Coloring Embedder: a Memory Efficient Data Structure for Answering Multi-set Query

Tong Yang[†‡], Dongsheng Yang[†], Jie Jiang[†], Siang Gao[†], Bin Cui[†‡], Lei Shi[§], Xiaoming Li[†]
[†]Department of Computer Science, Peking University, China
[‡]National Engineering Laboratory for Big Data Analysis Technology and Application (PKU), China
[§]Institute of Software, CAS, China

***Abstract*—Multi-set query is a fundamental issue in data science. When the sizes of multi-sets are large, exact matching methods like hash tables need too much memory, and they cannot achieve high query speed. Bloom filters are recently used to handle big data query, but they cannot achieve high accuracy when the memory space is tight. In this paper, we propose a new data structure named coloring embedder, which is fast, accurate as well as memory efficient. The insight is to first map elements to a high dimensional space to almost eliminate hashing collisions, and then use a dimensional reduction representation, which is similar to coloring a graph, to save memory. Theoretical proofs and experimental results show that compared to the state-of-the-art, the error rate of the coloring embedder is thousands of times smaller even with much less memory usage, and the query speed of the coloring embedder is about 2 times faster. The source code of coloring embedder is released on Github.**

I. Introduction

A. Background and Motivation

Given k sets $\mathcal{S}_1$, $\mathcal{S}_2$ $\ldots\mathcal{S}_k$ with no intersection and an element e from one of those sets, multi-set query is to query which set e belongs to. The formal definition is as follow.

Multiset query: U is the universe of elements, *i.e.*, $U = \{e_1, e_2, ...e_i..., e_m\}$, where e_i can be a string, an integer, or an IP address. U can be divided into s disjoint sets $S_1, S_2, ..., S_s$, such that $\forall i, j, S_i \cap S_j = \emptyset$, and $S_1 \cup S_2 \cup ... \cup S_s = U$. The membership of e is defined as a function $f : U \mapsto \{1, 2, ..., s\}$, such that $f(e) = i$ if $e \in S_i$, where i is also defined as the set ID of e. For any element $e \in U$, the multi-set query is to retrieve its set ID, which is denoted as $\hat{f}(e)$. Our goal is to design an algorithm for multi-set query, which encodes f into a data structure D, and answer queries based on D. If the answer $\hat{f}(e)$ for querying e is unequal to $f(e)$, we say this query incurs an *error*. In practice, small error is often acceptable, especially in big data scenarios.

Multi-set query is a fundamental problem in computer science. It is involved in many applications, including indexing in data centers [35], distributed file system [5], database indexing [5], data duplication [28], network packets processing [11], [43], [41], and network traffic measurement [12], [39]. Below we give two typical use cases.

Use Case 1: Distributed caching. The most classic distributed caching is the Summary Cache [20]. There are multiple proxy caches, and each proxy keeps a compact summary of the cache content of every other proxy cache. When a cache miss occurs, it first checks all the summaries to see if the request might be hit in other caches, and then sends a query message only to those proxies whose summaries show positive results. This is a typical multi-set query problem. Due to the significance of distributed caching, recent works [44], [46] are still optimizing the performance.

Use Case 2: MAC table query. In data centers, for each incoming packet, the switch needs to query the MAC table to find the outgoing port to forward the packet. A query on a MAC table can be seen as a multi-set query. Each MAC table entry includes a key (MAC address) and a value (port). In a typical MAC table [2], there are around ten thousand entries and tens of ports, while a switch often has limited memory, so it is challenging to support queries at high line-rate [43]. For this challenge, many solutions [43], [33] sacrifice the query accuracy, which means a query may get wrong answer (error). In the case of MAC tables, such errors are allowed, but may incur high time penalty.

Above all, the key metrics of multi-set query are query speed, error rate, and memory usage. High query speed is critical to catch up with the high throughput of query requests. Low error rate is highly desired because the time penalty of error is high. Small memory usage is also important, because cache is usually small, and data structures should be small enough to fit into the cache to achieve fast access speed. Prior works often focus on improving one or two of these three metrics. The design goal of this paper is to optimize all the three metrics at the same time.

B. Prior Arts and Limitations

There are mainly two kinds of solutions for multi-set query: hash table based solutions and Bloom filter based solutions.

Using a hash table is a straightforward solution for the multi-set query problem. We just use elements as keys and the set IDs as the values, and then we can build a hash table for these key-value pairs. Hash table based solutions are accurate but not memory efficient. Traditional hash table based solutions [30] achieve O(1) query speed at the cost of large memory usage, and unbounded query time due to hash collisions. Perfect hashing based solutions [10], [14] sacrifice insertion speed for query speed. They have bounded query time. However, they hardly support fast dynamic update.

Co-primary authors: Tong Yang {yangtongemail@gmail.com} and Dongsheng Yang {yangds@pku.edu.cn}

Another notable hashing scheme is called cuckoo hashing [31]. It achieves fast query speed using relatively small memory and supports slow updating.

A Bloom filter [8] is a compact data structure for membership query problem. It can achieve fast and constant query speed using very small memory, at the cost of sacrificing query accuracy. Many prior work [12], [29], [38], [40] focus on using Bloom filters for multi-set query problem. However, they suffer from a relatively high error rate because of hashing collisions. When an element is fully overlapped with other elements in a Bloom filter, a false positive happens.

In summary, the above two kinds of solutions cannot achieve the design goal of this paper.

C. Proposed Approach

Towards the design goal, we propose a novel data structure, named the *coloring embedder*, which can achieve fast query speed, small memory usage, and almost no error at the same time. Similar to hash table based solutions and Bloom filter based solutions, our coloring embedder is also based on hashing. Before introducing our solution, let us first consider the following scenario: given m elements, we randomly map these elements to $n = cm$ buckets. In this paper, a bucket means a unit in the memory that can store only one element. An element cannot be represented by its bucket if this bucket contains more than one element, and we call such case a collision. It is obvious that many collisions will occur when $c = 1$. To reduce the number of collisions to a considerable level only by hashing, c has to be very large.

The design principle of the coloring embedder is to almost eliminate collisions without increasing memory overhead. There are two challenges to design such a data structure: one is how to map the elements to eliminate collisions, and the other is how to use small memory to store the mapping results. To handle the above challenges, we propose two key techniques. The first one is hyper mapping, and the second one is coloring embedding. We first map all elements to a high dimensional space to almost eliminate hashing collisions, and then we perform dimensional reduction to embed the high dimension space into a low dimension space. We use the terminology of the graph to explain our algorithm. Suppose there are m elements, we first map them to an empty graph with cm nodes and $(cm/2)^2$ edge slots, where c is recommended to be 2.2. Each element is mapped to an edge slot to build an edge, and the set ID is recorded on the edge. Then we embed the graph with $(cm/2)^2$ edge slots into a node vector with cm nodes, while keeping the recorded set IDs of all elements accurate.

In this paper, we propose to use the colors of the nodes to represent the type of the edges, namely coloring embedding. To demonstrate the working principle of coloring embedding, we first consider a simple case where there are only two sets, set 0 and set 1. For convenience, we name edges mapped by elements in set 0 as positive edges, and edges mapped by elements in set 1 as negative edges. The graph is colored according to two `coloring rules`:

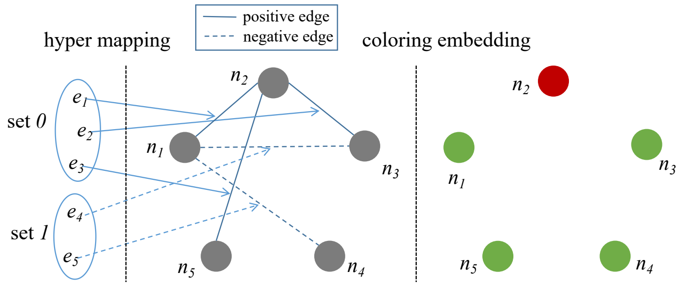


Fig. 1. Hyper mapping and coloring embedding.

1) If there is a positive edge between any two nodes, those two nodes should have different colors;
2) If there is a negative edge between any two nodes, they should have the same color.

If all nodes can be colored according to the two rules, the coloring embedding succeeds. Then we can answer multi-set query only with the vector of node colors. When an element in the sets is queried, we check the two end nodes of its mapped edge. If the two nodes have different colors, the element is in set 0. Otherwise, it is in set 1.

For more than two sets, we do not directly encode them in pairs. On the contrary, we encode the set ID by bits. If there are totally s sets, then the length of the set ID is $\lceil log(s) \rceil$ bits. As mentioned above, one graph can encode two sets, so one graph can encode the content of one bit. Therefore, we can create $\lceil log(s) \rceil$ graphs, and each graph encodes one bit of the binary representation of a set ID. To achieve faster query speed and better load balancing, we further propose the `shifting coloring embedder` which uses only one graph. More details can be seen in Section IV.

II. Background and Related Work

In this section, we first discuss two types of prior art for multi-set query. Then, we briefly introduce the property of random graphs, which is related to our proposed algorithm.

A. Exact-match Data Structures

Exact-match data structures based on hash tables [30] have no error. However, they need to store element keys in order to resolve hash conflicts. To reduce hash collision rate while supporting fast query and update, a large memory is needed. Perfect hashing [18], [7] achieve little memory redundancy, but can hardly support insertions. Another notable hashing scheme is called cuckoo hashing [31]. It achieves fast query speed using relatively small memory and supports slow updating. It maps each element to two positions. If both two positions are occupied by other elements, it expels one of those elements to make room for the new element, and inserts the expelled element to the other position of it. When the load factor is high, the update could fail.

B. Probabilistic Data Structures

Probabilistic data structures for multi-set query are mostly based on Bloom filters. [8] A Bloom filter is a compact data structure to represent a set, and supports approximate membership query, *i.e.*, answering whether an element belongs to the set, but the answer may be wrong. A Bloom filter consists of a bit array and k hash functions which map an element to k bits in the array. To insert an element, k hash functions are computed and all the mapped k bits are set to 1. To query an element, the Bloom filter checks the k mapped bits and returns true if and only if all of them are 1.

To support multi-set query, a straightforward method is to use multiple Bloom filters, each recording one set [43]. But this method has low memory efficiency, and slow query speed because it needs to access multiple Bloom filters. Several work take efforts to reduce the number of Bloom filters, by letting each Bloom filter represent a part of the encoded set IDs, such as Bloom Tree [42], Coded Bloom filter [12], Sparsely coded filter [29], and *etc*. Since the optimal length of a Bloom filter is related to the number of elements, the memory usage of these methods may be influenced by the distribution of set sizes, even if the total number of elements is given. There are also Bloom filter variances which records elements of different sets in a single filter, such as the Combinatorial Bloom filter [22], iSet [33], the Shifting Bloom filter [40], and more [17], [15], [16]. They share an advantage that they are not influenced by the distribution of set sizes.

Bloom filters are suitable for the scenario where the allowed error rate is relatively high. If the allowed error rate is very low (*e.g.*, 10^{-4}), it will need too much memory (19.13 bits per element using 13 hash functions) to reduce the collision rate to meet the requirement. By contrast, our algorithm has the property that if the memory is above a rather small threshold (2.2 bits per element using 2 hash functions), there will be almost no error (smaller than 10^{-4} for 10^5 elements) at all. So our algorithm is more memory efficient if the required error rate is very low.

C. Random Graph and Sharp Threshold

A random graph is generated by randomly connects m pairs of nodes in an empty graph containing n nodes. Random graphs have many elegant mathematical properties, a typical one is the existence of *sharp threshold* [45], [9]. The sharp threshold is also called *phase transition phenomenon*, which means some properties may suddenly change when an independent variable is changed. For example, in nature, water exists in the liquid state if the temperature is over a threshold, and in the solid state if the temperature is under this threshold. Similarly, it has been proved that, cycles exist in a random graph with high probability when m/n is larger than 1, and there are no cycles with high probability when m/n is smaller than 1 [19]. Therefore, 1 is the sharp threshold of the existence of cycles. Many properties in random graphs have phase transition phenomenon, *e.g.*, the emergency of a giant component, the diameter of the graph [19], [23], etc. We have found that there also is a sharp threshold of memory for successful construction of the coloring embedder. The construction will succeed with high probability when the memory size is larger than the threshold. This property can be used for choosing a proper initial memory size for a coloring embedder.

III. The Coloring Embedder

In this section, we describe the design of the coloring embedder in detail. For the ease of understanding, we first present the coloring embedder for two-set query, and then present two variances for multi-set query. Table I summarizes the symbols frequently used in this paper.

TABLE I
Symbols and abbreviations in this paper.

Symbol	**Description**
m	# edges or # elements
n	# nodes or # buckets
e	an element
n/m *ratio*	n divided by m
s	# sets
$h(.)$	hash functions
S^+	the 1^{st} set of elements
S^-	the 2^{nd} set of elements
m_+	# elements in S^+ or # positive edges
m_-	# elements in S^- or # negative edges

A. Rationale

The key idea of the coloring embedder is to *first map all elements to a high dimensional space to avoid hashing collisions, and then perform dimensional reduction to embed the high dimension space into a low dimension space.* There are two steps to construct a coloring embedder, hyper mapping and coloring embedding, as illustrated in Figure 1.

In the hyper mapping process, we first build an empty graph with cm nodes, where m is the number of elements and c is a constant. Then we map each element to an edge slot randomly using hash functions. Since there are about $(cm/2)^2$ edge slots, collisions rarely happen. We record the set IDs on the edges. There are two sets: set 0 ($\mathcal{S}^+$) and set 1 ($\mathcal{S}^-$), so there are two kinds of edges in the graph. The edge with set ID 0 is named as *positive edge*, and the edge with set ID 1 is named as *negative edge*.

In the coloring embedding process, we embed the graph into a node vector by coloring the nodes in the graph. The coloring rules are: 1) for each positive edge, the colors of its two associated nodes should be different; 2) for each negative edge, the colors of its two associated nodes should be the same.

To balance the success rate of coloring embedding and memory usage, we use at most four colors to color the graph. Although a graph can be successfully colored with higher probability if we use more than 4 colors, more bits are required to represent the color of a node, which incurs much more memory overhead. In addition, three colors cannot save memory compared to four colors, because three colors also need 2 bits to encode. Therefore, we choose to use four colors to color the graph.

If the constructed graph has errors, we can either reconstruct the graph using other hash functions or allow the errors to exist because the errors are always quite few. Users can set a maximum number of attempts for reconstruction depending on the expected probability of having no error.

B. Implementation

In this subsection, we describe the data structure and the operations in the coloring embedder, including construction, query, insertion, deletion, and migration.

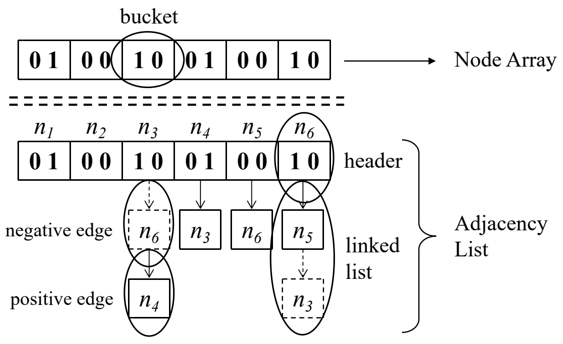


Fig. 2. Structure of the coloring embedder.

1) Data Structure:

The coloring embedder consists of two parts: a node array and an adjacency list. As shown in Figure 2, these two parts can be stored separately because they are used in different situations. Below we introduce them respectively.

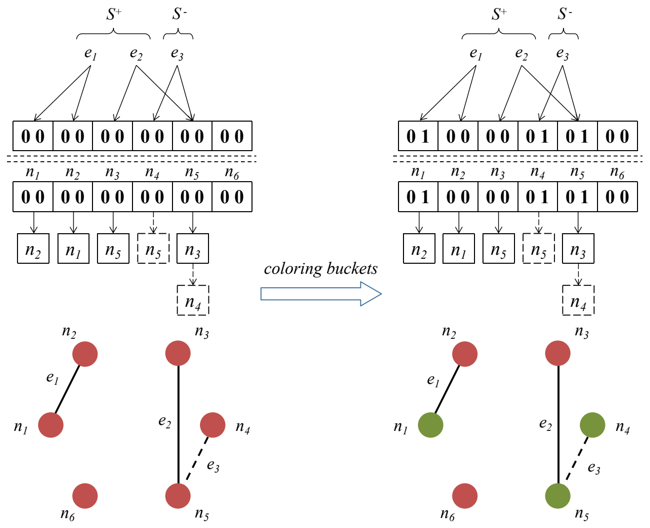
TABLE II
COLOR FOR EACH STATE OF THE BUCKET.

Bits	$(0,0)$	$(0,1)$	$(1,0)$	$(1,1)$
Color	Red	Green	Blue	Yellow

1) Node Array: The node array is used to store the results of the coloring embedding. A node array consists of n buckets, and each bucket consists of two bits denoted by b_1 and b_2. Each bit can be set to 0 or 1, so a bucket has 4 states: (0,0), (0,1), (1,0), and (1,1), corresponding to four colors: red, green, blue, and yellow, respectively (see Table II). A bucket in the node array corresponds to a node in the graph. We define coloring a bucket as setting the values of the two bits in a bucket. For example, if we color a bucket with green, it means setting its first bit to 0 and its second bit to 1.

2) Adjacency List: The adjacency list is used to store the edges of the graph during the hyper mapping process. It is composed of n linked lists, and the header of each linked list corresponds to a bucket in the node array. Let n_i denote the i^{th} bucket. If two nodes in the graph are connected by an edge, the two corresponding buckets in the node array are *logically adjacent*. The linked list of the i^{th} bucket stores the positions of all the buckets that are logically adjacent with bucket n_i. For each item in the linked list, we use a flag bit to indicate whether the edge is positive or negative. In Figure 2, positive edges are represented by solid lines, and negative edges are represented by dash lines. From the adjacency list in Figure 2 we can read that n_3 is logically adjacent to n_6 with a negative edge, and is logically adjacent to n_4 with a positive edge.

2) Operations:



S^+
S^-
e_1 e_2 e_3
00 00 00 00 00 00
n_1 n_2 n_3 n_4 n_5 n_6
00 00 00 00 00 00
n_2 n_1 n_5 n_5 n_3
n_4
coloring buckets
01 00 00 01 01 00
01 00 00 01 01 00
n_2 n_1 n_5 n_5 n_3
n_4
n_2 n_3 e_1 n_1 e_2 n_4 e_3 n_6 n_5

Fig. 3. An example of construction.

Construction: Initially, there is a node array with n buckets and a graph with n nodes and no edge. The i^{th} bucket with two bits corresponds to the i^{th} node with four colors, and we use n_i to denote them both. For each element e in $\mathcal{S}^+$ and $\mathcal{S}^-$, we compute two hash functions to map it to two nodes $n_{h_1(e)}$, $n_{h_2(e)}$, and we create an edge between these two nodes. If the element is in set $\mathcal{S}^+$, the edge is a positive edge, otherwise it is a negative edge. After all elements are inserted, we color the graph to make all nodes obey the coloring rules mentioned in Section I-C. Any coloring algorithm can be used, and we present an algorithm named RDG in Section III-B3. If the graph is colored successfully, we assign the value of bucket n_i with the color of node n_i ($1 \leqslant i \leqslant n$). Otherwise, we change hash functions and repeat construction until it succeed. When the memory size of the node array is larger than 2.2 bits per element so that the number of nodes exceeds the threshold, the construction will succeed in one time with high probability.

`Example (Figure 3):` Set $\mathcal{S}^+$ has two elements e_1 and e_2, and set $\mathcal{S}^-$ has one element e_3. First, every element is mapped to the adjacency list and three logical edges are created. Two of the them are positive and one is negative. The three edges are showed in the corresponding graph below the coloring embedder. Positive edges are represented by solid lines and negative edges are represented by dash lines. Second, we color the nodes. As shown in the graph on the right, n_1 and n_2, n_3 and n_5 are colored with different colors; n_4 and n_5 are colored with the same color. Two colors are enough to color the graph. After that, we set the values of the buckets in both the node array and the adjacency list according to the color of the graph.

Query: The query process only involves the node array. When querying an element e, we compute the two hash functions for e and check the colors of the two mapped buckets $n_{h_1(x)}$ and $n_{h_2(x)}$. If their colors are different, e belongs to $\mathcal{S}^+$. Otherwise, e belongs to $\mathcal{S}^-$.

`Example (Figure 3)`: When querying element e_1, we compute hash functions and get two buckets n_1 and n_2 with values (0,1) and (0,0), respectively. Since these two buckets have different colors, the edge between them is a positive edge. Thus we report that e_1 belongs to $\mathcal{S}^+$.

Insertion: There are two steps to insert an element e. First, we compute the two hash functions and map e to two buckets $n_{h_1(x)}$ and $n_{h_2(x)}$. If e belongs to $\mathcal{S}^+$, we add a positive edge between the two buckets in the adjacency list; if e belongs to $\mathcal{S}^-$, we add a negative edge in the adjacency list. Second, we perform the *RDG updating* algorithm to make all affected buckets follow the coloring rules.

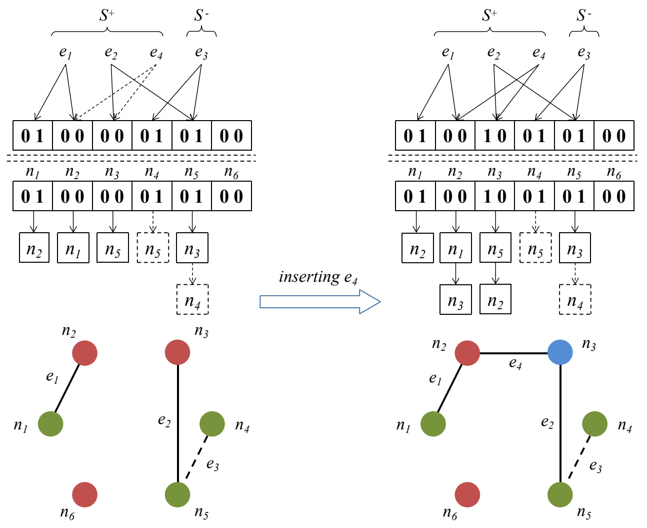


Fig. 4. An example of insertion.

`Example (Figure 4)`: A new element e_4 from $\mathcal{S}^+$ will be inserted. First, we map e_4 to two buckets and add a positive edge between n_2 and n_3. Second, we find out that the colors of bucket n_2 and n_3 are both red, while their colors should be different according to the coloring rules. Therefore, we need to perform the RDG updating algorithm. As a result, the color of bucket n_3 changes from red to blue.

Deletion: To delete an element e, we compute two hash functions to locate the buckets of e, and then remove the edge between $n_{h_1(x)}$ and $n_{h_2(x)}$ from the adjacency list. That means deleting $n_{h_1(x)}$ from the linked list of $n_{h_2(x)}$ and deleting $n_{h_2(x)}$ from the linked list of $n_{h_1(x)}$. The node array does not needed to be modified at once.

Migration: Migration means an element e changes its membership from $\mathcal{S}^+$ to $\mathcal{S}^-$ or vice versa. If e migrates from $\mathcal{S}^+$ to $\mathcal{S}^-$, the edge between $n_{h_1(x)}$ and $n_{h_2(x)}$ changes from positive to negative; if e migrates from $\mathcal{S}^-$ to $\mathcal{S}^+$, the edge changes from negative to positive. Then RDG updating algorithm is performed to color other affected nodes.

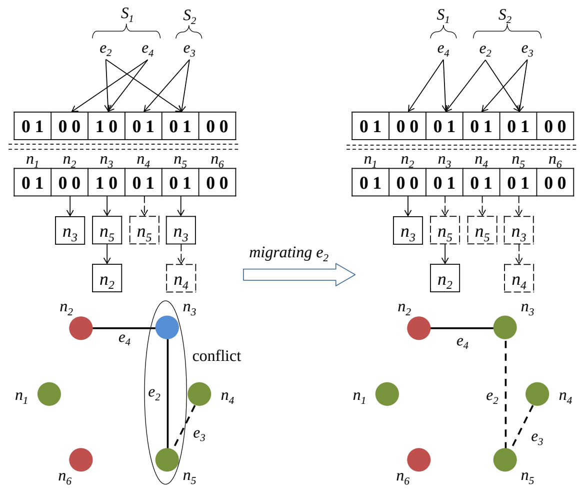


Fig. 5. An example of migration.

`Example (Figure 5)`: Element e_2 changes its membership from $\mathcal{S}^+$ to $\mathcal{S}^-$ and the edge between n_3 and n_5 needs to be changed from positive to negative. As a result, we need to change the colors of n_3 and n_5 to make them have the same color. The RDG updating algorithm is performed, and the color of n_3 changes from blue to green. Other buckets are not affected in this case.

3) *The RDG Coloring Algorithm:*

In this section, we describe our coloring algorithm in details. Coloring problem is a well known NP-hard problem [21], so we cannot give a polynomial time exact algorithm. Our coloring algorithm is named as *Recursively Delete or Give up coloring (RDG)*. It is an extension of the k-core decomposition algorithm in [6]. It is an approximation algorithm and is fast and accurate in practice.

Before going to the algorithm details, we introduce a well known term in graph theory – ***k*-core** [24][34][13]: *The k-core is the maximum subgraph in which the degree of every node is equal or larger than k.* Our RDG algorithm is based on the observation that *the graph will be quickly and successfully colored with k colors if there is no k-core in the graph.*

For convenience, we use **CSG** to represent *Connected SubGraph*. Our RDG coloring algorithm is divided into the following steps:

1) For every pair of nodes directly connected by negative edges, we merge those two nodes to a single node. After that, the graph only contains positive edges. An empty stack is built to record deleted nodes.

2) If all CSGs in the graph have been deleted, go to step 5. Otherwise, for each CSG$_i$ that is still not deleted yet, we compare its `number of nodes` N_{CSG_i} with the predefined threshold θ. If $N_{CSG_i} \leqslant \theta$, go to step 3; If $N_{CSG_i} > \theta$, go to step 4. Typically, we set θ to 16.

3) The incoming CSG is small, so we simply use a depth-first method to color it. If the coloring succeeds, we delete the

CSG and return to step 2. Otherwise, we report that the graph cannot be colored with four colors and the algorithm ends.

4) For the incoming CSG, if there is no node with degrees less than 4, we report that there is a *4-core* and the algorithm terminates. Otherwise, we push all the nodes with degress less than 4 into the stack and delete them from the CSG. After that we return to step 2.

5) We pop all nodes from the stack, and color them one by one. The algorithm ends.

Proof of correctness: Here we prove that if the algorithm reaches the 5^{th} step, the graph can be colored correctly. If coloring a node n_0 leads to conflicts in the the 5^{th} step, there must be more than 4 neighbors of n_0 already colored. However, when n_0 is pushed into the stack, it has less than 4 neighbors remaining in the graph. Therefore, when n_0 is popped, it also has less than 4 neighbors. As a result, we can safely draw the conclusion that all nodes can be colored successfully without conflicts.

Complexity Analysis: In our RDG algorithm, each node enters the stack at most once. The time complexity of processing each node is related to the number of edges the node has. For each edge, it is connected to two nodes so it is processed at most twice. Therefore, the overall time complexity of the construction is $O(n + m)$. We have to store all nodes and edges, along with a stack with at most n elements for k-core decomposition, so the space complexity is $O(n + m)$.

4) RDG Updating algorithm:

Updating refers to inserting or deleting an element into or from $\mathcal{S}^+$ or $\mathcal{S}^-$. For the updating of the coloring embedder, we propose a method named *RDN (Recursively Delete Neighbor)*: When a node n_i needs to change its color, if there is no candidate color for it, we involve all its neighbors into the modification and they make up a subgraph. We attempt to color that subgraph using the RDG algorithm. If the RDG algorithm fails, the neighbors of nodes in the above subgraph are all involved into the subgraph. This process is carried on recursively until a success. If the subgraph cannot be expanded and cannot be colored, a 4-core is found and the RDG updating algorithm fails.

C. Coloring Embedders for More Than Two Sets

To classify more than 2 sets, we propose two solutions. The first one is to apply a coding method and a one memory access scheme to organize multiple coloring embedders together. The second one is to use one large coloring embedder associated with multiple groups of hash functions, and those groups of hash functions are generated by shifting an original group of hash functions.

1) Coded Representation of Sets:

A coded coloring embedder is implemented by multiple coloring embedders. Suppose there are s sets, with IDs ranging from 1 to s. The IDs can be converted to binary codes, with maximum length $\log\lceil s\rceil$. To record the membership of an element, we can record each bit of the set ID binary code with a coloring embedder. This task can be handled by totally $\log\lceil s\rceil$ coloring embedders. If the i^{th} bit is 1, the i^{th} coloring embedder records the element with a positive edge. Otherwise, it records the element with a negative edge. The $\log\lceil s\rceil$ coloring embedders are together called the coded coloring embedder.

Next, we use the One Memory Access Technique to further optimize the query speed of the coded coloring embedder. In the above implementation, the number of memory accesses of a coded coloring embedder is as $\log\lceil s\rceil$ many as a single coloring embedder, which slows down the speed of query, insertion and deletion. To address this problem, we reorganize the layout of the $\log\lceil s\rceil$ embedders. All embedders are separated into single bits and the corresponding bits are put together. The i^{th} bits of each embedder are now in a word, so the binary code of one element can be fetched with only two memory accesses. By using this technique, the coded coloring embedder can work almost as fast as a single coloring embedder.

2) The Shifting Coloring Embedder:

The above coded coloring classifier with one memory access technique can represent more than 2 sets and reduce the number of memory access to 2. However, it uses many coloring classifiers and they suffer from load balancing problem. To address this issue, we propose the `Shifting coloring embedder`, which shares the similar idea with Shifting Bloom filters [40].

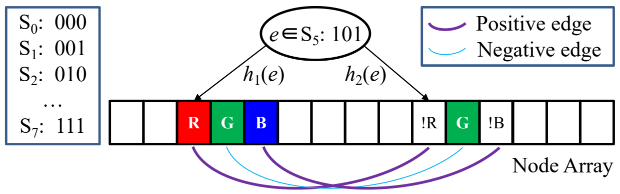


Fig. 6. Shifting coloring embedder.

Given s sets with set ID: 0, 1, 2, ..., $s-1$, we build one shifting coloring embedder. There is only one graph, and $\log_2 s$ edges are inserted in the graph for each element. We use an example in Figure 6 to show how the shifting coloring embedder works. In this example, there are 8 sets, which means $s = 8$ and $\log_2 s = 3$. We assign a code for each set: the code of S_i is i in the binary format. For example, the code of S_5 is 101. When inserting an element e which belongs to set S_5, we compute $h_1(e)$ and $h_2(e)$, and locate $2\log_2 s = 6$ buckets: $n_{h_1(e)}$, $n_{h_1(e)+1}$, $n_{h_1(e)+2}$, and $n_{h_2(e)}$, $n_{h_2(e)+1}$, $n_{h_2(e)+2}$. Because $e \in S_5$ and the code of S_5 is 101. The first bit of this code is 1, it corresponds to a positive edge, we build a positive edge between $n_{h_1(e)}$ and $n_{h_2(e)}$. It means that the colors of $n_{h_1(e)}$ and $n_{h_2(e)}$ need to be different. The second bit of this code is 0, we build an negative edge, and the colors of $n_{h_1(e)+1}$ and $n_{h_2(e)+1}$ need to be the same. Similarly, the colors of $n_{h_1(e)+2}$ and $n_{h_2(e)+2}$ need to be different. When $\log_2 s$ is smaller than the length of a machine word, we can answer multi-set query with only two memory accesses. Also, there is no load balancing problem because there is only one data structure to hold all elements.

IV. Analysis

Two types of errors can occur in our algorithm, which are collision error and color error. Next, we will calculate the expectation of the number of collision errors and the probability that no collision error happens. Then, we will analyze the condition that no coloring error happens. We suppose that there are n buckets in the node array of our data structure, m_+ elements in $\mathcal{S}^+$, and m_- elements in $\mathcal{S}^-$.

A. Collision error

When two edges of different types overlap, a collision error happens. The formal definition is as follow.

Collision error: Given a graph G, a negative connected component is defined as two or more nodes connected by only negative edges. For each negative connected component N^-, if there are two nodes $n_1, n_2 \in N^-$ which are directly connected by a positive edge, a collision error happens.

1) Simple Cases:

The analysis begins with the simplest case of collision error: a positive edge overlaps with a negative edge. To discuss the worst case, we suppose that there are m_+ non-overlapped positive edges and m_- non-overlapped negative edges in the graph. The probability that a negative edge collides with any positive edge is $m_+/\binom{n}{2} = 2m_+/[n(n-1)]$. To calculate the upper bound [25], [27], [26] of the expectation of the number of collision errors, we can directly sum up the probability of collisions for all negative edges because they follow a binomial distribution.

$$E(collision) \leqslant \frac{m_+ m_-}{\binom{n}{2}} = \frac{2m_+ m_-}{n(n-1)} \tag{1}$$

Given a negative edge, the probability that it does not collide with any positive edge is $1 - m_+/\binom{n}{2}$. Collisions are independent events for each negative edge because we suppose they do not overlap with each other, so we can apply the multiplication principle to get the lower bound of the probability that there is no collision error.

$$\begin{aligned} P(no\ collision) &\geqslant \left(1 - \frac{m_+}{\binom{n}{2}}\right)^{m_-} \approx \left(1 - \frac{2m_+}{n^2}\right)^{m_-} \\ &= \left(1 - \frac{2m_+}{n^2}\right)^{\frac{n^2}{2m_+} \times \frac{2m_+}{n^2} m_-} \approx \mathbf{e}^{-\frac{2m_+ m_-}{n^2}} \end{aligned} \tag{2}$$

2) General Cases:

In this section, we analyze a more complex situation of collision error: two nodes are indirectly connected by a list of continuous negative edges, and are at the same time directly connected by a positive edge. For convenience, we name the list of continuous negative edges as an *equivalent negative edge*. To calculate the expectation of the number of collision error, we need to count the number of equivalent negative edges, which is denoted as m'_-.

Our analysis begins with deriving the number of equivalent negative edges formed by two negative edges between three nodes. Given three nodes, the probability that two of them are directly connected by a negative edge is $\frac{3}{n} \times \frac{2}{n} m_-$. And the probability that another pair of nodes is also directly connected by a negative edge is $\frac{2}{n} \times \frac{1}{n}(m_- - 1)$. So the probability that three nodes are connected by two negative edges is

$$\frac{12m_-(m_- - 1)}{n^4}$$

For all the n nodes, the expectation of the number of equivalent negative edges formed by three nodes is then calculated by the following equation.

$$m'_{-(3)} = \frac{12m_-^2}{n^4}\binom{n}{3} \approx \frac{2m_-^2}{n} \tag{3}$$

The number of equivalent negative edges formed by four or more nodes can be similarly derived. Given any v nodes, the probability that they are connected by $v-1$ negative edges is

$$\frac{v!}{2}\left(\frac{2m_-}{n^2}\right)^{v-1}$$

The number of equivalent negative edges formed by v nodes is the product of that value and $\binom{n}{v}$.

$$m'_{-(v)} = \frac{2^{v-2} v! m_-^{v-1}}{n^{2v-2}}\binom{n}{v} \approx \frac{2^{v-2} m_-^{v-1}}{n^{v-2}} \tag{4}$$

From equation 4, we can find that the number of equivalent negative edges is approximate to a geometric progression when the value of v increases. We only show the case that n is larger than $2m_-$. Other cases can be deduced similarly.

$$m'_- = \sum_{v=3}^{n} m'_{2(v)} \approx \sum_{v=3}^{n} \frac{2^{v-2} m_-^{v-1}}{n^{v-2}} \approx \frac{2m_-^2}{n - 2m_-}, \quad n > 2m_- \tag{5}$$

To get the final result of the expectation of the number of collisions and the probability that no collision happens, the m_- In the simplified equation 1 and 2 is replaced by $m_- + m'_-$. n is required to be larger than $2m_-$ in practice. The reason is that if n is smaller than $2m_-$, the graph can hardly be colored successfully.

$$E(collision) \leqslant \frac{2m_+(m_- + m'_-)}{n(n-1)} = \frac{2m_+ m_-}{(n-1)(n-2m_-)} \tag{6}$$

$$P(no\ collision) \approx \mathbf{e}^{-\frac{2}{n(n-1)} m_+(m_- + m'_-)} = \mathbf{e}^{-\frac{2m_+ m_-}{(n-1)(n-2m_-)}} \tag{7}$$

Let `n/m ratio` be the quotient of n divided by m. According to equation 6 and 7, the expectation of the number of collision errors and the probability that no collision error happens are not influenced by the graph size when `n/m ratio` is fixed. When `n/m ratio` is larger than 1.1, which means each element uses more than $1.1 \times 2 = 2.2$ bits, the expectation of the number of collision errors is less than 5 no matter how many elements there are, and the probability that no collision error happens is larger than 50%.

B. Color error

When the graph is dense, 4 colors may be not enough to make all edges in the graph meet the coloring rule. For example, suppose there are 5 nodes in the graph and each pair of nodes is connected by a positive edge, then 5 different colors are required to make all pairs of nodes have different colors to meet the coloring rule. The formal definition of color error is as follows.

Color error: The graph cannot be colored successfully with four colors by the RDG coloring algorithm.

In our RDG coloring algorithm, we give up coloring if we find a 4-core in the graph. As a result, color error happens when there is a 4-core. Theories about k-cores in random graphs are established in [32].
1) If $k \geqslant 3$ and n is large, with high probability, there is a giant k-core when m_+ is larger than $c_k n/2$ and there is no k-core when m_+ is smaller than $c_k n/2$.
2) $c_k = k + \sqrt{k \log k} + O(\log k)$.

According to [32], c_4 is calculated to be *5.14*. The n/m *ratio* threshold for color error is equal to $2/c_4$. Therefore, when there is no negative edge in our graph, the n/m *ratio* threshold is *0.389*. When there are negative edges, our graph is not a random graph and thus the results in [32] do not apply. From the perspective of coloring, negative edges combine many nodes into a large single node because those nodes must have the same color. The large single node has many neighbors, and thus the subgraph containing that node can be very dense, leading to a higher probability of the emergence of a 4-core. As a result, n/m *ratio* threshold becomes larger when the percentage of negative edges is higher. In the worst case, when the negative edges account for 50% of all edges, the n/m *ratio* threshold is *1.10* according to our experiments. In conclusion, we need no more than 1.10×2 =*2.20* bits per element to build a coloring embedder to ensure that no color error happens.

V. Experimental Results

A. Experimental Setup

1) Datasets:

We use three real datasets and generate plenty of synthetic datasets for experiments. The statistics of the real datasets are shown in Table III.

MACTable: This dataset is drawn from the MAC table file in [2]. For each entry in the MAC table, we use the line number as the key, and use the type field (static or dynamic) to determine the set.

MachineLearning: This dataset is drawn from a dataset of classification task of the machine learning [4]. We use the training set as our dataset. For each entry in the training set, we use the line number as the key, and the label as the class.

DBLP: This dataset is drawn from DBLP[1]. We use the `key` attribute as our key. We use the records of articles as $\mathcal{S}^+$ and the records of inproceedings as $\mathcal{S}^-$.

TABLE III
Statistics of the real datasets.

	# items	m^+	m^-	$\mathcal{S}^-$ ratio
MACTable	3664	3144	520	0.1419
MachineLearning	912969	472605	440364	0.4823
DBLP	823132	623212	199920	0.2429

Synthetic dataset: We generate random strings as keys of elements in a dataset. We use synthetic datasets because our data structure have to be examined when the percentage of $\mathcal{S}^-$ is continuously changing, while real world datasets have fixed percentage of $\mathcal{S}^-$. We argue that for data structures using hash functions, including the coloring embedder, real datasets and synthetic datasets have no difference. The experiments in the next section also prove this fact.

2) The State-of-the-art Implementation:

To compare our data structure with the state-of-the-art, we implement three Bloom filter based data structures used for multi-set query. The first one is the Multiple Bloom filter [43]. The Multiple Bloom filter simply assembles the Bloom filters, each one representing one of the sets. This model is called MultiBF in short. The second data structure is the Coded Bloom filter [12], denoted as CodedBF. It is a typical variance of Bloom filter using multiple filters. It converts set IDs to binary codes and stores the code in the Bloom filters. The third one is the shifting Bloom filter [40], denoted as ShiftBF. It is a typical variance of Bloom filter using a single filter. ShiftBF uses the offset of bits to represent the set ID. To make those data structures comparable with ours, we allocate 2.5 times as much memory for those three data structures as the coloring embedder. The source code of coloring embedder is released on Github [3].

3) Experimental Setups:

We use general-purposed CPU to run all experiments, because we do not have FPGA or ASIC environment. We conduct all experiments on a standard off-the-shelf computer equipped with two 6-core Intel(R) Xeon(R) E5-2620 CPUs @2.00GHz and 62GB RAM running Ubuntu 16.04. For each core, the L1 data cache is 64KB and the L2 cache is 256KB.

B. Experiments on Two Sets

In this section, we conduct experiments on two-set query, which is the foundation of multi-set query. We use real datasets and synthetic datasets to comprehensively evaluate performance of hyper mapping and coloring embedding, and measure the throughput of construction, query and insertion.

1) Coloring Embedding:

First, we show that there is a sharp threshold for successful coloring embedding. Then, we test the condition of successful coloring embedding when the percentage of $\mathcal{S}^-$ varies.

Successful coloring rate vs. n/m *ratio* **(Figure 8):** *The experimental results show that there is a sharp threshold of*

n/m ratio for the success rate of coloring embedding. In this experiment, we test the success rate against the n/m *ratio* on all three real datasets. For each real dataset, with the same percentage of $\mathcal{S}^-$, we generate synthetic datasets of different sizes, varies from 10^3 to 10^6. The results are shown in Figure 8. As the *n/m ratio* increases, there is an almost-zero success rate when the *n/m ratio* is below the threshold, a similar surge when the *n/m ratio* is passing the threshold, and an almost-one success rate when the *n/m ratio* is above the threshold. The thresholds are 0.52, 1.07, and 0.66 for datasets of MACTable, MachineLearning, and DBLP, respectively. The threshold of synthetic datasets is the same with that of real datasets [37], [36], [47]. The larger the datasets are, the sharper the threshold is. The threshold for different real datasets are different, because the percentage of $\mathcal{S}^-$ is different. MACTable dataset has the smallest *n/m ratio* threshold because it has the smallest percentage of $\mathcal{S}^-$. There is no sharp threshold for small datasets in Figure 8(b), because they have different properties from large datasets.

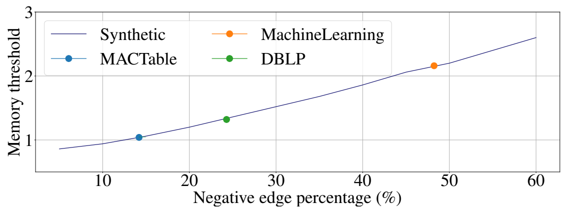


Fig. 7. Memory needed vs. percentage of $\mathcal{S}^-$.

Memory needed vs. Percentage of $\mathcal{S}^-$ (Figure 7): *The experimental results show that the memory needed for coloring embedding increases when the percentage of $\mathcal{S}^-$ increases.* We measure the memory usage (bits per element) in the condition that the successful coloring rate is above 99%. The three real datasets are displayed as points in the figure, while the synthetic datasets are displayed as a line. When the percentages of $\mathcal{S}^-$ is around 13%, the memory needed is below *1* bit per element.

When the percentages of $\mathcal{S}^-$ is around 50%, which is the worst case of our algorithm, the memory needed is *2.2m* bits, where m is the number of elements. When the memory size is larger than 2.2 bits per element, the graph is sparse enough so that there is no 4-core and thus can be colored successfully with 4 colors. *2.2* bits per element is always enough for all kinds of datasets because when the percentages of $\mathcal{S}^-$ is larger than 50%, we can simply exchange $\mathcal{S}^-$ with $\mathcal{S}^+$.

2) Hyper Mapping:

In this part, we evaluate the number and probability of edge collisions during hyper mapping under different settings of n/m *ratio* and the percentage of $\mathcal{S}^-$. We use synthetic datasets with sizes from 10^3 to 10^6. By default, the percentage of $\mathcal{S}^-$ is 50%, and the n/m *ratio* is *1.1*, which is the threshold of successful coloring.

Number of collisions vs. n/m *ratio* (Figure 9(a)): *The experimental results show that the average number of collisions decreases when the n/m ratio increases.* Specifically, when the n/m *ratio* is *1.4*, the expectations of the number of collisions of all datasets are 1. The number of collisions is not influenced by dataset sizes when the n/m *ratio* is above *1.15*. The experimental results fit well with the theory.

Probability of collisions vs. n/m *ratio* (Figure 9(b)): *The experimental results show that the probability that collisions happen decreases when n/m ratio increases.* The probability that collisions happen is not influenced by the set size. When the n/m *ratio* is *1.3*, the probability that collision happens is about 75%. And when the n/m *ratio* is *1.5*, the probability is 50%. The experimental results fit well with the theory.

Number of collisions vs. Percentage of $\mathcal{S}^+$ (Figure 9(c)): *The experimental results show that the number of collisions decreases when percentage of $\mathcal{S}^+$ increases.* In this experiment, we change the percentage of $\mathcal{S}^+$ from 45% to 100%, and fix the n/m *ratio* to *1.1*. From Figure 9(c) we can see that when $\mathcal{S}^+$ accounts for more than 50%, there are less than 2 collisions for datasets of all sizes. When there is no negative edge, there is no collision. The experimental results fit well with the theory.

Probability of collisions vs. Percentage of $\mathcal{S}^+$ (Figure 9(d)): *The experimental results show that the probability that collisions happen decreases when percentage of $\mathcal{S}^+$ increases.* The probability is almost not influenced by the size of datasets. It decreases almost linearly from about 90% to 0% when percentage of $\mathcal{S}^+$ increases from 50% to 100%. The experimental results fit well with the theory.

3) Throughput of Construction, Insertion and Query:

Throughput of construction vs. Dataset size (Figure 10(a)): *The experimental results show that the throughput of construction decreases slightly when the order of magnitude of dataset size increases.* In this experiment, the percentage of each of the two sets is fixed to 50%. The memory usage is *2.21* bits per element, which is the memory threshold for successful coloring. When the size of datasets increases from 10^3 to 10^7, the construction speed falls from around 1.6 million operations per second (MOPS) to around 0.4 MOPS.

Throughput of query vs. Dataset size (Figure 10(b)): *The experimental results show that our coloring embedder has up to 90 MOPS query speed.* In this experiment, we use the same settings with the previous one. Figure 10(b) shows that the query speed of the coloring embedder is 90 MOPS when the dataset size is 10^3, and is 35 MOPS when the size is 10^6.

Throughput of insertion vs. Dataset size (Figure 11): *The experimental results show that the throughput of insertion is high when the load rate is below a threshold.* In this experiment we first construct an empty coloring embedder using 2.21×10^6 bits, and then insert 10^6 elements into it. Figure 11(a) shows that when we insert less than 65% elements into the coloring embedder, few nodes are affected by the RDG updating algorithm. In contrast, when we insert more than 65% elements, tens of thousand nodes need to be recolored. Figure

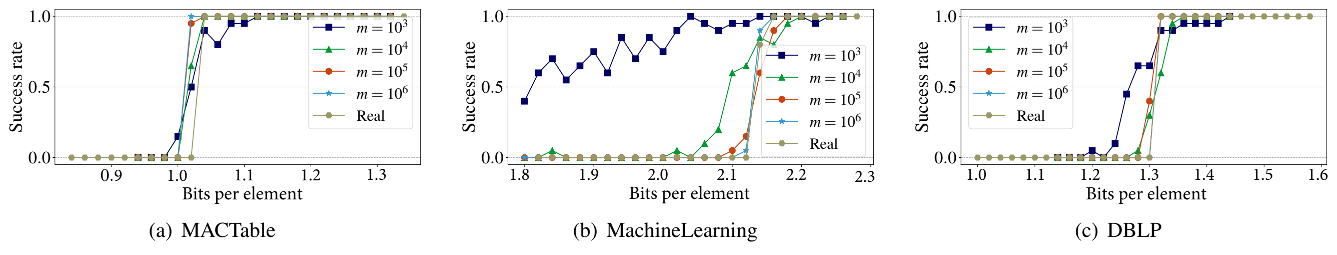


Fig. 8. Successful coloring rate vs. n/m ratio.

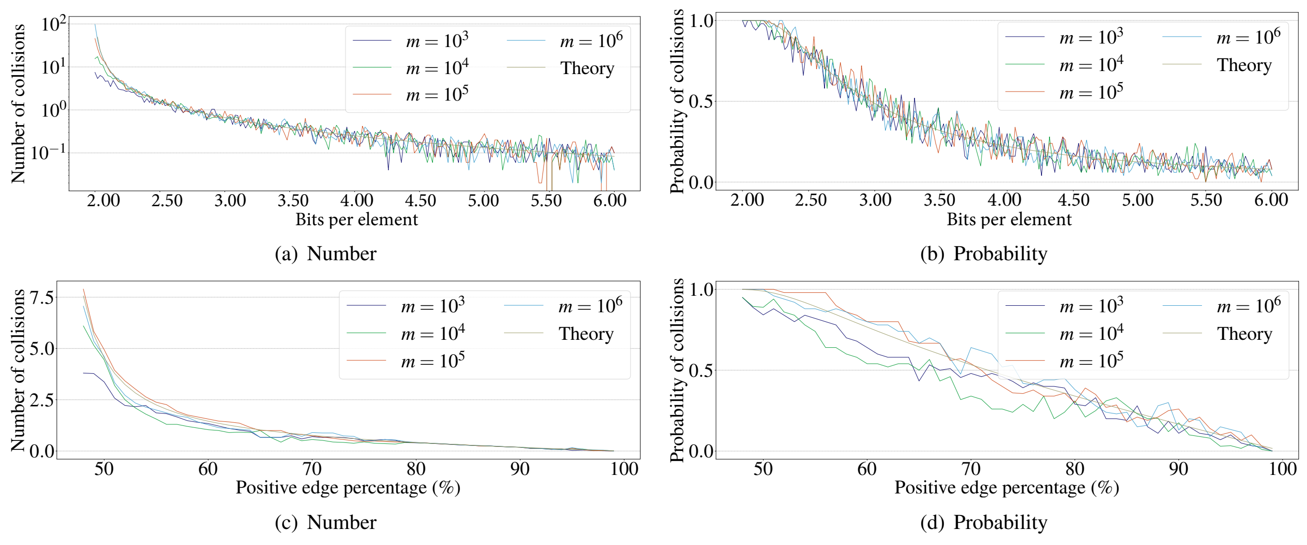


Fig. 9. Number and probability of edge collisions.

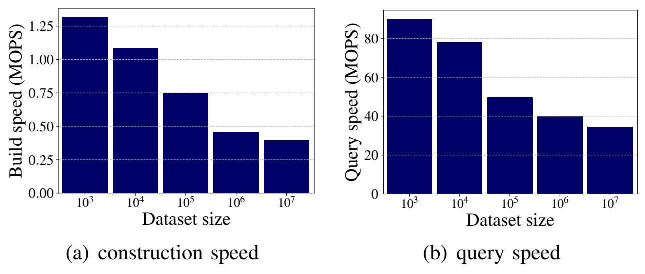


Fig. 10. Construction and query speed.

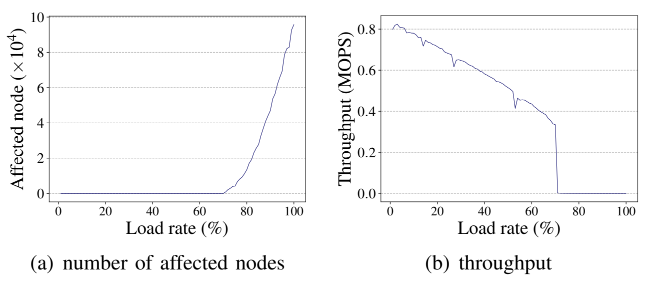


Fig. 11. Insertion speed vs. Load rate.

11(b) shows that the insertion speed decreases gradually when we insert less than 65% elements, and drops sharply when we insert more than 65% elements.

C. Experiments on Multi-sets

In this section, we compare shifting coloring embedder with Bloom filter variances on multi-set query. We use synthesis datasets to test the worst case of our coloring embedder (each sets has the same size). First we fix the number of sets to 16 and vary the dataset size from 10^3 to 10^6. Then we fix the set size to 10^6 and vary the number of sets from 2 to 16. When comparing error number and query speed, the Bloom filter variances use 2.5 times memory as much as the shifting coloring embedder to achieve an comparable accuracy for plotting.

Throughput of query vs. Dataset size (Figure 12): *The experimental results show that our shifting coloring embedder has faster query speed compared with the state-of-the-art for 16-set query.* The query speed of the shifting coloring embedder is around 80 MOPS when there are 10^3 elements. And the query speed drops to 30 MOPS when the number of elements increases to 10^6. The query speed of other data structures is always less than 40 MOPS.

Number of errors vs. Dataset size (Figure 13): *The experimental results show that our shifting coloring embedder has fewer errors compared with the state-of-the-art for 16-set query.* The number of errors of the shifting coloring embedder is around 5, not influenced by the size of datasets. On the contrary, the number of errors of Bloom filter variances is proportional to the dataset size, and are larger than the shifting coloring embedder when there are more than $10k$ elements.

Memory vs. dataset size (Figure 14): *The experimental results show that the shifting coloring embedder uses the least memory for different dataset sizes.* The number of sets is fixed to 16 and the number of errors is limited under 10. When varying the dataset size from 10^3 to 10^6, CodedBf, MultiBF, ShiftBF and our algorithm uses 24.0 to 51.6, 14.8 to 29.8, 14.8 to 29.8 and 11.2 to 8.8 bits per element memory, respectively.

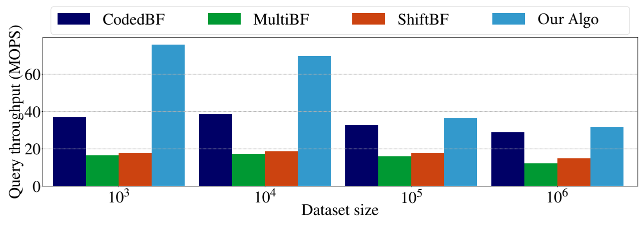


Fig. 12. Query Speed vs. Dataset size.

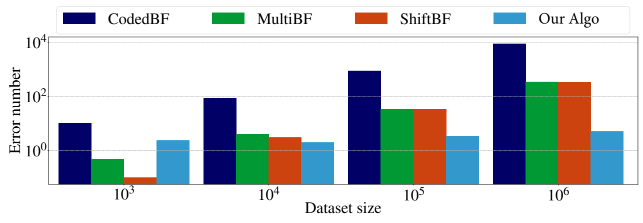


Fig. 13. Number of errors vs. Dataset size.

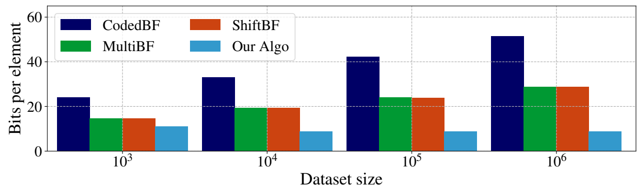


Fig. 14. Memory vs. Dataset size.

Throughput of query vs. Number of sets (Figure 15): *The experimental results show that the shifting coloring embedder has the fastest query speed for different number of sets.* We change the number of sets from 2 to 16 and test query speed on 10^6 elements. The query speed of the shifting coloring embedder is around 35 MOPS, and is almost not influenced by set number thanks to the shifting technique. On the contrary, the query speed of Bloom filter variances drops steadily, and is lower than the query speed of the shifting coloring embedder when there are more than 2 sets.

Number of error vs. Number of sets (Figure 16): *The experimental results show that the shifting coloring embedder has the fewest errors for different number of sets.* Bloom filters have 10^4 to 10^2 times more errors than the shifting coloring embedder. When the number of sets changes, the number of errors of the shifting coloring embedder is not influenced, staying below 10. The number of errors of MultiBF and ShiftBF decreases when set number increases, because the size of a single filter becomes larger for them. However, they always have thousands of errors.

Memory vs. Number of sets (Figure 17): *The experimental results show that the shifting coloring embedder uses the least memory when varying the number of sets.* The dataset size is fixed to 10^6 and number of errors is limited under 10. When varying the number of sets from 2 to 16, CodedBF, MultiBF, ShiftBF and our algorithm uses 11.5 to 51.6, 24.7 to 29.8, 24.8 to 29.8 and 2.2 to 8.8 bits per element memory, respectively. Our algorithm saves up to 90% memory.

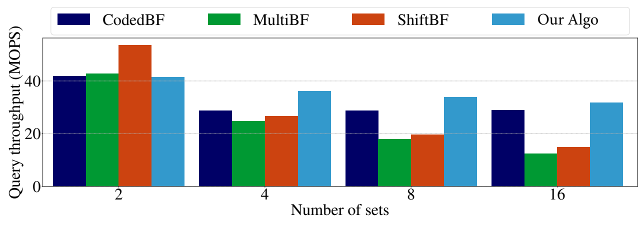


Fig. 15. Query Speed vs. Number of sets.

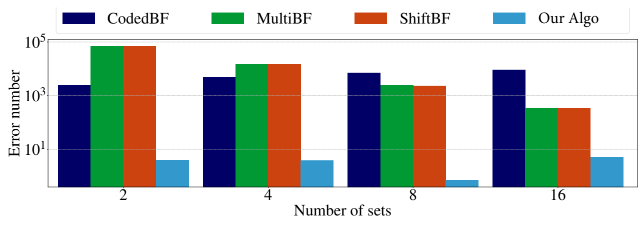


Fig. 16. Number of errors vs. Number of sets.

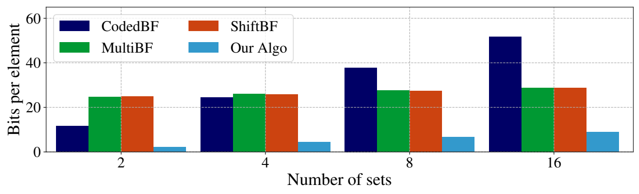


Fig. 17. Memory vs. Number of sets.

VI. Conclusion

In this paper, we propose a novel data structure named coloring embedder. The coloring embedder is used for two-set query, and a shifting model is designed for the coloring embedder to support multi-set query. Experimental results show that our coloring embedder can achieve up to 10^4 times smaller error rate than the state-of-the-art, even with only 40% memory of the latter. Specifically, it has less than 5 errors on data sets containing 10^7 elements with only $2.2 \log s$ bits per element memory in the worst case, where s is the number of sets. In addition, the coloring embedder achieves about 2 times faster query speed than the state-of-the-art because it always requires only 2 memory accesses for each query. The source code of coloring embedder is released on Github [3]. We believe that the insight of hyper mapping and coloring embedding can be applied to design more data structures.

Acknowledgement

We would like to thank the anonymous reviewers for their thoughtful suggestions. This work is partially supported by Primary Research & Development Plan of China (2018YFB1004403, 2016YFB1000304), and NSFC (61672061).

References

[1] dblp: computer science bibliography. http://dblp.org/xml/release/dblp-2017-09-03.xml.gz.

[2] Hassel library. https://bitbucket.org/peymank/hassel-public.

[3] The source code of coloring embedder. https://github.com/4colorclassifier/4colorclassifier.

[4] Youtube comedy slam preference data data set. https://archive.ics.uci.edu/ml/datasets/YouTube+Comedy+Slam +Preference+Data.

[5] M. K. Aguilera, W. M. Golab, and M. A. Shah. A practical scalable distributed b-tree. *Proceedings of the VLDB Endowment*, 1(1):598–609, 2008.

[6] V. Batagelj and M. Zaversnik. An o(m) algorithm for cores decomposition of networks. *Computer Science*, 1(6):34–37, 2003.

[7] D. Belazzougui, F. C. Botelho, and M. Dietzfelbinger. Hash, displace, and compress. *Lecture Notes in Computer Science*, 5757:682–693, 2009.

[8] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[9] B. Bollobs, S. Janson, and O. Riordan. The phase transition in inhomogeneous random graphs. *Random Structures & Algorithms*, 31(1):3–122, 2010.

[10] F. C. Botelho, Y. Kohayakawa, and N. Ziviani. *A Practical Minimal Perfect Hashing Method*. Springer Berlin Heidelberg, 2005.

[11] W. Bux, W. E. Denzel, T. Engbersen, A. Herkersdorf, and R. P. Luijten. Technologies and building blocks for fast packet forwarding. *IEEE Communications Magazine*, 39(1):70–77, 2001.

[12] F. Chang, W.-c. Feng, and K. Li. Approximate caches for packet classification. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 4, pages 2196–2207. IEEE, 2004.

[13] J. Cheng, Y. Ke, S. Chu, and M. T. Ozsu. Efficient core decomposition in massive networks. In *IEEE International Conference on Data Engineering*, pages 51–62, 2011.

[14] Z. J. Czech, G. Havas, and B. S. Majewski. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters*, 43(5):257–264, 1992.

[15] H. Dai, L. Meng, and A. X. Liu. Finding persistent items in distributed, datasets. In *Proc. IEEE INFOCOM*, 2018.

[16] H. Dai, M. Shahzad, A. X. Liu, and Y. Zhong. Finding persistent items in data streams. *Proceedings of the VLDB Endowment*, 10(4):289–300, 2016.

[17] H. Dai, Y. Zhong, A. X. Liu, W. Wang, and M. Li. Noisy bloom filters for multi-set membership testing. In *Proc. ACM SIGMETRICS*, pages 139–151, 2016.

[18] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. M. A. D. Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: upper and lower bounds. In *Foundations of Computer Science, 1988., Symposium on*, pages 524–531, 1988.

[19] R. Durrett. *Random graph dynamics*, volume 200. Cambridge university press Cambridge, 2007.

[20] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM ToN*, 8(3):281–293, 2000.

[21] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to NP-Completeness*. W. H. Freeman, 1979.

[22] F. Hao, M. Kodialam, T. V. Lakshman, and H. Song. Fast dynamic multiple-set membership testing using combinatorial bloom filters. *IEEE/ACM Transactions on Networking*, 20(1):295–304, 2012.

[23] S. Janson and M. J. Luczak. A simple solution to the k -core problem. *Random Structures & Algorithms*, 30(1-2):5062, 2007.

[24] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo. K-core decomposition of large networks on a single pc. *Proceedings of the VLDB Endowment*, 9(1):13–23, 2015.

[25] Z. Li, B. Chang, S. Wang, A. Liu, F. Zeng, and G. Luo. Dynamic compressive wide-band spectrum sensing based on channel energy reconstruction in cognitive internet of things. *IEEE Transactions on Industrial Informatics*, 2018.

[26] Z. Li, Y. Liu, A. Liu, S. Wang, and H. Liu. Minimizing convergecast time and energy consumption in green internet of things. *IEEE Transactions on Emerging Topics in Computing*, 2018.

[27] Z. Li, F. Xiao, S. Wang, T. Pei, and J. Li. Achievable rate maximization for cognitive hybrid satellite-terrestrial networks with af-relays. *IEEE Journal on Selected Areas in Communications*, 36(2):304–313, 2018.

[28] G. Lu, Y. J. Nam, and D. H. Du. Bloomstore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–11. IEEE, 2012.

[29] Y. Lu, B. Prabhakar, and F. Bonomi. Bloom filters: Design innovations and novel applications. (1):201–206, 2005.

[30] W. D. Maurer and T. G. Lewis. Hash table methods. *ACM Computing Surveys (CSUR)*, 7(1):5–19, 1975.

[31] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[32] B. Pittel, J. Spencer, and N. Wormald. Sudden emergence of a giant k -core in a random graph. *Journal of Combinatorial Theory*, 67(1):111–151, 1996.

[33] Y. Qiao, S. Chen, Z. Mo, and M. Yoon. When bloom filters are no longer compact: Multi-set membership lookup for network applications. *IEEE/ACM Transactions on Networking*, 24(6):3326–3339, 2016.

[34] A. E. Saryce, B. Gedik, G. Jacques-Silva, K. L. Wu, and mit V. atalyrek. Incremental k -core decomposition: algorithms and evaluation. *VLDB Journal*, 25(3):425–447, 2016.

[35] S. Wu, D. Jiang, B. C. Ooi, and K.-L. Wu. Efficient b-tree based indexing for cloud data processing. *Proceedings of the VLDB Endowment*, 3(1-2):1207–1218, 2010.

[36] F. Xiao, L. Chen, C. Sha, L. Sun, R. Wang, A. X. Liu, and F. Ahmed. Noise tolerant localization for sensor networks. *IEEE/ACM Transactions on Networking*, 26(4):1701–1714, 2018.

[37] F. Xiao, Z. Wang, N. Ye, R. Wang, and X.-Y. Li. One more tag enables fine-grained rfid localization and tracking. *IEEE/ACM Transactions on Networking (TON)*, 26(1):161–174, 2018.

[38] D. Yang, D. Tian, J. Gong, S. Gao, T. Yang, and X. Li. Difference bloom filter: A probabilistic structure for multi-set membership query. In *IEEE International Conference on Communications*, pages 1–6, 2017.

[39] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proc. ACM SIGCOMM 2018*, pages 561–575.

[40] T. Yang, A. X. Liu, M. Shahzad, and et al. A shifting bloom filter framework for set queries. *Proceedings of the VLDB Endowment*, 9(5):408–419, 2016.

[41] T. Yang, G. Xie, Y. Li, Q. Fu, A. X. Liu, Q. Li, and L. Mathy. Guarantee ip lookup performance with fib explosion. In *Proc. ACM SIGCOMM 2014*, volume 44, pages 39–50.

[42] M. K. Yoon, J. W. Son, and S. H. Shin. Bloom tree: A search tree based on bloom filters for multiple-set membership testing. In *INFOCOM, 2014 Proceedings IEEE*, pages 1429–1437, 2014.

[43] M. Yu, A. Fabrikant, and J. Rexford. Buffalo: bloom filter forwarding architecture for large organizations. In *ACM Conference on Emerging NETWORKING Experiments and Technology, CONEXT 2009, Rome, Italy, December*, pages 313–324, 2009.

[44] V. Zakhary, D. Agrawal, and A. E. Abbadi. Caching at the web scale. *Proceedings of the VLDB Endowment*, 10(12):2002–2005, 2017.

[45] L. Zdeborov and F. Krzakaa. Phase transitions in the coloring of random graphs. *Physical Review E Statistical Nonlinear & Soft Matter Physics*, 76(3 Pt 1):031131, 2007.

[46] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment*, 8(11):1226–1237, 2015.

[47] H. Zhu, F. Xiao, L. Sun, R. Wang, and P. Yang. R-ttwd: Robust device-free through-the-wall detection of moving human with wifi. *IEEE Journal on Selected Areas in Communications*, 35(5):1090–1103, 2017.