2-3 Cuckoo Filters for Faster Triangle Listing and Set Intersection

David Eppstein
Department of Computer Science
Univ. of California, Irvine
eppstein@uci.edu

Michael T. Goodrich
Department of Computer Science
Univ. of California, Irvine
goodrich@uci.edu

Michael Mitzenmacher
Department of Computer Science
Harvard Univ.
michaelm@eecs.harvard.edu

Manuel R. Torres
Department of Computer Science
Univ. of California, Irvine
mrtorres@uci.edu

ABSTRACT

We introduce new dynamic set intersection data structures, which we call ***2-3 cuckoo*** filters and hash tables. These structures differ from the standard cuckoo hash tables and cuckoo filters in that they choose two out of three locations to store each item, instead of one out of two, ensuring that any item in an intersection of two structures will have at least one common location in both structures. We demonstrate the utility of these structures by using them in improved algorithms for listing triangles and answering set intersection queries in internal or external memory. For a graph G of n vertices and m edges, our internal-memory triangle listing algorithm runs in $O(m\lceil(\alpha(G)\log w)/w\rceil + k)$ expected time, where $\alpha(G)$ is the arboricity of G, w is the number of bits in a machine word, and k is the number of output triangles. Our external-memory algorithm uses $O(\mathrm{sort}(n\,\alpha(G)) + \mathrm{sort}(m\lceil(\alpha(G)\log w)/w\rceil) + \mathrm{sort}(k))$ expected number of I/Os.

1. INTRODUCTION

Set intersection queries have a multitude of database applications. They are the "inner loop" computation for performing inner joins, and can be used to answer ***triangle listing*** queries for databases of graphs and networks, where we must report every triple of mutually-adjacent vertices [13, 46]. In this paper, we provide asymptotic improvements to set intersection and triangle listing queries, by taking advantage of the bit-level parallelism available in the ***practical RAM*** model [41] (which is also known as the ***word-RAM*** model [33]) or in the ***external-memory (EM)*** model [1,52]. We present a dynamic set representation that outperforms previous representations: it allows for updates in $O(1)$ expected time, and supports the output-sensitive intersection of two sets of size $O(n)$ in expected time $o(n) + O(k)$ in the practical RAM model, where k is the number of elements in the intersection, and with related efficiency in the EM model. Further, we apply this data structure to the triangle listing problem.

Our solutions exploit a novel combination of two powerful algorithmic techniques. The first of these two techniques is an interesting variant of an algorithmic paradigm known as the ***power of two choices***, which has previously been used for load balancing and data structures [6, 39, 43, 44]. In a nutshell, the main idea of the power of two choices paradigm is to place items in one of two randomly chosen places so as to improve space or time performance. This paradigm is used, for example, in ***cuckoo***[1] hashing [20, 31, 47]. The second technique we exploit is ***filtering***, which involves storing data in compressed form using hash functions to support fast set-query operations. A well-known, classic example of this paradigm is the Bloom filter [10, 18, 22]. Instead of Bloom filters, however, we modify a more recent filtering structure known as the ***cuckoo filter***, which improves on Bloom filters in its space usage, locality of reference, and ease of updating [17, 20].

In this paper, we generalize the power-of-two-choices paradigm to a ***two-out-of-three*** paradigm. We explore how this idea leads to useful variations of cuckoo filters and hash tables, which we call ***2-3 cuckoo*** filters and hash tables. We use these structures to design improved algorithms for dynamic set intersection, which can in turn be used for answering triangle listing queries faster than with previous approaches, in the practical RAM and EM models. As with previous works, we characterize our algorithms in terms of multiple parameters, including the following:

- n: the size of a set or the number of vertices in a graph (depending on the context).
- m: the number of edges in a given graph, G.
- k: the number of elements to be output, e.g., the number of elements in the intersection of two sets or the number of triangles (i.e., cycles of length three) in an undirected graph.

[1]This name comes from the reproductive behavior of the female common cuckoo, which lays an egg in the nest of another bird after pushing out the other bird's egg.

PODS '17, May 14-19, 2017, Chicago, IL, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4198-1/17/05...$15.00
DOI: http://dx.doi.org/10.1145/3034786.3056115

- $\alpha(G)$: the ***arboricity*** [12] of a graph G, i.e., the minimum number of forests that cover all its edges. This parameter is within a factor of 2 of a related parameter, the ***degeneracy*** $d(G)$ [37], which is the maximum over subgraphs of G of the minimum degree of the subgraph.
- w: the number of bits in a word of memory.
- M: the number of words in internal memory.
- B: the number of words in a block of memory that can be transferred in one I/O between internal and external memory.

For more on the external-memory model, see the seminal work of Aggarwal and Vitter [1] and survey of Vitter [52].

1.1 Related Work

Miltersen [41] introduced the ***practical RAM*** model in 1996. This model is like the standard RAM model in that it has a single processor that can access any memory word in constant time. It also allows constant-time bit-parallel operations [4, 50], such as shifts, bitwise Boolean operations, and most-significant-bit finding, that are included in modern programming languages such as C, C++, Java, Python, and T-SQL. Exploiting such operations can speed up algorithms for large-scale problems on conventional hardware and software, including solutions for sorting and searching [4, 21, 50, 53].

Bille *et al.* [8] present a data structure that can compute the intersection of t sets of total size n in expected time $O(\lceil n(\log^2 w)/w\rceil + kt + \log w)$, where k is the size of the output. Kopelowitz *et al.* [33] introduce a dynamic set intersection data structure and use it to list the triangles in a graph G in $O(m\lceil(\alpha(G)\log^2 w)/w + \log w\rceil + k)$ expected time. Both of these papers work in the practical RAM model. The lower-order $O(\log w)$ terms in their analysis come from their reliance on a sophisticated bit-parallel list-merging algorithm of Albers and Hagerup [2]. The data structure of Kopelowitz *et al.* supports item insertion and deletion in $O(\log w)$ expected time and supports pairwise set intersection queries of sets of size $O(n)$ in expected time $O(\lceil n(\log^2 w)/w\rceil + k + \log w)$. The triangle listing algorithm of Kopelowitz *et al.* improves an $O(m\,\alpha(G))$ time algorithm by Chiba and Nishizeki [13] for triangle listing. Ortmann and Brandes [46] show that almost every previous internal-memory triangle listing algorithm can be implemented to run in $O(m\,\alpha(G))$ worst-case optimal time for any non-output-sensitive algorithm, or slightly worse (see their paper for an extensive survey). Hence, although it does not appear to be very practical, the method of Kopelowitz *et al.* [33] has the best asymptotic expected running time of any triangle listing algorithm prior to our work. Alternatively, Björklund *et al.* [9] study triangle listing from an asymptotic worst-case perspective, but do not characterize their running times in terms of graph parameters such as arboricity or degeneracy.

Amossen and Pagh [3] introduce a data structure, the "batmap," using the same two-out-of-three placement strategy that we do. However, they do not combine it with the filtering techniques that are essential for our improved dynamic set intersection and triangle listing algorithms. Moreover, they do not combine 2-3 cuckoo hashing with a stash, as we do, which allows us to significantly boost success probabilities.

Hu *et al.* [27, 28] study the I/O complexity of triangle listing in the EM model. They present an algorithm using $O(m^2/(MB) + \text{scan}(k))$ I/Os to list all k triangles in a graph of m edges, where $\text{scan}(n)$ is the I/O complexity for scanning an array of n elements.[2] In PODS 2014, Pagh and Silvestri [48] study the I/O complexity of the related ***triangle enumeration*** problem, where, rather than outputting all triangles in a graph, one is interested instead in calling an internal emit function for each triangle, which could, for example, be used to compute an aggregation function on all triangles in a graph.[3] Their method uses $O(m^{3/2}/(M^{1/2}B))$ expected I/Os. In PODS 2015, Hu *et al.* [26] present a triangle enumeration algorithm that uses $O(m^{3/2}/(M^{1/2}B))$ I/Os in the worst case. They also survey other previous work on the I/O complexity of triangle listing (please see their paper for additional references) and they show that the triangle listing algorithm of Dementiev [16] can be implemented to use $O(\text{sort}(m)\cdot\alpha(G))$ I/Os, where $\text{sort}(n)$ is the I/O complexity for sorting n elements.[4] Our approach is complementary to this prior work in PODS focusing on the I/O complexity of the triangle enumeration problem, in that we focus on the triangle listing problem.

1.2 Our Results

We present new dynamic set intersection data structures based on cuckoo filtering and two-out-of-three hashing. Our structure uses both cuckoo hash tables, which store full items, and ***cuckoo filters*** [17,20], which instead store smaller fingerprints of the items. We use a "stash" to improve the success likelihood, and use parallel pairs of 2-3 cuckoo filters and hash tables to guide the rearrangement of fingerprints within the filter. Our data structures can maintain sets of size $O(n)$, support element insertions and deletions in $O(1)$ expected time, and perform set-intersection queries in $O(\lceil(n\log w)/w\rceil + k)$ expected time, where k is the number of output elements. This speeds up the set-intersection times for the data structure of Kopelowitz *et al.* [33] by a logarithmic factor. Our solution is also simpler.

As an application of our data structures, we solve the triangle listing problem for a graph G of n vertices and m edges in $O(m\lceil(\alpha(G)\log w)/w\rceil + k)$ expected time, where k is the number of triangles in G. This also improves upon the previous solution of Kopelowitz *et al.* [33] by a logarithmic factor. All of our algorithms are for the practical RAM model [41] and can be easily implemented using standard instructions used in modern programming languages, as we discuss in a section on experimental results.

Further, we show how to implement external-memory versions of cuckoo-filters and hash tables. We use these to solve triangle listing in external memory with an expected $O(\text{sort}(n\,\alpha(G)) + \text{sort}(m\lceil(\alpha(G)\log w)/w\rceil) + \text{sort}(k))$ number of I/Os. This improves previous external-memory solutions for triangle listing in sparse graphs with small internal memory in scenarios where the output is asymptotically less than its worst case. Our external-memory solution, therefore, is well suited for large naturally sparse graphs that typically arise in practice (e.g., see [19, 24]).

[2] $\text{scan}(n) = O(n/B)$.

[3] Note that the I/O complexity of triangle listing is always at least that for triangle enumeration.

[4] $\text{sort}(n) = \Theta((n/B)\log_{M/B}(n/B))$ [1].

2. 2-3 CUCKOO HASH-FILTERS

In this section, we describe our 2-3 cuckoo hashing data structures, explaining how they can be used to maintain sets of size $O(n)$ to support insertions, deletions, and set intersection queries in internal memory. In subsequent sections, we describe how to use these for set intersection and triangle listing queries and how to adapt them for external memory.

Suppose, then, we wish to maintain a collection of sets, $\mathcal{S} = \{S_1, S_2, \ldots, S_l\}$, such that each set is of size at most $O(n)$, taken from a universe such that each element can be stored in a single memory word (which itself could be a unique pointer to a larger data file). For each S_i, we construct the following (see Figure 1):

- A hash table T_i of size $O(n)$, using three random hash functions h_1, h_2, and h_3, which map elements of S_i to triples of distinct integers in the range $[0, n-1]$. (We assume that our hash functions are perfectly random for our theoretical results, although our experiments show realistic hash functions perform similarly. See [5, 15] for more on this issue.) Each element x in S_i is stored, if possible, in two of the three possible locations for x based on these hash functions. We refer to each T_i as a ***2-3 cuckoo hash table***. Each table, T_i, also has a stash cache [31], C_i, of size λ_i, where λ_i is constant w.h.p. C_i stores elements for which it was not possible to store properly in two distinct locations in T_i (e.g., due to collisions with other elements). We maintain each C_i as a sorted linked list or growable hash table of size $O(\lambda_i)$, so that it is possible to insert and delete elements into/from any C_i in $O(1)$ amortized time and to list the elements in any C_i in $O(\lambda_i)$ time.

- A table, F_i, having $O(n)$ cells, that parallels T_i, so that $F_i[j]$ stores a non-zero fingerprint digest, $f(x)$, for an element x, if and only if $T_i[j]$ stores a copy of x. The digest $f(x)$ is a non-zero random hash function comprising δ bits, where $\delta = \Theta(\log w)$ is a parameter whose value is set to achieve a good false positive rate (e.g., $\delta = 2\log w$). The table F_i is called a 2-3 ***cuckoo filter***, and it is stored in a packed format, so that we store $O(w/\log w)$ cells of F_i per memory word. In addition to the vector F_i, we store a bit-mask, M_i, that is the same size as F_i and has all 1s in the corresponding cell of each cell of F_i that is occupied. The total space (in words) needed for F_i and M_i is $O(\lceil n(\log w)/w \rceil)$.

We assume we can read and write individual cells of F_i and M_i in $O(1)$ time. These cells amount to subfields of words of $O(\log w)$ size, which can be read from or written to using standard bit-level operations, such as AND, OR, XOR, and shift, built into modern programming languages, like C, C++, Java, Python, and T-SQL. We omit the details. Any read and write operation takes $O(1)$ such operations; hence, we assume that reading and writing individual cells of F_i takes $O(1)$ time in the practical RAM model [41].

Let us next describe how to perform insertions and deletions in pairs of parallel cuckoo filters and hash tables. We begin with deletion, because it is the easiest.

2.1 Deletion

Suppose we are given an element x, to be deleted from a 2-3 cuckoo hash-filter structure for a set S_i. To perform this operation, we first search for x in the stash, C_i. This takes $O(\lambda_i)$ time, which is constant w.h.p. If $x \in C_i$, then we remove it from C_i and we are done, since in this case, x is not stored in the cells of T_i or F_i. Otherwise, if $x \notin C_i$, then we search for x in $T_i[h_1(x)]$, $T_i[h_2(x)]$, and $T_i[h_3(x)]$. We remove x from each of these cells in which it is stored. For each such index, a_x, because F_i is parallel to T_i, we then zero-out the cell $F_i[a_x]$, and we zero-out the corresponding bit-mask cell, $M_i[a_x]$, to show this cell as empty. This step takes $O(\lambda_i + 1)$ time in the practical RAM model [41].

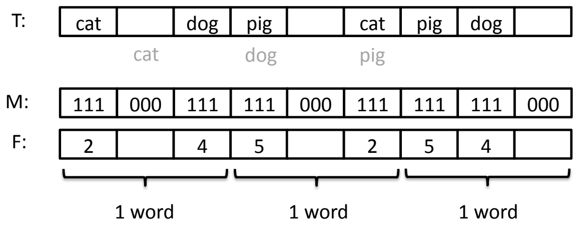

Figure 1: A simplified example of a parallel 2-3 cuckoo hash table, T, and filter, F, including the bit-mask, M. In this example, we are representing the set, $S = \{\mathsf{cat}, \mathsf{dog}, \mathsf{pig}\}$, in an instance of the practical RAM model that can store three fingerprint values per word. In this case, the filters are defined so $f(\mathsf{cat}) = 2$, $f(\mathsf{dog}) = 4$, and $f(\mathsf{pig}) = 5$. Note: every element is stored in two out of three locations; we are showing the third potential location for each element below its third location in grey. In this figure, we are not showing the stash cache, C, which in this simplified example would be empty. © Michael Goodrich.

2.2 Insertion

Suppose we are given an element, x, to be inserted into some set S_i. We first attempt to insert x into the 2-3 cuckoo table, T_i, performing a 2-3 cuckoo (two-out-of-three) insertion for x. Following [3], let us consider this as our inserting two instances of x into T_i via a one-out-of-three cuckoo insertion. Each element x has three possible locations, $T_i[h_1(x)]$, $T_i[h_2(x)]$, and $T_i[h_3(x)]$, where it may be stored. If one of these is empty, then we add x to it, completing that instance of inserting x. (If this is the second insertion for x, then we are completely done). If none of these three cells is empty, we choose one of them at random and add x to it, evicting its previous occupant y. We add y to a temporary buffer queue, Q. When we add x to cell $T_i[j]$ we zero out cell $F_i[j]$ and insert $f(x)$ into that cell, and we also perform an OR operation that sets the corresponding mask cell, $M_i[j]$, to all 1s, to show this cell as occupied.

We then process the buffer, Q, while it is non-empty. We take the next element, y, from Q. We then read the cells, $T_i[h_1(y)]$, $T_i[h_2(y)]$, and $T_i[h_3(y)]$. One of these cells may already store y; in this case, we consider the other two. If one of these two is empty, we add y to it, and we are done with y. If both of these cells are occupied, however, we choose one of them at random, evict its previous occupant z, and insert y into that cell. Then we add z to Q. Each time we add x to a cell $T_i[j]$ we zero-out the cell $F_i[j]$ and insert $f(x)$ into that cell (the mask cell $M_i[j]$ is already all 1s in this case). We repeat this processing of Q until we either succeed in emptying all the elements in Q or we reach a ***stopping condition***, which is defined to be the condition that we have spent more than L iterations processing Q during this insertion, where L is a threshold parameter set in the analysis. We then perform a deletion operation in T_i and F_i for each element in Q, and we add each such element to the stash, C_i. This step takes $O(L+1)$ steps in the practical RAM model (possibly amortized, if we are implementing C_i as a standard growable hash table).

As we show in Section 3, we can set L to be logarithmic in n and set a constant threshold $s > 0$ such that, with high probability throughout the process, $\lambda_i \leq s$. Thus, with high probability, our 2-3 cuckoo hash filters has stashes of constant size. This property provides a "safety net" for our data structures: with low probability, they could always fall back to a standard hash table or sorted linked list to store their sets, but with high probability, our structures will be much faster than this.

3. ANALYSIS

In this section, we give an analysis of 2-3 cuckoo hashing with a stash, which forms the theoretical foundation for our set intersection and triangle listing algorithms.

3.1 A Review of Standard Cuckoo Hashing

We now review useful characterizations and results for standard cuckoo hashing, which are good to know for when we generalize them to 2-3 cuckoo hashing.

We begin by recalling that in (standard) cuckoo hashing, as proposed by Pagh and Rodler [48], one inserts n items into a table T with $2(1+\epsilon)n$ buckets (for a fixed constant $\epsilon > 0$), via hash functions h_1 and h_2 whose values are always distinct. (Cuckoo hashing is often instead described using two separate tables for h_1 and h_2, avoiding the need for distinctness, but this makes little difference to its analysis.) Each bucket can store at most one item. To insert an item x, we place it in $T[h_1(x)]$ if that bucket is empty. Otherwise, we ***evict*** the item y in $T[h_1(x)]$, replace it with x, and attempt to insert y into its other cell $T[h_1(y)+h_2(y)-h_1(x)]$. If that location is free, then we are done. If not, we evict the item z in that location, attempt to insert z into its other cell, and so on. We can view the hash functions as defining a random multigraph with m vertices corresponding to the buckets in T, with each of the n items x yielding an edge $(h_1(x), h_2(x))$. The insertion procedure successfully places all n items if and only if each connected component in this ***cuckoo graph*** has at most one cycle. With high probability all connected components have at most cycle and no component is of size larger than $O(\log n)$. In what follows we allow the eviction process to occur up to $C \log n$ times for some constant C before declaring a failure in the placement.

There is, however, a small probability that an insertion cannot be done or takes longer time than expected. In such cases, rather than failing, unplaced items can be placed in a small additional memory, called a *stash* cache (e.g., similar to how we describe above for 2-3 cuckoo hash tables). The main previous result for stashing in standard cuckoo hash tables is the following theorem by Kirsch *et al.* [31].

THEOREM 1. *For any constant integer $s \geq 1$, for a sufficiently large constant C, the size S of the stash in a standard cuckoo hash table after all items have been inserted satisfies* $\Pr(S \geq s) = O(n^{-s})$.

In fact, the result of Theorem 1 can be extended somewhat beyond constant s with some care; for $s = O(\log\log n)$, for example, we have $\Pr(S \geq s) = \tilde{O}(n^{-s+1})$ [31]. Our main goal here is to extend such results on stashes to the setting of 2-3 cuckoo hashing.

3.2 Modeling 2-3 Cuckoo Hashing

We start our analysis of 2-3 cuckoo hashing by reviewing known results. In 2-3 cuckoo hashing, we insert n items into a table T via three hash functions h_1, h_2, and h_3 with distinct values. Each item is stored at two of the three locations given by its hash values, except that an item that cannot be stored successfully may be placed into a stash. In this setting, we require the table to have $6(1+\epsilon)n$ total bucket spaces, so that the final "load" of the hash table is less than $1/6$. (This contrasts with the standard cuckoo hash table, where it suffices that the final load is less than $1/2$.) It is known that a load strictly less than $1/6$ allows all items to be placed with high probability, but a load strictly greater than $1/6$ will fail to place all items with high probability. These results are discussed and proven by Amossen and Pagh [3] and Loh and Pagh [38] (see also the related combinatorial results in [30]). Indeed, they also show the following results for 2-3 cuckoo hashing, which are similar to the known results for standard cuckoo hashing:

- A natural insertion procedure succeeds in placing all items with high probability.
- This insertion procedure has constant expected time.
- With high probability the maximum insertion time is $O(\log n)$.

These results are derived by considering the corresponding *cuckoo hypergraph*, where each bucket is represented by a vertex and each item is represented by a hyperedge attached to the three vertices (buckets) that are the chosen locations for that item. The following subsection defines these hypergraphs more formally, including what it means for them to be acyclic or unicyclic (corresponding to trees or pseudotrees in ordinary graphs), and what their connected components are. The previous results show that the insertion procedure succeeds exactly when every component of this hypergraph is acyclic or unicyclic, and the time to insert an item is proportional to the size of its component. When n hyperedges are randomly selected for $6(1+\epsilon)n$ vertices, all hypergraph components are acyclic or unicyclic with high probability; all components are of size $O(\log n/\epsilon^2)$ with high probability; and the average size of the component containing any given vertex is constant.

Our result shows how to improve these results by using a stash, similar to how stashes improve the success probabilities for standard cuckoo hashing. For convenience, we prove a slightly easier and weaker version of the result that is sufficient for our purposes. Specifically, we prove the following:

THEOREM 2. *For any constant integer $s \geq 1$, for a sufficiently large constant C, the size S of the stash in a 2-3 cuckoo hash table after all items have been inserted satisfies* $\Pr(S \geq s) = \tilde{O}(n^{-s})$.

The $\tilde{O}$ notation allows for extra polylogarithmic factors. These do not appear to actually be necessary; we believe

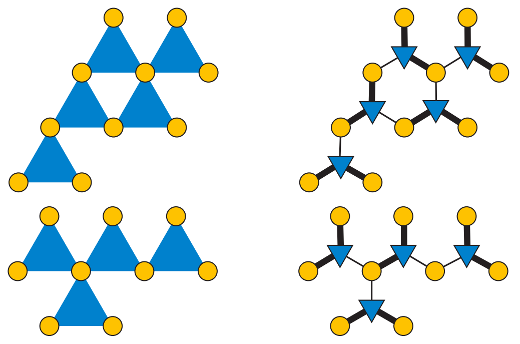

Figure 2: A 3-uniform hypergraph (left) with a unicyclic component (top) and an acyclic component (bottom), and a 2-assignment in its incidence graph (right, with the edges of the assignment shown using thick edges). © David Eppstein.

with more work the bound of Theorem 1 applies here as well. The proof, however, is simpler by allowing these factors, and the resulting bound is sufficient for our purposes.

Our proof follows the approach of Kirsch *et al.* [31], suitably modified for this situation, including a number of interesting changes that come from translating the previous results for cuckoo graphs to cuckoo hypergraphs.

3.3 Formalization of Acyclic Hypergraphs and Related Concepts

Before proving our result on stashes, this subsection provides a basis for thinking about acyclic hypergraphs and unicyclic hypergraphs more formally in this context, as there are various ways they can be defined.

A 3-uniform hypergraph $H = (V, T)$ consists of a set V of vertices and a set or multiset T of triples of vertices, the hyperedges of the hypergraph. It can be represented by a bipartite ***incidence graph*** (V, T, I), a graph that has $V \cup T$ as its vertices and the set of vertex-hyperedge incidences $I = \{(v, t) \mid v \in V, t \in T, v \in t\}$ as its edges. Even if the hypergraph has repeated pairs or triples among its vertices, the incidence graph is a simple graph. Following Berge [7], we call a hypergraph ***acyclic*** if its incidence graph is acyclic as an undirected graph. The ***connected components*** of a hypergraph are the subsets of vertices and hyperedges corresponding to connected components of the incidence graph. A hypergraph is ***connected*** if it has exactly one connected component. We define acyclicity of components in the same way as acyclicity of the whole hypergraph. We say that a connected component of the hypergraph is ***unicyclic*** if the corresponding connected component of the incidence graph is unicyclic: that is, that it has exactly one undirected cycle. For examples of these concepts, see Figure 2.

If a 3-uniform hypergraph is connected and acyclic, then its incidence graph must be a tree. Each hyperedge forms an interior vertex of degree three, so all leaves of the tree must be vertices of the hypergraph. We define a ***twig*** of this tree to be a hyperedge that is incident to exactly two leaves. If we remove the hyperedge and its two leaves, we are left with a smaller hypergraph that remains connected and acyclic.

Lemma 1. *Every connected acyclic 3-uniform hypergraph with more than one hyperedge has at least two twigs.*

Proof. Apply the standard result that every nontrivial tree has at least two leaves to the subtree of the incidence graph formed by removing all its leaves. □

It follows by induction (repeatedly removing one twig, with a one-edge hypergraph as base case) that a connected acyclic 3-uniform hypergraph with $n > 1$ vertices must have n odd and exactly $(n-1)/2$ hyperedges. Similarly, a connected unicyclic 3-uniform hypergraph must have an incidence graph in the form of a ***pseudotree***, a graph formed by adding one edge to a tree. We can modify this pseudotree into a tree by replacing this edge of the incidence graph by a different edge from the same hyperedge to a new vertex. It follows that, in a connected unicyclic 3-uniform hypergraph with n vertices, n must be even and there must be exactly $n/2$ hyperedges. If a connected 3-uniform hypergraph with n vertices is neither acyclic nor unicyclic, then any spanning tree of its incidence graph omits more than one edge, and it must have more than $n/2$ hyperedges.

We define a ***2-assignment*** of a 3-uniform hypergraph (V, T) to be a subset A of incidence graph edges such that each vertex of V is incident to at most one edge in A and each hyperedge in T is incident to exactly two edges in A. An example is shown by the thick edges on the right of Figure 2. We say that a 2-assignment A ***omits*** a vertex v if there is no edge in A incident to v.

Lemma 2. *For any connected acyclic 3-uniform hypergraph (V, T), and any vertex $v \in V$, there is a 2-assignment of (V, T) that omits v.*

Proof. By induction on the number n of vertices in (V, T). As a base case, the result is true when there is one vertex and no hyperedge, or when there are three vertices and one hyperedge. Otherwise, by Lemma 1 there exists a twig t neither of whose incident leaves is v. By induction, the hypergraph formed by removing this twig and its leaves has a 2-assignment A omitting v. This can be augmented to a 2-assignment of (V, T) by adding to A the two edges of the incidence graph that connect t to its two leaves. □

Lemma 3. *Every connected unicyclic 3-uniform hypergraph (V, T) has a 2-assignment.*

Proof. Decompose the incidence graph of (V, T) into a tree and an added edge (v, t) with $v \in V$ and $t \in T$. Let w be a new vertex, not in V, and consider the 3-uniform hypergraph (V', T') where $V = V \cup \{w\}$ and T' is formed by replacing t by a new triple with w in the place of v. Then (V', T') is a connected acyclic hypergraph and by the previous lemma it has a 2-assignment A that omits w. Then A also forms a valid 2-assignment for (V, T). □

Lemma 4. *Any 3-uniform hypergraph has a 2-assignment if and only if each of its connected components is acyclic or unicyclic.*

Proof. A 2-assignment can be found separately within each of the connected components, if it exists at all. By the previous two lemmas, it does exist within components that are acyclic or unicyclic. However, if a component with k vertices is neither acyclic nor unicyclic, then it has more than $k/2$ hyperedges, and any 2-assignment would necessarily include more than k edges of the incidence graph. This is incompatible with the requirement that a 2-assignment can only have one edge incident to each vertex. □

3.4 Stashes for 2-3 Cuckoo Hashing

We now prove Theorem 2 through a sequence of lemmas.

In analyzing the stash for standard cuckoo hashing, the approach of Kirsch *et al.* [31] begins by noting that the cyclomatic number of a graph G, denoted by $\gamma(G)$, is defined to be the smallest number of graph edges which should be removed from G so that there are no cycles. It is well known that if G is connected and has $v + k$ edges, then $\gamma(G) = k + 1$. As explained by Kirsch *et al.* [31], if $c(G)$ is the number of cyclic components in G, the size of the stash equals $\gamma(G) - c(G)$.

For 2-3 cuckoo hashing, our corresponding "random graph" H consists not of random edges, but instead random triangles in a 3-uniform hypergraph. A placement in the 2-3 cuckoo hash table corresponds to a 2-assignment, or equivalently to a matching in a graph with a triangle for each hyperedge such that each triangle has a matched edge. As shown above, a 2-assignment can be found in a 3-uniform hypergraph if and only if each of its connected components is acyclic or unicyclic. Following the analysis for standard cuckoo hashing, we now let the cyclomatic number $\alpha(H)$ be the smallest number of triangles which should be removed from a 3-uniform hypergraph H in order to make H become acyclic. We have $\alpha(H) = \sum_h \alpha(h)$ where the sum is understood to be over components h of H, and the size of the stash required for 2-3 cuckoo hashing is $\alpha(H) - c(H)$, where $c(H)$ is the number of cyclic components in H. (Subtracting $c(H)$ accounts for the fact that we can tolerate components that are unicyclic instead of acyclic.)

To characterize the use of stashes in 2-3 cuckoo hashing, we follow the argument of Kirsch *et al.* [31] for stashes in standard cuckoo hashing, simplified somewhat so that we can extend the analysis to deal with random hypergraphs (albeit accordingly achieve slightly worse bounds, with additional polylogarithmic terms, due to this simplification). Also, following Kirsch *et al.*, we work in the setting where our table is split, so there are three subtables each with m edges, and the hash functions choose one vertex from each subtable. Again, this setting is essentially equivalent to requiring distinct hash values—as long as the load is less than 1/6, the 2-3 cuckoo table will function suitably. We begin with some useful lemmas, well known for the graph case, but that apply here as well.

Lemma 5. *Consider a random triangle graph (3-uniform hypergraph) where the number of triangles (hyperedges) included is a Poisson random variable with mean λ. Conditioned on the number of triangles being at least n, the cyclomatic number of of the Poisson hypergraph stochastically dominates the cyclomatic number of a random hypergraph chosen with exactly n triangles.*

This follows readily from the fact that the cyclomatic number is stochastically increasing in the number of triangles, and after condtioning on the number of triangles in a random triangle graph with a Poisson distributed number of triangles, the resulting graph is a random triangle graph with that number of triangles. The lemma as stated is meant to hold for the case where triangles may be chosen more than once, but it also holds as well for the case where the random triangles chosen are required to be distinct.

Lemma 6. *Let $\lambda = (1 + \epsilon')n$ for some constant $\epsilon' > 0$. Then with probability $e^{-\Omega(n)}$, the value of a Poisson random variable with mean λ is at least n.*

This is a simple application of Chernoff bounds.

We choose $\lambda = (1+\epsilon')n$ so that if the total number of cells in our 2-3 table is $3m = 6(1 + \epsilon)n$, we have $m(1 - \epsilon') > 2\lambda$. This ensures that with all but exponentially small probability the number of triangles is at least n but still less than $\lambda(1 + \epsilon') < m/2$.

We now bound the probability that the required size of the stash, which we denote by S, is at least s as follows. Let H be a random triangle hypergraph chosen with a Poisson number of triangles as above. later. Let C_v be the component containing v in the randomly chosen hypergraph, and let E_v represent the set of edges in C_v. Let C be a constant to be determined Then

$$\Pr(S \geq s) \quad \leq \quad \Pr(\max_v |E_v| > C \log n) + \Pr(\alpha(H) \geq s \mid \max_v |E_v| \leq C \log n) + e^{-\Omega(n)}.$$

Here the first term corresponds to the probability that some component is too large, and the last term corresponds to the case where the random triangle hypergraph has fewer than n triangles or more than $\lambda(1 + \epsilon)$ triangles.

We first bound the probability of a large component.

Lemma 7. *There exists a constant $\beta \in (0, 1)$ such that for any fixed vertex v and integer $k > 0$,*

$$\Pr(|E_v| \geq k) \leq \beta^k.$$

Proof. We picture doing a breadth first search starting from v. We overcount by assuming each additional edge we explore in the breadth first search provides two new vertices. Let $X_1, X_2, \ldots$ be independent binomial random variables $\text{Bin}(m^2, (1+\epsilon')n/m^3)$. Then the number of new edges added to the component as we explore the ith vertex in the breadth first search is stochastically dominated by X_i (as there are at most m^2 possible choices for another pair of vertices, and each of the m^3 edges appears independently with probability at most $(1 + \epsilon')n/m^3$), and the number of new vertices is stochastically dominated by $2X_i$. For there to be more than k edges in the component, as we explore the first $2k + 1$ vertices, we must generate more than $k + 1$ edges, or the breadth first search process will terminate. Hence,

$$\Pr(|E_v| \geq 2k + 1) \leq \Pr\left(\sum_{i=1}^{2k+1} X_i \geq k + 1\right),$$

which is equal to

$$\Pr\left(\text{Bin}\left((2k + 1)m^2, (1 + \epsilon')n/m^3\right) \geq k + 1\right).$$

We have $(2k+1)(1+\epsilon')n/m \leq (k+1)/(1+\gamma)$ for a constant γ; if $\gamma \leq 1$, then applying a standard Chernoff bounds the above probability by

$$e^{-(1+\gamma)^2(k+1)/3} \leq \beta^k,$$

for $\beta = e^{-(1+\gamma)^2/3}$. (For $\gamma > 1$, we note that the bound for when $\gamma = 1$ would hold as well.) $\square$

Lemma 7 already implies that for any constant ζ, for a sufficiently large constant C, all components are at most $C \log n$ with probability at most $n^{-\zeta-1}$.

We next bound the probability of a single component having a large cyclomatic number.

LEMMA 8. *For every vertex v and $t, k \geq 1$, $k \leq m^{1/3}$,*

$$\Pr(\alpha(C_v) \geq a \mid |E_v| \leq k) \leq 2\left(\frac{126e^5k^3}{m}\right)^a,$$

PROOF. In what follows we begin by ignoring duplicate edges; we return to them at the end of the proof.

We reveal the edges in C_v following a breadth-first search starting at v. Let us assume that the number of edges in our breadth first search stays below k; otherwise, it will not matter how the process continues. Note that we may therefore assume at any point of our breadth first search there are at most $3k$ vertices in the component.

Equivalently, we can consider the process up through the first k edges. We let $K(u)$ be the number edges that, when we first connect to u, add to cyclomatic number of C_v. This can occur because u is involved with an edge that contains two vertices already found in the breadth first search, or because two (or more) distinct edges from vertices already in the component connect to u. We therefore see that $K(u)$ is stochastically dominated by the sum of two random variables. The first is a Poisson random variable $X_u = \text{Po}(\lambda\binom{3k}{2}/m^3)$ for the first case. For the second case, consider a distribution $Y_u = \max \text{Po}(3\lambda k/m^2) - 1, 0$; then the contribution to the cyclomatic number due to two (or more) distinct edges from vertices already in the component connecting to u is stochastically dominated by the distribution L_u.

As we go through at most $3k$ steps of our breadth first search, each step has $2m$ possible vertices that can be added. Let $X_1, X_2, \ldots, X_{6mk}$ and $Y_1, Y_2, \ldots, Y_{6mk}$ be independent copies of the random variables X_u and Y_u described above. Thus the contribution to the cyclomatic number of C_v is stochastically dominated by $\sum_{i=1}^{6mk} X_i + \sum_{i=1}^{6mk} Y_i$. The first summation corresponds to a Poisson random variable of mean less than $27k^3\lambda/m^2$. To bound $\sum_{i=1}^{6mk} Y_i$, note that the probability that any individual Y_i is greater than 0 is at most the mean squared, $9\lambda^2k^2/m^4$. Let Z be the number of non-zero Y_i in $\sum_{i=1}^{k} Y_i$. Then the probability,

$$\Pr(\sum_{i=1}^{6mk} Y_i \geq a)$$

is at most

$$\Pr(Z > a) + \sum_{\ell=1}^{a} \Pr(Z \geq \ell) \cdot \Pr(\sum_{i=1}^{6mk} Y_i \geq a \mid Z = \ell).$$

We note

$$\begin{aligned}
\Pr(Z \geq \ell) &\leq \binom{mk}{\ell}\left(\frac{9\lambda^2k^2}{m^4}\right)^\ell \\
&\leq \left(\frac{mke}{\ell}\right)^\ell \left(\frac{9\lambda^2k^2}{m^4}\right)^\ell \\
&= \left(\frac{9e\lambda^2k^3}{m^3\ell}\right)^\ell.
\end{aligned}$$

Also, let Y_i' be a (the) Poisson random variable $\text{Po}(3\lambda k/m^2)$ associated with Y_i. Then a straightforward calculation gives:

$$\Pr(Y_i = j+1 \mid Y_i > 0) \leq \frac{2m^4}{\lambda^2k^2}\Pr(Y_i' = j+2).$$

We then have that

$$\Pr\left(\sum_{i=1}^{6mk} Y_i \geq a \mid Z = \ell\right)$$

is equal to

$$\begin{aligned}
&\sum_{j_1,\ldots,j_\ell; \sum_{i=1}^{\ell} j_i \geq a-\ell} \prod_{i=1} \ell \Pr(Y_i = j_i + 1 \mid Y_i > 0) \\
\leq\ & \left(\frac{2m^4}{\lambda^2k^2}\right)^\ell \sum_{j_1,\ldots,j_\ell; \sum_{i=1}^{\ell} j_i \geq a-\ell} \prod_{i=1} \ell \Pr(Y_i' = j_i + 2) \\
\leq\ & \left(\frac{2m^4}{\lambda^2k^2}\right)^\ell \Pr\left(\sum_{i=1}^{\ell} Y_i' \geq a + \ell\right) \\
=\ & \left(\frac{2m^4}{\lambda^2k^2}\right)^\ell \Pr\left(\text{Po}(3\lambda k\ell/m^2) \geq a + \ell\right) \\
\leq\ & \left(\frac{2m^4}{\lambda^2k^2}\right)^\ell \left(\frac{3ek\ell\lambda}{m^2(a+\ell)}\right)^{a+\ell} \\
=\ & \left(\frac{\lambda}{m^2}\right)^{a-\ell}\left(\frac{6e^3\ell}{k(a+\ell)}\right)^\ell\left(\frac{3ek\ell}{a+\ell}\right)^a.
\end{aligned}$$

Here we use a standard tail bound on a Poisson random variable (Theorem 5.4 of [45]).

It follows from the above bounds that the probability

$$\Pr\left(\sum_{i=1}^{6mk} Y_i \geq a\right)$$

is at most

$$\begin{aligned}
&\left(\tfrac{9e\lambda^2k^3}{m^3a}\right)^a + \textstyle\sum_{\ell=1}^{a}\left(\tfrac{9e\lambda^2k^3}{m^3\ell}\right)^\ell\left(\tfrac{\lambda}{m^2}\right)^{a-\ell}\left(\tfrac{6e^3\ell}{k(a+\ell)}\right)^\ell\left(\tfrac{3ek\ell}{a+\ell}\right)^a \\
\leq\ & \left(\tfrac{9e\lambda^2k^2}{m^3a}\right)^a + \left(\tfrac{3e\lambda k}{m^2}\right)^a \textstyle\sum_{\ell=1}^{a}\left(\tfrac{54e^4\lambda k^2}{m(a+\ell)}\right)^\ell\left(\tfrac{\ell}{(a+\ell)}\right)^\ell \\
\leq\ & \left(\tfrac{9e\lambda^2k^2}{m^3a}\right)^a + \left(\tfrac{3e\lambda k}{m^2}\right)^a\left(54e^4k^2\right)^a \\
\leq\ & \left(\tfrac{63e^5k^3}{m}\right)^a.
\end{aligned}$$

Finally, we require $\Pr(\sum_{i=1}^{6mk} X_i + \sum_{i=1}^{6mk} Y_i \geq a)$. The first summation is a Poisson random variable wth mean less than $63e^5k^3/m$. Some additional calculations give

$$\Pr\left(\sum_{i=1}^{6mk} X_i + \sum_{i=1}^{6mk} Y_i \geq a\right) \leq \left(\frac{126e^5k^3}{m}\right)^a.$$

The above assumed no duplicate edges. The number of duplicate edges in total, however, is stochastically dominated by a binomial random variable $\text{Bin}(n, (1+\epsilon')n/m^3)$. The probability of j duplicate edges is easily seen to be $O(n^{-j})$ and our additional factor of two in the lemma statement can account for the possiblity of duplicate edges. □

We note that in our analysis above, we are conditioning on the largest component size being at most logarithmic in n. The above shows that for any single vertex, the probability that it is in a component that requires putting an element in a stash, which corresponds to $\Pr(\sum_{i=1}^{6mk} X_i + \sum_{i=1}^{6mk} Y_i \geq 2)$, is $\tilde{O}(1/n^2)$. As there are $3m$ vertices total, we have shown the probability a stash is used at all is $\tilde{O}(1/n)$, and intuitively the path to show that the probability a stash of size s overflows is $\tilde{O}(n^{-s+1})$ follows readily.

We can formalize this by next bounding the probability of all components together having a large assignment number. We first note that the sum of the assignment numbers is bounded above by the sum of $3m$ independent copies of the random variable which gives the number of excess edges given to the assignment number for a component of a single vertex. (See, e.g., Lemma 2.10 of [31]; intuitively this follows from the fact that we need only count the excess once for each component, and as each component's vertices can be handled separately, the distinction of components reduces the possible excess for other vertices.) Let $V_i = \alpha(C_{v_i})$, the cyclomatic number of the ith vertes. Then for any constant s and for sufficiently large n, we have for some constants $c_1, c_2 > 0$,

$$\begin{aligned}
& \Pr(\alpha(H) - c(H) \geq s) \\
\leq\ & \Pr\left(\sum_{i=1}^{6m} V_i \geq s + |\{i\ :\ V_i \geq 1\}|\right) \\
\leq\ & \sum_{\{j_1,\ldots,j_{6m}\ :\ \sum_{i=1}^{6m} j_i = s\}} \prod_{\{i\ :\ j_i \geq 1\}} \Pr(V_i \geq j_i + 1) \\
\leq\ & \sum_{\{j_1,\ldots,j_{6m}\ :\ \sum_{i=1}^{6m} j_i = s\}} \prod_{\{i\ :\ j_i \geq 1\}} c_1 n^{-j_i} (\log n)^{c_2} \\
\leq\ & \sum_{\{j_1,\ldots,j_{6m}\ :\ \sum_{i=1}^{6m} j_i = s\}} c_1^s n^{-s-|\{i\ :\ j_i \geq 1\}|} (\log n)^{sc_2} \\
\leq\ & \sum_{k=1}^{s} \binom{6m}{k} s^k c_1^s n^{-s-k} (\log n)^{sc_2} \\
=\ & \tilde{O}(n^{-s}).
\end{aligned}$$

This gives the desired result on the stash size for a 2-3 cuckoo table, thereby proving Theorem 2.

4. SET INTERSECTION ALGORITHM

In this section, we describe our algorithm for performing set intersection queries.

Let S_i and S_j be two sets represented using our 2-3 cuckoo hash-filter pairs, as described above, and suppose we wish to answer a set-intersection query for S_i and S_j, to list out the elements in $S_i \cap S_j$.

We begin our set-intersection algorithm by computing a vector that identifies the matching non-empty cells in F_i and F_j. For example, we could compute the vector defined by the following bit-wise vector expression:

$$A = (M_i \text{ AND NOT } (F_i \text{ XOR } F_j)).$$

We view A as being a parallel vector to F_i and F_j. Note that a cell, $A[r]$, consists of δ bits and this cell is all 1s if and only if $F_i[r]$ stores a fingerprint digest for some element and $F_i[r] = F_j[r]$, since fingerprint digests are non-zero. Thus, we can create the list, L, of members of the common intersection of S_i and S_j, by visiting each word of A and storing to L the element in $T_i[r]$ corresponding to each cell, $A[r]$, that is all 1s, but doing so only after confirming that $T_i[r] = T_j[r]$. Doing the listing can be done in time $O(1 + k + t)$, where k is the number of elements in the intersection and t is the number of false positives, by using bit-level operations in the practical RAM model (e.g., see [17]). Then, the check to weed out false positives is a constant-time operation per element in the list, involving a lookup in the two cuckoo hash tables, to remove elements that map to the same locations and have the same fingerprint digests but are nevertheless different elements. That is, we remove from this list any elements, x and y, that happen to map to the same cell, r, in their respective 2-3 cuckoo hash tables and they also have the same fingerprint digest, that is, $f(x) = f(y)$. Note that by requiring $\delta > \log w$, we can guarantee that the probability of such false positives is at most $1/w$. Figure 3 illustrates a simplified example of this algorithm.

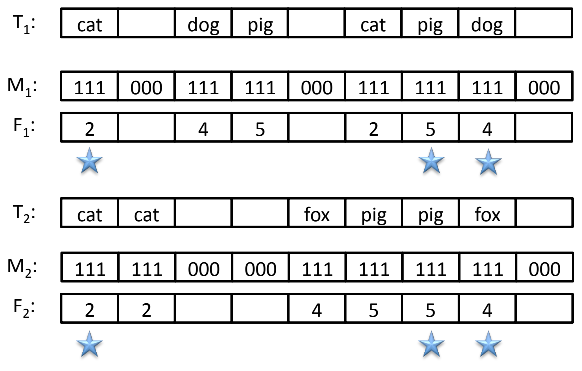

Figure 3: An example of parallel 2-3 cuckoo hash tables and filters for two sets. In this example, we are representing the set, $S_1 = \{\text{cat, dog, pig}\}$, and the set, $S_2 = \{\text{cat, pig, fox}\}$. The intersection algorithm does bit-parallel operations to find the matching non-zero cells in F_1 and F_2, and then checks the matches found using T_1 and T_2, to identify the set of intersecting items, {cat, pig}. In this case the filters are defined so $f(\text{cat}) = 2$, $f(\text{dog}) = 4$, $f(\text{pig}) = 5$, and $f(\text{fox}) = 4$. We show the matching filter locations with stars, including the false positive match for dog and fox, which would be culled when we check the potential matches against T_1 and T_2. In this figure, we are not showing the stash caches, C_1 and C_2, which in this simplified example would be empty. © Michael Goodrich.

Because we only include in L elements that are confirmed to belong to both S_i and S_j, we can guarantee that every member of L is a member of $S_i \cap S_j$. Nevertheless, we might still be missing some elements in this intersection. In particular, some elements might be stored in the stashes C_i and C_j, so we have additional work to do. For each x in C_i, we search for x at locations $T_j[h_1(x)]$ and $T_j[h_2(x)]$, and add x to L if it is found. Likewise, for each x in C_j, we perform a search for x at the locations $T_i[h_1(x)]$ and $T_i[h_2(x)]$, and add x to L if it is found. Finally, for each x in C_i, we do a search for x in C_j, adding it to L if found. If we let λ_i denote the size of C_i and if we let λ_j denote the size of C_j, then the time needed for this extra work to complete our algorithm is $O(\lambda_i + \lambda_j)$.

The running time of our entire algorithm for computing the intersection of S_i and S_j, therefore, is

$$O(n(\log w)/w + k + t + \lambda_i + \lambda_j),$$

where k is the size of the intersection (i.e., the output) and t is the number of false positives. As we have shown in the previous section, λ_i and λ_j are $O(1)$ with high probability. Thus, the expected running time for our set intersection algorithm is at most

$$O(\lceil n(\log w)/w \rceil + k + t).$$

In addition, by choosing $\delta > \log w$, we can bound the probability that two different elements have the same fingerprint value as being at most $1/2w$. Thus, $\mathbf{E}[t] \le n/w$, since each element is stored in at most two places in a 2-3 cuckoo hash table. Therefore, we have the following.

THEOREM 3. *Let S_i and S_j be two sets of size $O(n)$, represented by using parallel 2-3 cuckoo filters and hash tables. Then we can compute the intersection $S_i \cap S_j$ in $O(\lceil n(\log w)/w \rceil + k)$ expected time, where k is the size of the intersection.*

5. TRIANGLE LISTING

In this section, we describe how to use 2-3 cuckoo filters to list the triangles in an undirected graph G.

We begin by computing a ***degeneracy ordering*** of G, an order of the vertices as $(v_1, \ldots, v_n)$ such that each vertex v_i has at most d edges to vertices v_j with $i < j$, for a suitable value d. The smallest possible d for such an ordering is the ***degeneracy*** [37] of G, which satisfies $d \le 2\alpha(G)$. Such a degeneracy ordering can be found in time $O(n + m)$ by a simple greedy algorithm (e.g., see [40]) that repeatedly chooses the next vertex in the ordering to be the one with the fewest edges to other not-yet-selected vertices. Given this ordering, note that any triangle in G can be written as (v_i, v_j, v_l) with $i < j < l$.

For each node v_i in G, we construct a 2-3 cuckoo hash filter (including stash caches, as needed) for the set S_i of adjacent vertices v_j with $i < j$, by inserting these vertices into an initially-empty cuckoo hash-filter structure. Note that the size of each S_i is at most d. Because of this fact, in order for our algorithm to work in the desired expected time bounds, we need to be a little more careful in how we choose the sizes of our 2-3 cuckoo hash filters.

By Theorem 2, the probability that a set, S_i, stored in a two-three cuckoo hash table of size $n_i = \Omega(|S_i|)$, can be represented using a stash of size at most s, is $\tilde{O}(n_i^{-s})$. Thus, we can choose a stash threshold size, λ, depending on s, so that this probability is at most $1/n_i^2$. In order to obtain the time bounds we desire, however, we need this failure probability to be at most $1/2w$ (even if $|S_i|$ is small). For this reason, our algorithm depends on the value of d as follows:

- If $d \ge w/\log w$, then we use our 2-3 cuckoo hash-tables (with sorted or growable hash-table stash caches) to have size $\Theta(d)$ and we choose λ to be a sufficiently large constant so that the failure probability for any stash growing past λ size in our representation of any given set is at most $1/2w$.
- If $d < w/\log w$, then we use our 2-3 cuckoo hash-tables (with sorted or growable hash-table stash caches), with parameters chosen for tables of size $\Theta(w/\log w)$, even though the actual sets to be intersected will be at most d. This oversizing of our hash tables improves their success probability, and the choice of table size allows intersections to be performed in expected time $O(1 + k_{i,j})$, where $k_{i,j}$ is the number of elements in the intersection, when no failure occurs. To keep the failure probability less than $1/2w$, we again set $\lambda = O(1)$.

If the stash for any set grows to be over size λ, then we say that the 2-3 cuckoo hash filter for the set is in an ***overflow*** state. Nevertheless, we still use our cuckoo hash-filter representation to compute intersections with such sets in the overflow state—we just have to be careful to account for the time needed for such intersection computations.

Our triangle listing algorithm, then, considers each edge (v_i, v_j) in G and performs a set-intersection query for S_i and S_j, using the algorithm given in Section 4. The output of all these $O(m)$ queries will give us a listing of all the triangles in G. We characterize the performance for this algorithm in the following theorem, including the time needed to process sets in an overflow state.

THEOREM 4. *Given a connected graph G with n vertices, m edges, arboricity $\alpha(G)$, and k triangles, we can list all triangles in G in $O(m\lceil(\alpha(G)\log w)/w\rceil + k)$ expected time in the practical RAM model.*

PROOF. If $d < w/\log w$, our desired expected time bound is itself bounded by $O(m + k)$, since d is $\Theta(\alpha(G))$. Furthermore, we would get this same desired expected time bound even if $d = w/\log w$. Note that when $d < w/\log w$ we expand our 2-3 cuckoo hash filters to be of size $\Theta(w/\log w)$; hence, without loss of generality, let us assume for the remainder of the proof that $d \ge w/\log w$.

By the algorithm and analysis in Section 4, for any edge (i, j), representing two sets, S_i and S_j, we can report the intersection, $S_i \cap S_j$, in time

$$O(\lceil(\alpha(G)\log w)/w\rceil + k_{i,j} + t_{i,j} + \lambda_i + \lambda_j)$$

where $k_{i,j}$ is the number of elements in their common intersection, $t_{i,j}$ is the number of false positives found in the computation of their common intersection, and λ_i and λ_j are the respective sizes of the stashes for S_i and S_j. Summing these time bounds across all the edges of G, then, gives us a running time that is $O(m\lceil(\alpha(G)\log w)/w\rceil + k)$ plus time proportional to

$$\sum_{(i,j)\in G} t_{i,j} + \lambda_i + \lambda_j.$$

By a similar analysis as in Section 4, and the facts that, for each i, $|S_i| \le 2\alpha(G)$, and we chose δ so that fingerprint collisions occur with probability at most $1/2w$,

$$\mathbf{E}[t_{i,j}] \le \alpha(G)/w,$$

for any edge (i, j) in G. Thus, by the linearity of expectation, the expected value of $\sum_{(i,j)\in G} t_{i,j}$ is $O(m\lceil\alpha(G)/w\rceil)$. For any i, clearly we have that $\lambda_i \le |S_i| \le 2\alpha(G)$. In addition, by Theorem 2 and how we built our 2-3 cuckoo hash tables to have capacity at least $w/\log w$, so that the probability λ_i is more than a constant (i.e., the 2-3 cuckoo hash-filter for S_i is in an overflow state) is at most $1/2w$, $\mathbf{E}[\lambda_i]$ is $O(1+\alpha(G)/w)$. Thus, by the linearity of expectation, the expected value of $\sum_{(i,j)\in G} \lambda_i + \lambda_j$ is $O(m\lceil\alpha(G)/w\rceil)$. Therefore, by another application of the linearity of expectation, the expected time to list all the triangles in G is $O(m\lceil(\alpha(G)\log w)/w\rceil + k)$. $\square$

6. EXPERIMENTS

Even though the contributions of this paper are primarily theoretical explorations of topics central to data management, we present some preliminary experimental results in this section. Our main goal in this preliminary empirical work is to test whether our algorithms are easy to implement and are competitive with the state of the art. Although we believe there are further optimizations available to improve the performance of our code, as we describe elsewhere in this section, our experiments provide evidence that 2-3 cuckoo hash filters are indeed simple and fast in practice.

For our experiments, we first present some simple empirical results that provide evidence of the practicality of 2-3 cuckoo hash filters and provide insight into the parameters involved in the definitions of 2-3 cuckoo hash filters. We then turn to empirical results for triangle listing, first considering performance on a collection of random graphs, and then on real-world networks. We compare our algorithm with the algorithm of Chiba and Nishizeki [13], using an implementation we constructed based on their description. This simple algorithm is known to have both good theoretical guarantees and good performance in practice, with asymptotic running times matching other existing triangle-listing algorithms (e.g., see Ortmann and Brandes [46]). We did not empirically compare our algorithm with the triangle-listing algorithm of Kopelowitz *et al.* [33], however, because, even though it provides an asymptotic improvement in theory, it appears too complicated for practice.

We ran our experiments on a machine with a 3.1 GHz Intel Core i7 CPU and 16 GB of RAM. Throughout the experiments, we set $w = 64$, as that is the native word-length on the machine used for the experiments. For our hash functions, we use random polynomial hash functions (e.g., see [25]) modulo an appropriate value to get values within the desired range. We set the number of bits used in the fingerprint digest, δ, to be 8 unless otherwise specified.

We begin by describing empirical results considering the average insertion time for a 2-3 cuckoo hash filter as we vary the size of the table. Recall the 2-3 cuckoo hash filter includes both the 2-3 cuckoo filter and the associated 2-3 cuckoo hash table. For our experiments we tested size values n = 1,000, 10,000, and 100,000. Recall that table sizes should be at least $6n$; hence, we vary the size of the table as cn for $c = 6.2, 6, 5, 6.8, 7.1, 7.4, 7.7, 8.0$. We chose to bound the number of iterations L that we perform during an insertion before putting an item in the stash at 200. Our experiment involved the insertion of a random permutation of the integers from 1 to n into a 2-3 cuckoo hash table. The average insertion time per element, averaging the results over 1,000 experiments, is presented in Figure 4. As might be expected, the insertion time decreases slightly as we increase the memory used, as then there are fewer collisions, but the decrease from larger table sizes is relatively small.

We also plot the average number of elements in the stashes at the end of the process against c in Figure 5, using the same setup as for the average insertion time. As can be seen, the average stash sizes are very small, as predicted by the theory in Section 3. Similarly in Figure 6, we plot the maximum number of elements in the stash over the 1,000 trials. The stash sizes are sufficiently small that it suggests that they will not be a performance issue for our triangle-listing algorithm. We have found that the stash sizes decrease significantly if we allow a larger maximum number of iterations, $L = 500$.

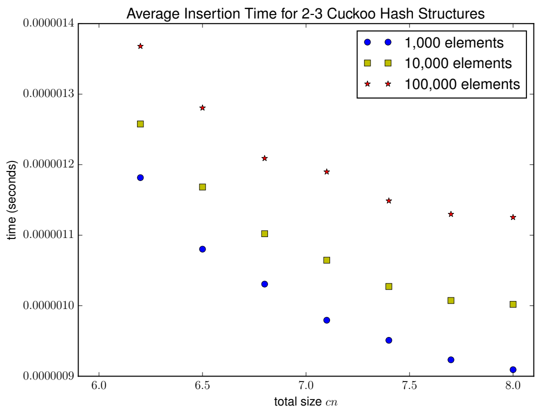

Figure 4: Wall-clock times for the average insertion time into a 2-3 cuckoo hash structure as a function of the size of the 2-3 cuckoo hash tables.

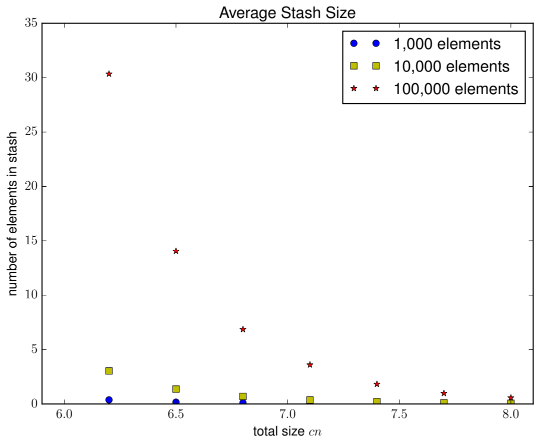

Figure 5: Average size of the stash cache as a function of the size of the 2-3 cuckoo hash tables.

The need for so many iterations to find an empty location suggests that the "random walk" approach is inefficient in this setting. A more efficient alternative approach may be to either use breadth first search on an insertion as in [36], or assign elements to locations in the hash tables in an offline rather than online fashion, using methods such as peeling [29] to quickly place many elements. We plan to explore these options in future implementations.

We now turn to examining our triangle-listing algorithm. To begin, we consider an artificial data set, namely, Erdős-Rényi random graphs with n vertices and edge probability $p = 1/\sqrt{n}$. The choice of p yields sparse enough graphs to make both our algorithm and Chiba and Nishizeki's algorithm significantly faster asymptotically than an algorithm using, say, a direct bitmap structure for its sets (which would be a natural alternative representation in the practical RAM model for sets that are dense relative to the universe size). In Figure 7, we present results for our algorithm for n = 100,000, which shows a typical behavior we see at other sizes. Each data point is the average of three runs. Recall that in this setting, our table size depends on the degeneracy, as

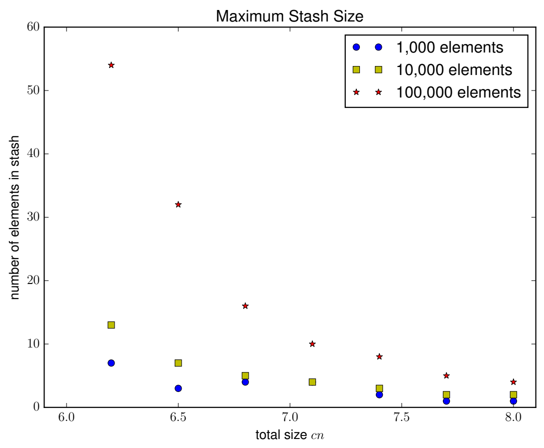

Figure 6: Maximum size of the stash cache as a function of the size of the 2-3 cuckoo hash tables T_i. The elements inserted are a random permutation of the integers starting from 1.

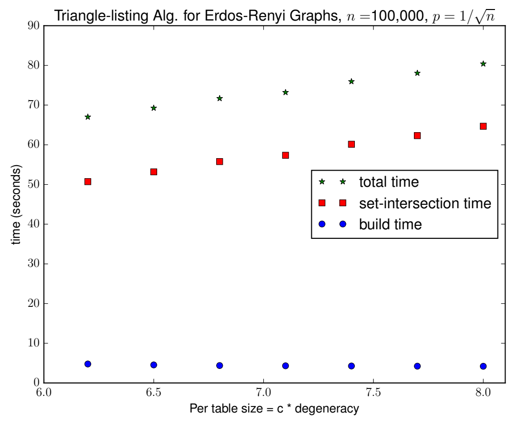

Figure 7: Wall-clock times for our triangle-listing algorithm as a function of the size of the 2-3 cuckoo hash tables for 100,000 node Erdős-Rényi random graphs with $p = 1/\sqrt{n}$.

that determines the maximum sized set of neighbors we must consider for each vertex. We vary the table size as c times the calculated degeneracy. We find that increasing the table size decreases the build time (as expected), but increases the set-intersection time at a faster rate. Thus we find that choosing a c close to 6 is best for our performance.

The time to find the set intersection appears significant, and we believe here our implementation can be improved. Indeed, we note that the set intersection algorithm we provide can naturally be parallelized, which would immediately improve performance.

In Figure 8, we compare the result of our algorithm against the algorithm of Chiba and Nishizeki [13], for Erdős-Rényi random graphs with $p = 1/\sqrt{n}$. The purpose of this exercise is not to claim that one algorithm is necessarily better than the other, but to show that there are families of graphs where our algorithm competes successfully against the state of the art. We use $c = 6.2$, $L = 200$, and $\delta = 8$ in this experiment. As can be seen, our algorithm outperforms the algorithm of Chiba and Nishizeki handily for this graph density, with the improvement increasing with the size of the graph.

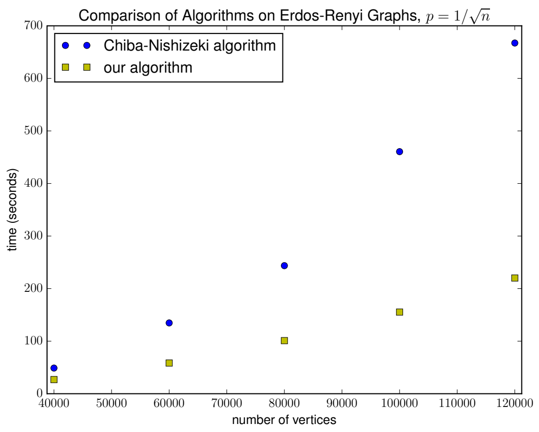

Figure 8: Comparing Chiba and Nishizeki's triangle-listing algorithm with our algorithm on Erdős-Rényi graphs.

We further compare Chiba and Nishizeki's algorithm with our algorithm by running both on a collection of real-world graphs, with the results in Table 1. The graphs used in the comparison are from various domains. We give a name to each graph in Table 1 and we describe each one of them here. "Amazon" is a snapshot of a co-purchasing network of Amazon [54]. "YouTube" is a snapshot of a subgraph of the social network of users on YouTube [42]. "Enron" represents the email communication network of the former Enron corporation [32]. "Astrophysics," "Condensed," and "High Energy" are the collaboration networks from the preprint server arXiv submitted to the astrophysics, condensed matter, and high energy physics categories, respectively [34]. "DBLP" is a collaboration network of the DBLP computer science bibliography [54]. "Skitter," "Oregon 1," and "Oregon 2" are all Internet autonomous systems networks [11, 51]. "Gowalla" and "Brightkite" are location-based online social networks [14]. "California," "Pennsylvania," and "Texas" are road networks [35]. The data for the graphs was gathered from the Stanford Network Analysis Project [49].

For each of the graphs in Table 1, we average the running times in seconds over 10 trials, as reported in the last two columns of the table. The constant c determines the size of the 2-3 cuckoo hash tables: each table is of size c times the calculated degeneracy. For each of the graphs in Table 1, we set $c = 6.2$, $\delta = 7$ and $L = 200$; we found $\delta = 7$ worked slightly better in these experiments. On the real-world graphs tested, our algorithm outperforms Chiba and Nishizeki's algorithm on 5 out of the 15 graphs. We believe that these real-world graphs are generally sparse and have heavy-tailed degree distributions, and these features may generally favor Chiba and Nishizeki's algorithm. In particular, since our tables are based on the graph degeneracy, a large maximum-to-average degree ratio appears less favorable to our algorithm; this explains in part why we outperform Chiba and Nishizeki's algorithm on Erdős-Rényi random graphs, where the vertex degrees are tightly concentrated around their expectation. We believe that additional optimizations may reduce the impact of this issue.

graph	# nodes	# edges	# triangles	degeneracy	our time (sec)	Chiba-Nishizeki time (sec)
Amazon	334,863	925,872	667,129	6	1.315	1.302
Astrophysics	18,772	198,080	1,351,441	56	0.555	0.463
Brightkite	58,228	214,078	494,728	52	0.610	0.413
California	1,965,206	2,766,607	120,676	3	4.165	4.571
Condensed	23,133	93,468	173,361	25	0.173	0.106
DBLP	317,080	1,049,866	2,224,385	113	5.484	1.697
Enron	36,692	183,831	727,044	43	0.452	0.444
Gowalla	196,591	950,327	2,273,138	51	2.735	2.948
High Energy	12,008	118,505	3,358,499	238	0.992	0.388
Oregon1	11,174	23,409	19,894	17	0.041	0.021
Oregon2	11,461	32,730	89,541	31	0.075	0.034
Pennsylvania	1,088,092	1,541,898	67,150	3	2.260	2.491
Skitter	1,696,415	11,095,298	28,769,868	111	51.048	43.282
Texas	1,379,917	1,921,660	82,869	3	2.857	3.154
YouTube	1,134,890	2,987,624	3,056,386	51	9.839	11.230

Table 1: Wall-clock times for our triangle-listing algorithm and Chiba and Nishizeki's algorithm for real-world graphs.

7. EXTERNAL-MEMORY ALGORITHM

In this section, we describe our external-memory algorithm for listing triangles in a large graph, G. Our method follows the general framework described above, but it batches constructions and searches so as to be more I/O efficient. Specifically, our external-memory algorithm is as follows:

1. Compute an approximate degeracy ordering for G, so that each vertex, v, has its edges oriented so that every vertex has $O(\alpha(G))$ outgoing edges. This can be done using $O(\text{sort}(n\,\alpha(G)))$ I/Os, by an algorithm due to Goodrich and Pszona [24].

2. Construct a data structure, $(H(v), F(v))$, of parallel 2-3 cuckoo hash tables and filters for the outgoing adjacencies for each vertex, v. This step amounts to doing two cuckoo insertions for each element (so as to implement the two-three cuckoo hashing paradigm). Goodrich and Mitzenmacher [23] show how to construct a cuckoo table of size N in external memory using $O(\text{sort}(N))$ I/Os. Thus, we can use this algorithm by making two copies of each element and repeating this construction in a batched fashion for all the nodes of G, which requires $O(\text{sort}(n\,\alpha(G)))$ I/Os.

3. For each edge, (u, v), perform an intersection operation for the sets represented by $F(u)$ and $F(v)$, without any culling of false positives. That is, we produce a list, P, of potential triangles, each of which is represented as triple, $(u, v, i_{u,v})$, where $i_{u,v}$ is the index in $F(u)$ and $F(v)$ for a matching fingerprint digest in these two cuckoo filters. We can do this by creating degree(v) copies of each $F(v)$ and sorting this list to bring together the filters that need to be merged and intersected. (Note that the EM model doesn't limit the kinds of operations that are performed in internal memory; hence, we can do these internal-memory computations using the practical-RAM algorithm described above.) Thus, this step takes $O(\text{sort}(m(\alpha(G)\log w)/w))$ I/Os.

4. Remove the false positives from P, depending on the size of $\alpha(G)$, as follows.
   (a) Sort the triples in P lexicographically by first and third coordinates.
   (b) Do a mergesort-type scan of P and the $H(u)$ tables, replacing each triple, $(u, v, i_{u,v})$, with $(u, v, z, i_{u,v})$, where z is the vertex listed at index $i_{u,v}$ of $H(u)$, i.e., at $H(u)[i_{u,v}]$.
   (c) Sort this new list, P, lexicographically by second and fourth coordinates.
   (d) Do a mergesort-type scan of P and the $H(u)$ tables, replacing each tuple, $(u, v, z, i_{u,v})$, with (u, v, z), if z is the vertex listed at index $i_{u,v}$ of $H(v)$, i.e., at $H(v)[i_{u,v}]$. Otherwise, discard this tuple.

   The total number of I/Os for performing this last step is $O(\text{sort}(n\,\alpha(G)) + \text{sort}(k + t))$, where t is the number of false positives. Recall that in our use of cuckoo filters, false positives occur with probability at most $1/w$; hence, the expected value of t is $O(m\,\alpha(G)/w)$. Thus, the expected number of I/Os for this last step is $O(\text{sort}(n\,\alpha(G)) + \text{sort}(m\,\lceil\alpha(G)/w\rceil) + \text{sort}(k))$.

Therefore, the expected number of I/Os for our entire algorithm is $O(\text{sort}(n\,\alpha(G)) + \text{sort}(m\lceil(\alpha(G)\log w)/w\rceil) + \text{sort}(k))$.

THEOREM 5. *Given a graph, G, with n vertices and m edges, one can list all the triangles in G using an expected number of I/Os in the external-memory (EM) model that is $O(\text{sort}(n\,\alpha(G)) + \text{sort}(m\lceil(\alpha(G)\log w)/w\rceil) + \text{sort}(k))$.*

8. CONCLUSION

We have studied new data structures based on 2-3 cuckoo hashing, and we showed how to use these data structures to derive improved algorithms for answering set intersection and triangle listing queries. Open problems include whether our techniques can be extended so as not to require each cuckoo hash-filter be of the same size and whether it is possible to intersect as many as t sets of total size n in expected time $O(\lceil n(\log w)/w\rceil + kt)$, where k is the output size.

Acknowledgments

This article reports on work supported by the DARPA under agreement no. AFRL FA8750-15-2-0092. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government. This work was also supported in part from NSF grants CCF-1228639, CCF-1526631, CCF-1618301, CCF-1616248, CNS-1228598, CCF-1320231, CCF-1535795, and CCF-1563710.

9. REFERENCES

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *C. ACM* 31(9):1116–1127, 1988, doi:10.1145/48529.48535.

[2] S. Albers and T. Hagerup. Improved parallel integer sorting without concurrent writing. *Inf. & Comput.* 136(1):25–51, 1997, doi:10.1006/inco.1997.2632.

[3] R. R. Amossen and R. Pagh. A new data layout for set intersection on GPUs. *IEEE Int. Parallel Distributed Processing Symp. (IPDPS)*, pp. 698-708, 2011, doi:10.1109/IPDPS.2011.71.

[4] A. Andersson, P. B. Miltersen, and M. Thorup. Fusion trees can be implemented with AC^0 instructions only. *Theor. Comput. Sci.* 215(1-2):337–344, 1999, doi:10.1016/S0304-3975(98)00172-8.

[5] M. Aumüller, M. Dietzfelbinger, and P. Woelfel. Explicit and efficient hash families suffice for cuckoo hashing with a stash. *Algorithmica* 70(3):428–456, 2014.

[6] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. *SIAM J. Comput.* 29(1):180–200, 1999, doi:10.1137/S0097539795288490.

[7] C. Berge. *Graphs and Hypergraphs.* North-Holland Mathematical Library 6. North-Holland, 2nd edition, 1976.

[8] P. Bille, A. Pagh, and R. Pagh. Fast evaluation of union-intersection expressions. *Int. Symp. Algorithms and Computation (ISAAC)*, pp. 739–750. Springer, LNCS 4835, 2007, doi:10.1007/978-3-540-77120-3_64.

[9] A. Björklund, R. Pagh, V. V. Williams, and U. Zwick. Listing triangles. *Int. Coll. Automata, Languages, and Programming (ICALP)*, pp. 223–234. Springer, LNCS 8572, 2014, doi:10.1007/978-3-662-43948-7_19.

[10] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics* 1(4):485–509, 2004, doi:10.1080/15427951.2004.10129096.

[11] Center for Applied Internet Data Analysis. Skitter. http://www.caida.org/tools/measurement/skitter/.

[12] G. Chartrand, H. V. Kronk, and C. E. Wall. The point-arboricity of a graph. *Israel J. Math.* 6(2):169–175, 1968, doi:10.1007/BF02760181.

[13] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.* 14(1):210–223, 1985, doi:10.1137/0214017.

[14] E. Cho, S. A. Myers, and J. Leskovec. Friendship and mobility: user movement in location-based social networks. *17th ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining*, pp. 1082–1090, 2011, doi:10.1145/2020408.2020579.

[15] K.-M. Chung, M. Mitzenmacher, and S. P. Vadhan. Why simple hash functions work: Exploiting the entropy in a data stream. *Theory OF Computing* 9:897–945, 2013.

[16] R. Dementiev. *Algorithm Engineering for Large Data Sets.* Ph.D. thesis, Saarland Univ., 2006.

[17] D. Eppstein. Cuckoo filter: Simplification and analysis. *15th Scand. Symp. and Worksh. Algorithm Theory (SWAT)*, pp. 8:1–8:12. Leibniz-Zentrum für Informatik, LIPIcs 53, 2016, doi:10.4230/LIPIcs.SWAT.2016.8.

[18] D. Eppstein and M. T. Goodrich. Straggler identification in round-trip data streams via Newton's identities and invertible Bloom filters. *IEEE Trans. Knowledge and Data Engineering* 23(2):297–306, 2011, doi:10.1109/TKDE.2010.132.

[19] D. Eppstein, M. Löffler, and D. Strash. Listing all maximal cliques in large sparse real-world graphs. *J. Exp. Algorithmics* 18:3.1–3.21, 2013, doi:10.1145/2543629.

[20] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo filter: Practically better than Bloom. *10th ACM Int. Conf. on Emerging Networking Experiments and Technologies (CoNEXT)*, pp. 75–88, 2014, doi:10.1145/2674005.2674994.

[21] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.* 47(3):424–436, 1993, doi:10.1016/0022-0000(93)90040-4.

[22] M. T. Goodrich and M. Mitzenmacher. Invertible Bloom lookup tables. *2011 49th Allerton Conference on Communication, Control, and Computing (Allerton)*, pp. 792–799, 2011, doi:10.1109/Allerton.2011.6120248.

[23] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. *38th Int. Colloq. Automata, Languages and Programming (ICALP)*, pp. 576–587, 2011, doi:10.1007/978-3-642-22012-8_46, http://dx.doi.org/10.1007/978-3-642-22012-8_46.

[24] M. T. Goodrich and P. Pszona. External-memory network analysis algorithms for naturally sparse graphs. *19th European Symp. on Algorithms (ESA)*, pp. 664–676. Springer, LNCS 6942, 2011, doi:10.1007/978-3-642-23719-5_56.

[25] M. T. Goodrich and R. Tamassia. *Algorithm design and applications.* Wiley Publishing, 2015.

[26] X. Hu, M. Qiao, and Y. Tao. Join dependency testing, Loomis-Whitney join, and triangle enumeration. *34th ACM Symp. on Principles of Database Systems (PODS)*, pp. 291–301, 2015, doi:10.1145/2745754.2745768.

[27] X. Hu, Y. Tao, and C.-W. Chung. Massive graph triangulation. *ACM SIGMOD Int. Conf. on Management of Data*, pp. 325–336, 2013, doi:10.1145/2463676.2463704.

[28] X. Hu, Y. Tao, and C.-W. Chung. I/O-efficient algorithms on triangle listing and counting. *ACM Trans. Database Syst.* 39(4):27:1–27:30, 2014, doi:10.1145/2691190.2691193.

[29] J. Jiang, M. Mitzenmacher, and J. Thaler. Parallel peeling algorithms. *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 319–330, 2014.

[30] M. Karoński and T. Łuczak. The phase transition in a random hypergraph. *J. Computational and Applied Mathematics* 142(1):125–135, 2002.

[31] A. Kirsch, M. Mitzenmacher, and U. Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM J. Comput.* 39(4):1543–1561, 2010, doi:10.1137/080728743, MR2580539.

[32] B. Klimt and Y. Yang. Introducing the enron corpus. *1st Conf. Email and Anti-Spam (CEAS)*, 2004, http://www.ceas.cc/papers-2004/168.pdf.

[33] T. Kopelowitz, S. Pettie, and E. Porat. Dynamic set intersection. *14th Symp. on Algorithms and Data Structures (SODA)*, pp. 470–481, 2015,

doi:10.1007/978-3-319-21840-3_39.

[34] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowledge Discovery from Data* 1(1):2, 2007, doi:10.1145/1217299.1217301.

[35] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* 6(1):29–123, 2009, doi:10.1080/15427951.2009.10129177.

[36] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. *Proceedings of the Ninth European Conference on Computer Systems*, p. 27, 2014.

[37] D. R. Lick and A. T. White. k-degenerate graphs. *Canad. J. Math.* 22:1082–1096, 1970, doi:10.4153/CJM-1970-125-1, MR0266812.

[38] P.-S. Loh and R. Pagh. Thresholds for extreme orientability. *Algorithmica* 69(3):522–539, 2014, doi:10.1007/s00453-013-9749-4, MR3201231.

[39] S. Lumetta and M. Mitzenmacher. Using the power of two choices to improve Bloom filters. *Internet Mathematics* 4(1):17–33, 2007, doi:10.1080/15427951.2007.10129136.

[40] D. W. Matula and L. L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM* 30(3):417–427, 1983, doi:10.1145/2402.322385, MR0709826.

[41] P. B. Miltersen. Lower bounds for static dictionaries on RAMs with bit operations but no multiplication. *23rd Int. Colloq. on Automata, Languages and Programming (ICALP)*, pp. 442–453. Springer, LNCS 1099, 1996, doi:10.1007/3-540-61440-0_149.

[42] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. *5th ACM/Usenix Internet Measurement Conference (IMC'07)*, pp. 29–42, 2007, doi:10.1145/1298306.1298311.

[43] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Trans. Parallel and Distributed Systems* 12(10):1094–1104, 2001, doi:10.1109/71.963420.

[44] M. Mitzenmacher, A. W. Richa, and R. Sitaraman. The power of two random choices: A survey of techniques and results. *Handbook of Randomized Computing*, vol. 1, pp. 255–312, 2001.

[45] M. Mitzenmacher and E. Upfal. *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.

[46] M. Ortmann and U. Brandes. Triangle listing algorithms: Back from the diversion. *ACM-SIAM Works. Algorithm Engineering & Experiments (ALENEX)*, pp. 1–8, 2014.

[47] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms* 51(2):122–144, 2004, doi:10.1016/j.jalgor.2003.12.002, MR2050140.

[48] R. Pagh and F. Silvestri. The input/output complexity of triangle enumeration. *33rd ACM Symp. on Principles of Database Systems (PODS)*, pp. 224–233, 2014, doi:10.1145/2594538.2594552.

[49] Stanford University. Stanford network analysis project. https://snap.stanford.edu/data/index.html.

[50] M. Thorup. On AC^0 implementations of fusion trees and atomic heaps. *14th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pp. 699–707, 2003, http://dl.acm.org/citation.cfm?id=644108.644221.

[51] University of Oregon. University of Oregon route views project. http://www.routeviews.org/.

[52] J. S. Vitter. Algorithms and data structures for external memory. *Found. Trends Theor. Comput. Sci.* 2(4):305–474, 2008, doi:10.1561/0400000014.

[53] D. E. Willard. Examining computational geometry, Van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM J. Comput.* 29(3):1030–1049, 2000, doi:10.1137/S0097539797322425.

[54] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems* 42(1):181–213, 2015, doi:10.1007/s10115-013-0693-z, arXiv:1205.6233.