(u_i)⌋
        if k = j then
            M[k].b_0 += Δ_i
            M[k].b_1 += Δ_i u_i
            M[k].b_2 += Δ_i M[k].z^{u_i}  mod p
for j ∈ [1..m] do
    v ← M[j].b_0 M[j].z^{M[j].b_1/M[j].b_0}  mod p
    if M[j].b_2 = v then
        return M[j].b_1/M[j].b_0
report FAILURE
```

Each variant either succeeds in returning a single nonzero coordinate of $\boldsymbol{a}$ or reports a failure. The probability of failure is given by the joint probability of failure of all m 1-sparse recoveries. In Variant (b), these are independent events. The probability that a single recovery $M[j]$ fails is the complement of the probability that $\boldsymbol{a}^{(j)}$ is 1-sparse, that is, assuming $\boldsymbol{a}$ has $r \gg 1$ nonzero coordinates:

$$\Pr[\textsc{Failure}] = \prod_{j=1}^{m} \left(1 - r2^{-j}(1 - 2^{-j})^{r-1}\right) \approx \prod_{j=1}^{m} \left(1 - r2^{-j} e^{-r2^{-j}}\right).$$

Theorem 2. *If* $5 \leq \log_2 r \leq m - 5$*, then* $\Pr[\textsc{Failure}] \leq 0.31$ *for Variant (b).*

Proof (sketch). It is easy to see that the lowest probabilities of failure concentrate around j such that $2^j \leq r < 2^{j+1}$. Letting $q = r/2^{\lfloor \log_2 r \rfloor}$, it holds that

$$\Pr[\textsc{Failure}] \leq \prod_{k=-5}^{5} \left(1 - q2^k e^{-q2^k}\right).$$

Note that $1 \leq q < 2$. In this interval, all factors $1 - q2^k e^{-q2^k}$ are either monotonically increasing or decreasing. Analyzing their global maxima, we arrive at a maximum product of approximately 0.3071, therefore $\Pr[\textsc{Failure}] \leq 0.31$. □

This result shows that, as n grows, choosing $m = 5 + \lceil \log_2 n \rceil$ is enough to ensure a constant upper bound on the probability of failure. Furthermore, to ensure a success probability of at least $1 - \delta$, it is sufficient to run $\lceil \log_{0.31} \delta \rceil$ instances of the sketch.

In order to assess the algorithm's behavior in a real implementation, an experiment was set up. Both variants were implemented and tested with a vector of size $n = 4\,096$ and increasing values of r. We tested both a correctly sized (i.e., for $m = 17$) and an undersized instance of the ℓ_0-sampling sketch. The empirical cumulative distribution was also recorded. The experiment was run 100 000 times and the mean value for each data point is reported in Figure 5.

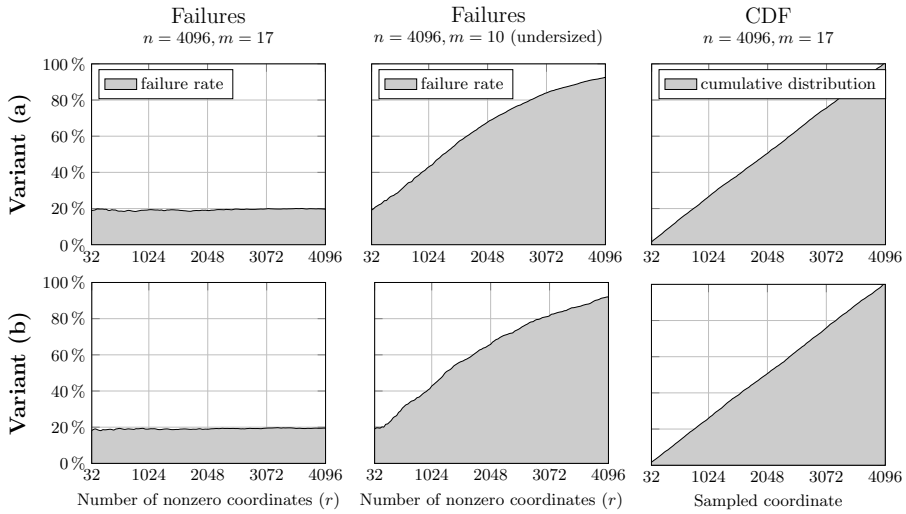

Fig. 5. Failure rate and cumulative distribution of successes.

This experiment suggests that in a correctly sized ℓ_0-sampling sketch, the failure probability stays almost constant and under 20%. There is little difference

between Variants (a) and (b). Furthermore, in an undersized setup, the failure rate rapidly reaches critical levels.

3.2 Dynamic connectivity using ℓ_0-samplers

It is possible to use ℓ_0-sampling sketches to determine whether a dynamic graph $G = (V, E)$ is connected. One possible randomized algorithm runs in $O(\log n)$ turns and either answers affirmatively with certainty or negatively with a constant probability of error [1].

The algorithm starts with an empty subgraph of G. In each turn, for each connected component $S \subset V$, an edge is drawn (if any) from the cut-set $[S, V \setminus S]$, connecting two components. It is possible to prove that this procedure finishes in at most $\lceil \log_2 n \rceil$ turns, yielding a spanning tree of G if it is connected.

The ℓ_0-sampling sketches are used to represent each vertex set's adjacency, in the form of a modified incidence vector, where each edge is represented twice, one for each ordering of its ends. More formally, given an ordering $u_1 w_1, \ldots, u_m w_m$ of the edges of E, we define a vector $\boldsymbol{a}^v = (a^v_{u_1 w_1}, a^v_{w_1 u_1}, \ldots, a^v_{u_m w_m}, a^v_{w_m u_m})$, for each vertex $v \in V$, in a way that $a^v_{u,w} = 1$ if $v = u$; $a^v_{u,w} = -1$, if $v = w$; or $a^v_{u,w} = 0$, otherwise.

This representation has the useful property that, for each set of vertices $S = \{v_1, v_2, \ldots, v_q\}$, the nonzero coordinates of $\boldsymbol{a}^S = \sum_{i=1}^{q} \boldsymbol{a}^{v_i}$ represents the edges across the cut $[S, V \setminus S]$. Therefore, considering that the ℓ_0-sampling representation of any vector $\boldsymbol{a}$ is a linear transformation of that vector, this implies that a set of ℓ_0-sampling sketches can be used to sample edges in any cut-set of a graph.

It is important to note that an ℓ_0-sampling sketch cannot be reused to sample another edge with the same failure probability. Nevertheless, a different sampling sketch can be used in each turn of the algorithm. Keeping $\lceil \log_2 n \rceil$ ℓ_0-sampling sketches (one for each turn) for each vertex allows performing the connectivity algorithm just described using $O(n \log^3 n)$ bits.

4 Conclusion

In this paper we explored the use of sketching data structures for massive graph problems. We have established the concept of probabilistic implicit graph representations, introducing two new representations. One, based on Bloom filters, can represent sparse graphs with $O(m)$ bits; another, based on MinHash, can represent trees with $O(n)$ bits. We have provided empirical evidence confirming the theoretical predictions about these representations.

We have also described a variant of the ℓ_0-sampling sketch and proved its failure probability to be bounded by a constant value, provided a certain structure-size condition is met. A simple dynamic graph connectivity algorithm using this sketch was explained. Research is ongoing on the proof of exact probabilities of failure for both algorithm variants. Future research may also include novel graph algorithms that use ℓ_0-sampling as a primitive.

Acknowledgements

The authors acknowledge partial financial support from CNPq, CAPES, and a FAPERJ BBP grant.

References

1. Ahn, K.J., Guha, S., McGregor, A.: Analyzing graph structure via linear measurements. In: Proceedings of SODA'12. pp. 459–467 (2012)
2. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM **13**(7), 422–426 (1970)
3. Broder, A.Z.: On the resemblance and containment of documents. In: Proceedings of SEQUENCES'97. pp. 21–29 (1997)
4. Cormode, G., Firmani, D.: A unifying framework for ℓ_0-sampling algorithms. Distributed and Parallel Databases **32**(3), 315–335 (2014)
5. Cormode, G., Muthukrishnan, S., Rozenbaum, I.: Summarizing and mining inverse distributions on data streams via dynamic inverse sampling. In: Proceedings of VLDB'05. pp. 25–36 (2005)
6. Eppstein, D., Galil, Z., Italiano, G.F.: Dynamic graph algorithms. In: Atallah, M.J. (ed.) Algorithms and Theory of Computation Handbook, chap. 8. CRC Press (1999)
7. Flajolet, P., Fusy, É., Gandouet, O., Meunier, F.: HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In: Proceedings of AofA'07. pp. 127–146 (2007)
8. Jowhari, H., Sağlam, M., Tardos, G.: Tight bounds for L_p samplers, finding duplicates in streams, and related problems. In: Proceedings of PODS'11. pp. 49–58 (2011)
9. Kannan, S., Naor, M., Rudich, S.: Implicit representation of graphs. SIAM Journal on Discrete Mathematics **5**(4), 596–603 (1992)
10. Li, P., König, A.C.: b-bit minwise hashing. In: Proceedings of WWW'10. pp. 671–680 (2010)
11. Lopes, J.P.A.: Probabilistic data structures applied to implicit graph representation. Master's thesis, State University of Rio de Janeiro (2017), in Portuguese
12. Lopes, J.P.A., Oliveira, F.S., Pinto, P.E.D.: Estimating the intersection cardinality of sets using MinHash and HyperLogLog. In: Proceedings of CNMAC'16. pp. 010077-1–2 (2017), in Portuguese
13. McGregor, A.: Graph stream algorithms: a survey. ACM SIGMOD Record **43**(1), 9–20 (2014)
14. Monemizadeh, M., Woodruff, D.P.: 1-pass relative-error L_p-sampling with applications. In: Proceedings of SODA'10. pp. 1143–1160 (2010)
15. Muller, J.H.: Local structure in graph classes. Ph.D. thesis, Georgia Institute of Technology (1988)
16. Spinrad, J.P.: Efficient graph representations. American Mathematical Society (2003)